

The Development and Evaluation of an Intelligent Supervisory System for Process Control

Andrzej Korpala

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, 1991

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Andrew Clarke
28th day of June 1991

Abstract

As industrial plants become more complex, there is a growing need for new approaches to control and supervision. This research investigates the issues involved in applying Artificial Intelligence (AI) techniques in the real-world engineering problem of process control supervision. Current AI theory is examined and some techniques modified to design a general-purpose, reactive planner. The planner forms the basis of a supervisory control system. The system is implemented and interfaced with an existing laboratory plant, so that its performance can be tested and evaluated by comparing with a conventional feedback controller. This real-life testing necessitates explicit treatment of issues such as data sampling, situation assessment and CPU scheduling. The case study shows that by combining AI techniques with conventional control, a system can be built which displays superior performance under normal operating conditions and which can deal with abnormal conditions such as equipment failures.

In memory of my cousin and best friend

Andrzej Zajac

1958-1989

Acknowledgements

I would like to thank Professor Ian MacLeod for his encouragement and guidance, and for creating a relaxed but creative and productive atmosphere in which this research was conducted.

Funding for this research was provided mainly by the Foundation for Research Development and in part by a Process Control Engineering Research Grant and a University Senior Bursary. They are gratefully acknowledged here.

Very special thanks go to my parents.

Contents

1	Introduction	1
1.1	Background	1
1.2	Statement of the Problem	2
1.3	Literature Survey	2
1.3.1	Reconfigurable Control	2
1.3.2	Knowledge and its Representation	3
1.3.3	Planning	6
1.4	Overview of the Dissertation	9
2	Supervision in Process Control	12
2.1	Supervision in the Context of Control Systems	12
2.1.1	Limitations of Control Systems	12
2.1.2	The Role of Supervision	13
2.1.3	Supervisory Tasks	14
2.2	Automatic Process Control	15
2.3	Automatic Supervision	20

2.3.1	Supervisory Objectives	20
2.3.2	Supervisory Actions	21
2.4	Conclusion	23
3	Introduction to Planning	24
3.1	Approaches to Problem Solving	24
3.1.1	Expert Problem Solvers	24
3.1.2	General Problem Solvers	25
3.2	Definition of Planning	27
3.3	Basic Search Strategies	29
3.4	World and Action Representation Methods	31
3.5	Advanced Planning Strategies	35
3.5.1	Reactive Planning	36
3.5.2	Hierarchical Planning	39
3.5.3	Planning with Learning Abilities	40
3.5.4	Opportunistic Planning	41
3.6	Advanced Representation Methods	41
3.6.1	Temporal Logic	41
3.7	Conclusion	42
4	Design of the Planner	43
4.1	Proposed Representation	44

4.1.1	World Representation	44
4.1.2	Action Representation	45
4.2	Proposed Search Strategy	46
4.3	Organization and Processing of Knowledge	48
4.3.1	Data Structures	48
4.3.2	Planning Process	50
4.4	Planning Example	52
4.4.1	Problem Definition	52
4.4.2	Problem Representation	53
4.4.3	Solution	54
4.5	Conclusion	56
5	Design of the Supervisor	57
5.1	Overall System Architecture	57
5.1.1	Read and Write Modules	57
5.1.2	Monitoring Module	58
5.1.3	Decision-Making Module	60
5.1.4	Action Executing Module	62
5.1.5	Structure of the Supervisor	63
5.2	Run-Time Requirements of the System	64
5.2.1	Scheduling of the CPU Time	64
5.2.2	Round-Robin Non-Preemptive System Executive	64

5.2.3	Multitasking Requirements of the Supervisor	66
5.2.4	The Multitasking Kernel	67
5.3	Conclusion	70
6	Outline of Implementation	71
6.1	Data Structures	71
6.1.1	Global Data Structures for Intertask Communication	71
6.1.2	Other Global Data Structures	73
6.1.3	Local Data Structures	73
6.2	Top-Down Analysis of the Software Routines	74
6.2.1	Actions	74
6.2.2	Monitor	76
6.2.3	Read	78
6.2.4	Write	79
6.2.5	Plan	80
6.2.6	Task Initialization Routines	87
6.3	Conclusion	89
7	Case Study	91
7.1	The Laboratory Plant	91
7.2	Control and Supervisory Objectives	93
7.3	The Supervisory System	95

7.3.1	Monitoring	95
7.3.2	Supervisory Actions	97
7.3.3	Other Tasks	106
7.4	Test Results	106
7.4.1	User Interface	106
7.4.2	Standard PI Control	108
7.4.3	Supervisory Control	109
7.5	Conclusion	114
8	Conclusions	116
8.1	AI in Control	116
8.2	Approach to Planning	117
8.3	Software Implementation	117
8.4	Experimental Results	118
8.5	Future Scenarios	118
A	Data definitions	119
A.1	Sup-defs.c	119
A.2	Plan-def.c	123
A.3	PI-co-df.c	124
B	Supervisory tasks and actions	128
B.1	Sup-acts.c	128

B.2 Sup-fcns.c 144

B.3 Sup-ifac.t 151

C Planning task

C.1 Sup-plan.c 158

C.2 Plan-fcn.c 159

C.3 Pl-co-fc.c 174

C.4 Pl-su-fc.c 180

D Additional code

184

D.1 Sup-list.c 184

D.2 Sup-kern.c 185

D.3 Sup-adac.c 187

List of Figures

1.1	Layout of the dissertation.	10
3.1	Depth-first search in a suitable space.	30
3.2	Breadth-first search in a suitable space.	31
3.3	An open-loop control system.	37
3.4	A closed-loop, feedback control system.	37
3.5	An open-loop planning system.	38
3.6	A closed-loop, reactive planning system.	38
4.1	Logical structure of an operator.	45
4.2	Data structure for storing operators.	48
4.3	The stack data structure.	49
4.4	Currently explored search path.	50
4.5	A single stack item.	50
4.6	The Nickel and Dime Problem.	53
5.1	Basic structure of the supervisory system.	58
5.2	The read and write module.	58

5.3	Detailed structure of the supervisory system.	63
5.4	Round-robin non-preemptive scheduling using the system executive.	66
5.5	An example application using the multitasking kernel.	69
7.1	The plant and the supervisory system.	92
7.2	Symbolic representation of the level in tank.	97
7.3	Standard PI control.	108
7.4	High gain PID control.	109
7.5	Supervisory control: initial plan.	110
7.6	Supervisory control: second plan.	111
7.7	Supervisory control: overload.	112
7.8	Supervisory control: valve failure.	113
7.9	Supervisory control: controller failure.	114
7.10	Supervisory control: sensor failure.	115

Chapter 1

Introduction

1.1 Background

Over the years, control theory has become increasingly advanced and complex. Conventional and modern control methods are based on solid engineering concepts and use very sophisticated tools from well-established sciences such as mathematics and physics. Most of these methods have proved themselves unequivocally in real-world conditions.

However, automatic feedback controllers, no matter how sophisticated and reliable, have to be supervised. Supervision represents those activities that take place above the low-level control loops and whose objectives are to improve the performance of the system and ensure its safe operation. As control systems become more complex, this need for supervision increases.

Control supervision is still mainly the responsibility of plant operators and engineers. Shifting this responsibility onto automatic systems could have a great economic impact (by increasing consistency and reliability, while lowering costs). The difficulties encountered in automating supervision are in direct contrast with the natural and straightforward approach to supervision exhibited by humans. This suggests that intelligence could be the major ingredient of successful supervision.

Some commercial control packages include a limited amount of supervision, typically in the form of an automatic start-up or shut-down sequence or an optimizing algorithm used to manipulate the setpoints to low-level controllers. However, existing

implementations of control supervisors fall short of providing truly automatic supervision. Indeed, the task of automatic supervision is so complex, that the tools used in most implementations—mathematical algorithms and preprogrammed logic—are not sufficiently powerful. Therefore, new, more sophisticated tools are being investigated and some promising ones come from the domain of Artificial Intelligence.

1.2 Statement of the Problem

Artificial Intelligence (AI) is a relatively new science, ambitious and promising, but controversial and often misunderstood. In the short space of about forty years, it has produced some valuable techniques which resemble certain aspects of human intelligence. These techniques are very computationally intensive and often perceived as too inefficient to be used for real-world problems (e.g., in control). Even if they are applied, they are typically used in off-line, advisory systems, limited to a specific problem.

Application of the AI technique of planning to create an intelligent supervisory system for process control, is the subject of this research. A supervisory system is designed, implemented and then tested on a real plant, to determine the validity of this approach. The purpose of this research is not to devise a new AI technique, nor to develop new control theory, but to show how the two fields can be brought together using a pragmatic approach. The objective is to show that by incorporating some intelligence in the system, truly automatic supervision can be achieved. Therefore, a real-world case study is central to this research.

1.3 Literature Survey

1.3.1 Reconfigurable Control

Adaptive control in the form of on-line, auto-tuning devices has been around for some time. This section summarizes recent developments in the area, which attempt to produce truly adaptive systems, particularly the idea of a controller capable of changing its control configuration according to the circumstances.

While there are some proposals of expert-systems exercising direct control over a plant, they are not pursued here. This stems from the belief that the conventional

PID algorithm with its firm theoretical foundations and widespread acceptance in the processing industry, is a very competent tool for low-level control. Indeed, the most frequently mentioned criticism lies in the fact that PID controllers are often not adequately tuned.

Åström, in [Åström et al., 89], describes an intelligent feedback controller which takes over the task of the instrument engineer in addition to that of the operator. It is capable of choosing a control algorithm and then monitoring its performance, to determine how effective that control scheme is. An example of such control would be a PID controller using the standard Ziegler-Nichols method for auto-tuning, but one that could also automatically switch between P, PI or PID algorithm and, if necessary, indicate if a different control configuration should be used (e.g., feedforward compensation, dead-time compensation, Smith predictor, pole placement etc.).

In [Verbruggen and Åström, 89] the ability of a system to change its control configuration is viewed as a higher level of automation. It requires a large body of heuristic knowledge about the process dynamics, specifications, objectives, implementation constraints, disturbances etc., to enable it to choose the best control strategy. Thus, not only are the dynamics of the process taken into account, but more importantly, the purpose of control.

1.3.2 Knowledge and its Representation

There is little doubt that the concept of knowledge representation is in a sense at the root of Artificial Intelligence and perhaps intelligence in general. There are many different approaches to this problem and they lead to different AI techniques. Therefore, questions such as—what is the difference between data and knowledge; how to extract, classify and manipulate knowledge—should be answered before committing to a particular AI method.

Traditional AI, which employs computational methods, adopts symbolic representation, while there is no localized, explicit symbolic representation in neural networks [Pao, 88]. For this reason, neural networks are seen as more suitable for perception-cognition tasks as their sub-symbolic, parallel distributed processing can deal with large amounts of continuously varying data (such as in real-world systems). Also, their auto-associative and generalization properties could make them ideal for knowledge acquisition, while their speed of convergence does not depend on the physical size (important for complex, real-time systems). On the other hand, symbolic ap-

proach of traditional AI seems better suited for high level reasoning.

An example of integrating a neural network and a traditional AI method is presented in [Handelman *et al.*, 88]. A CMAC (Cerebellar Model Articulation Controller) network with its implicit knowledge encoding is used as the reflexive component (i.e., reacting automatically, without the need for inference), whereas the declarative component is implemented by an explicit knowledge encoding, i.e., a rule-based system. This integrated system illustrates how declarative reasoning is used when a new situation is encountered, but soon becomes reflexive, through repetition.

In the context of a control system there are two fundamentally different types of knowledge. One of them is essentially static knowledge about the plant, control system, objectives etc. The other is dynamic and it captures the current state of the system, environment etc., by analysing the sensor data.

Knowledge used in the entire control life-cycle is analysed in [Shadbol: *et al.*, 89]. Basically, it contains different abstractions of the process requirements, a model of the process and elements of control theory. These are needed in all three stages: design (i.e., choosing a control algorithm to match process characteristics and objectives), commissioning (i.e., tuning the chosen algorithm to achieve required performance) and supervision (i.e., monitoring the behaviour of the resulting control system and the process, and if necessary calling for recommissioning or redesign).

This knowledge is further divided into three types: generic aspects (e.g., overall objective, choice of measurements, control parameters, speed, accuracy, robustness and cost of control etc.), domain context (e.g., ranges of parameters, overshoot value, lag value etc.) and control theory context (e.g., characteristics of P, PI and PID controllers). Although the supervision stage does not necessarily require deep knowledge of underlying principles (as exemplified by most human operators), its inclusion enhances diagnostic and fault correction capabilities.

The most popular medium for storing and manipulating static knowledge is a rule-based expert system. If **IF-THEN** rules are used for encoding expert knowledge and one of the available inference methods used for chaining rules to produce some action. This type of expert system is termed first-generation [Simmonds, 88], where its limitations are considered. Serious problems can arise when such systems are interfaced with the real world characterized by incomplete knowledge, non-monotonic reasoning and real-time constraints.

While rules are ideal for representing causal relationships, they are not well suited

for other kinds of relationships (e.g., temporal or spatial). The type of logical inference used (rule chaining, heuristic search) can be successfully applied to analytical tasks such as diagnosis, but is totally inappropriate for synthesis (as in planning or design).

There is also the fundamental shortcoming that if a rule chain is not afforded enough time to execute, there will be no decision or action at all. Thus, real-time requirements are something of an afterthought, addressed by limiting rule lengths and decreasing execution time. Finally, there is generally no integration among individual rules, which leads to an unstructured, homogeneous body of rules—a serious problem for large and complex systems, particularly from the point of view of maintenance.

A second-generation expert system is proposed, which could reason with real knowledge. Such knowledge would be characterized by the following attributes: breadth (related branches), depth (governing principles), certainty and richness (different forms). It would be represented as a framework of hierarchies, with sub-classes inheriting knowledge from their parent classes (more generic frames), giving rise to vertically oriented shells. Also, the importance of a model-based reasoning for creative tasks of planning and design is emphasized.

The difficulty with dynamic knowledge of the state of a system lies in its extraction from a multitude of continuous data. As noticed in [Voss, 88], one needs to construct a mapping between such low-level data and the information needed by the expert system. Such mapping should overcome the abstraction gap which exists between the level directly accessible by the traditional control structure and that of the knowledge-based system by, for instance, transforming numerical data into qualitative descriptions and time stamps into temporal relations. The very need for an expert system follows from the fact that not all low-level signal combinations can be immediately associated with a control action.

The abstraction of numerical data into qualitative domain has been studied and used by researchers of qualitative reasoning techniques, for example, [Kuipers, 86]. Typically, the magnitude of a variable is quantized into positive, negative or zero categories; the change in a variable is described as increasing, decreasing or steady. Sometimes other landmark values may be used (a particular value may have far more significance than zero, depending on the problem). Simple arithmetic operations are shown capable of manipulating these qualitative representations and complete simulations are possible.

1.3.3 Planning

The pioneering work in planning was characterized by the choice of highly predictable worlds as the application domain. This allowed for construction of the entire plan prior to its execution and no need for monitoring, as the plan was inherently static. The obvious inability of such plans to deal with very unpredictable and quickly changing real-world situations was soon realized, resulting in planning systems with sensory feedback, interleaving of planning and execution and most recently reactive planning (i.e., planning at execution time).

One of the main problems of the first general problem solvers is addressed by Sacerdoti in [Sacerdoti, 74], namely the combinatorial expansion of search spaces. In his approach, attention to detail in problem domain representation is viewed as the major reason for this behavior. To reduce it, abstraction spaces are introduced. They are simplified representations of the problem space, without unimportant details.

This strategy markedly reduces the problem search space (therefore improving efficiency) as compared with earlier methods, due to the elimination of work on fruitless branches of the search tree.

It is also recognized that complex, real-world problems are characterized by large numbers of outcomes for each operation. This means an exponential growth in the hypothetically possible states of the world being considered by a traditional problem solver. An interaction of planning and execution is suggested, through alternatively advancing plan construction, followed by its partial execution. To compensate for serious differences between the predicted model and the actual state of the world, replanning would be necessary.

Using a temporal logic to represent knowledge in a planning system is described in [Allen and Koomen, 83]. This overcomes some of the limitations of earlier problem solvers, using less adequate world models. Simple representations in which actions and goals must be instantaneous and sequential, while changes can only be caused by the planner, are not applicable to the majority of problems. By introducing interval-based representation and some simple relations, traditional problem solving methods can be used for a much wider range of problems.

Planning looks for facts which are required to hold, but have no causal explanations, i.e., goals. Once these causal gaps are found, the planner proceeds to eliminate them by proposing new actions. This is repeated in a hierarchical fashion, one level of

abstraction at a time. The temporal representation used, allows far more general descriptions of actions and goals which are no longer limited to instances, but may take time, for example, action ordering allows for overlapping of different actions, in addition to actions following one another or occurring simultaneously.

The implementation presented suffers from an exponential growth in constraint propagation, as new constraints are added automatically, based on the structure of the domain and the defined relations. Another unsolved area is that of past and future events, particularly the inability to reason about changing future events.

Planning as a time dependent function is discussed in [Dean and Boddy, 88]. The traditional approach of computer science has been to focus on the correctness of a particular algorithm and accept that it needs certain time to be computed. If the computational capability is looked at as a function of time, four different classes can be identified, as shown in Figure 1. Three of them are characterized by a gradual improvement, while one shows a one-shot improvement (step function).

This last function is typical of most existing planners. If given enough time it produces a correct answer, otherwise some predetermined default action has to be taken by the system. When the time available for making decisions is highly variable and sometimes very short (such as in real-time applications), these methods generally fail. Planners belonging to one of the other classes should be developed, ones that provide the best answers they can in the time available. Therefore, if necessary, they could be terminated at any time.

Fixed, detailed plans generated before execution are limited to structured, artificial domains. Reactive planning for dynamic worlds requires that some or even all of it is performed during execution. This idea is recognized in [Firby, 87], where purely reactive planning is investigated to determine its advantages and shortcomings. There, the world state is constantly monitored during execution, so that a direct assessment of the situation can be made and immediately used for planning. It is assumed that no anticipation at all is possible and autonomous entities are used to pursue different goals in parallel.

The world model update is automatic through corrective feedback and inconsistencies, which arise as a result of uncertainty in the world model, are dealt with by replanning (using the updated, consistent model). Such purely reactive planner proves to be very tolerant in the presence of uncertain knowledge, through its quick adaptation. However, building plan decisions purely on the basis of the current world state often becomes ineffective and can even lead to dangerous situations, as

they cannot be identified and prevented in advance. Therefore, a kind of integrated strategic planner with look-ahead capabilities should not be discarded.

Reactivity in reasoning and planning is seen as necessary for survival in dynamic, uncertain worlds in [Georjeff and Lansky, 87]. They list the following major problems of typical planning systems. Often most information needed in planning is only acquired during plan execution and not before; search based techniques are time consuming and not suitable for real-time applications. Traditional planning does not take into account the compromise between time available for planning and for execution; it lacks the procedural knowledge about how to function. The necessity to take notice of, and to react to critical changes not only during plan execution, but also during plan formation is ignored.

The proposed remedy is to construct a system which, at any time, considers its actions depending on the current goals, beliefs and intentions. Goals or desires are described as sequences of desired world states (i.e., desired behaviours), which use simple temporal operators. Beliefs express the current facts about the world, obtained from observation or which are predefined. Intentions correspond to a state of the stack of knowledge areas (procedures for achieving goals).

These knowledge areas contain procedural information, as well as triggering conditions which make them react in response to a certain goal or data. The task of the reasoner is to choose the best knowledge area for given beliefs and goals. Beliefs, goals and intentions can all be manipulated and modified by certain procedures.

Although such architecture allows a complete change of focus in reasoning and displays reactive behaviour, it is not really planning in terms of predefined actions. In [Kaelin-Lang, 88] it is described as 'run-time interpretation of highly conditional non-specified plans'. This is due to the fact that the procedures which describe how to achieve different goals have to be predefined.

An extreme case of reactive planning, called *activity* rather than *planning*, is presented in [Agre and Chapman, 87]. The major objective is to deal with highly complex, uncertain and unpredictable situations (an example of an arcade video game is used) without the problem of combinatorial explosion. Actions are used as continual improvisation in response to the current situation, using simple opportunistic patterns of interaction, called routines.

They are neither plans nor procedures, but rather one-step activities which become applicable under certain conditions. Therefore, their computational requirements

are very low. This approach does not need an explicit model of the world nor the typical knowledge representation methods which involve binding constants to variables. The agent only has to focus on that region which is currently important from its point of view. Representation of objects is totally relative to the agent situation and purpose, in contrast to the standard absolute representation.

This type of activity is envisaged for situations where events are moving too quickly for any planning to be possible, or even for all the necessary data to be collected and analysed.

1.4 Overview of the Dissertation

The first three chapters (1-3) of this dissertation are introductory in nature. The next three chapters (4-5) describe the design part of the project. Chapter 7 covers a case study and the last chapter (8) contains concluding remarks. Two important issues, namely planning in the context of AI and supervision in the context of control, have to be developed simultaneously. Therefore, there is a slight discontinuity in going from Chapter 2 to Chapter 3. Both of these issues then converge in Chapter 5 to form a cohesive system. Figure 1.1 shows how the chapters are integrated in this dissertation.

A summary of each of the remaining chapters is given below.

Chapter 2

This chapter is entitled *Supervision in Process Control*. It introduces the concept of supervision against the background of control and, in particular, process control. It illustrates how the conventional and modern control methods lead to increasing levels of sophistication, culminating in intelligent systems. The reasons behind supervision are explained and the tasks and actions involved in supervising process control systems are defined.

Chapter 3

This chapter, entitled *Introduction to Planning*, leaves aside supervision and control, to concentrate on some of the most successful AI techniques. The emphasis is on the

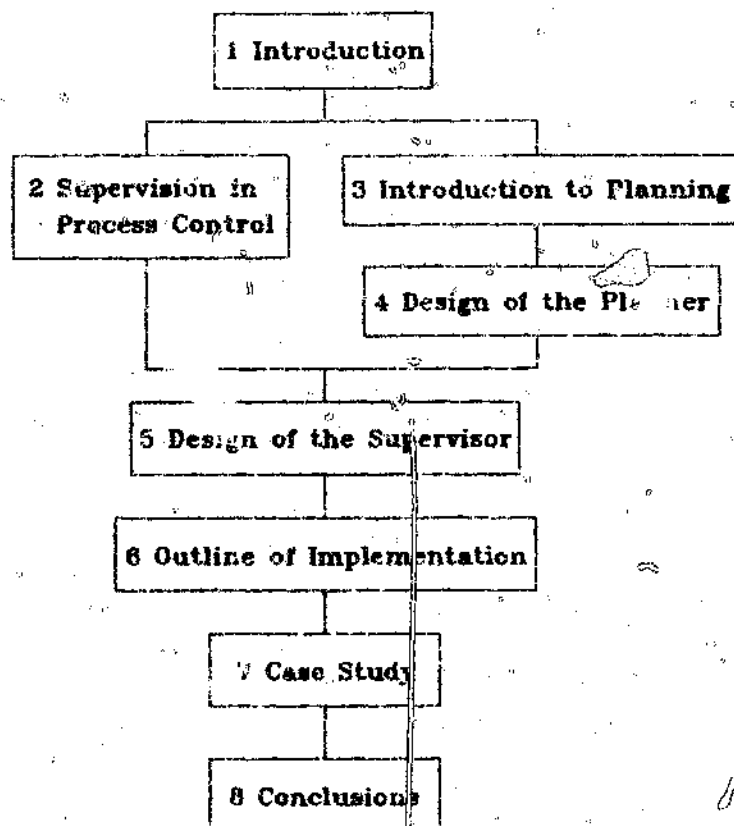


Figure 1.1: Layout of the dissertation.

existing planning methods and their advantages and disadvantages when compared with one another. This discussion includes some of the latest developments in the field and provides the background information on which Chapter 4 is built.

Chapter 4

Chapter 4 is entitled *Design of the Planner*. It shows the development of our proposed AI part of the system, from theoretical concepts, down to a description of the search algorithm. To make the issues involved in the design clearer, a simple example is given at the end of this chapter. It is intended to illustrate the planning process only and has nothing to do with supervision. In fact, it comes from the game playing domain, to show the flexibility and general-purpose character of the proposed planner.

Chapter 5

This is a very important chapter, entitled *Design of the Supervisor*. It brings together the ideas developed in the earlier chapters. It presents the development of our proposed intelligent supervisor, by surrounding the AI part with other modules to create a real-time system. Therefore, it deals with the issues of data communication, monitoring, diagnosing and action execution. Other very important issues discussed in this chapter are the run-time requirements of the supervisor and the multitasking kernel used to enable concurrent execution.

Chapter 6

This chapter, entitled *Outline of Implementation*, describes the program which implements the intelligent supervisory system. Thus, the discussion is from the software point of view, although it is kept free from any specific computer language (in this case, the C language). Firstly, the most important data structures are described and then, a top-down explanation of the most important routines follows. The routines which are absolutely central to the entire program are described in terms of their inputs, outputs and their algorithms.

Chapter 7

This is the chapter on *Case Study*. This part of the dissertation is very important as all the concepts developed in the earlier chapters are put to the test. The chapter starts by introducing the laboratory plant used in the experiment and by explaining the objectives set up before the test. Because some parts of the supervisory system are domain-dependent, this chapter describes them in detail. Finally, the test results are presented in a form which should make comparisons easy.

Chapter 8

The last chapter provides a summary of the major findings made during this research

Chapter 2

Supervision in Process Control

This chapter discusses the concepts of supervision and intelligent supervision, and their role in control systems. This is done against the background of conventional and modern automatic control methods.

2.1 Supervision in the Context of Control Systems

2.1.1 Limitations of Control Systems

We can define the general aim of control as that of *achieving a desired response from the controlled plant or process, operating under changing conditions*. However, in reality, present control methods fall a little short of this target. The major pitfall follows from the fact that the changes in the plant or environment characteristics they cater for, are not truly *arbitrary*. They are limited to parameter changes (not arbitrarily large either) and generally not capable of responding to structural changes.

This is an inherent characteristic of all low-level (direct) control systems and a consequence of their design. The mathematical and engineering tools used during the design process require a mathematical model of the plant in the form of its transfer function. Parameter variations within the transfer function correspond to changing operating conditions, but certain structural assumptions are always built into these transfer functions. Once a parameter goes beyond its anticipated range or the very structure of the plant or the control system itself undergoes a change,

the initial assumptions used at the design stage may become invalid, rendering the system inadequate.

The amount of the initial knowledge of the plant or process varies with the different control techniques used, for instance, *feedforward control* can only be used if disturbances are predictable and measurable. *Feedback control* is typically used where large and unpredictable disturbances can be expected. *Optimizing control* assumes far better knowledge of the operating conditions and their effects (i.e., a more accurate model), but provided the assumptions are met, it rewards with optimal (or close to optimal) control. *Adaptive control* tries to bridge that gap, by realizing that higher quality of control needs a more accurate model of the plant, but that obtaining such a model is not always possible at the design stage and may have to be done on-line. Adaptive control can respond to plant parameter variations, but the basic model of the plant must still be known in advance.

Generally, to achieve better quality of control, the plant and its environment need to be modelled more accurately and, inevitably, more assumptions must be made along the way. Such a control system runs a higher risk of finding itself in a set of circumstances it was not designed for and which contradicts some of its assumptions. This goes against the trend to more robust controllers, which can deal with the majority of expected and unexpected problems.

2.1.2 The Role of Supervision

In the end, all control systems need supervision. It may be lower priority and occasional, like in many industrial applications running detuned PI controllers in the interest of greater robustness, rather than out-and-out performance. On the other hand, there are some processes and plants which may require accurate control and, at the same time, stay within the limits of safe operation at all times. Such systems depend heavily on close and effective supervision.

Control involves constant checking and regulating, whereas supervision involves occasional inspecting and directing. A controller tries to achieve a desired response unwittingly assuming its own competence. A supervisor is then necessary to ensure that the operating conditions are indeed within the scope of the controller's capabilities and if not, that alternative action is taken. An action such as this would typically be concerned with maintaining some performance level or at least maintaining a minimum safety margin.

3.5.3 Planning with Learning Abilities

Just as reactive and hierarchical planning can be observed in human problem solving, so can utilizing past experience. It is not often that a radically new plan has to be generated. Most of the time, a basic plan for each task can form a good starting point and it can then be modified or extended depending on the need. Such approach eliminates the wasteful effect of generating identical or very similar plans more than once. The consequent increase in efficiency for cases previously encountered can be very significant, although a more complex database and search strategy would be needed to store and recall these 'template' plans.

Despite its recent resurgence into the ranks of the most actively researched areas of AI, machine-learning is still far behind the capabilities of humans. The few techniques that have been used in automatic planners are therefore rather limited, but nevertheless account for large reductions in the search time.

- Parametrizing successful plans into new operators for later use was implemented with good results in STRIPS. This method works by gathering previous solutions into macro operators (MACROPS), i.e., groups of basic operators chained together, with parameters instead of constants to make them generally applicable.
- Script-based planning, also called refinement of skeletal plans, was used in MOLGEN2. However, this approach does not provide a mechanism for storing plans. Instead, the task of the planner is to choose one of a selection of pre-entered skeletal or abstract plans which contain the main steps leading to a solution, and then refining it to suit particular problem requirements. Because there is no way of generating a new abstract plan, MOLGEN2 is limited to its target domain, i.e. it is not a general problem solver.
- Modelling skill acquisition was the purpose of HACKER. Its technique can be described as using old plans (skills) to solve subgoals of a new problem, even if they are not completely suitable. In such cases 'bugs' occur in the new solution and these have to be eliminated by a special 'debugging' mechanism. Once workable, this new solution becomes a new skill of the system.

considerable promise, also when it comes to interfacing with the real-world.

3.2 Definition of Planning

If the goals of AI, i.e., vision, natural language understanding, learning and various problem solving, form the basis of intelligence, then not only humans are intelligent, but so are most animals (certainly capable of vision, simple problem solving and learning all necessary in the fight for survival). Nevertheless, it is highly unlikely that animals share with us the notion of *planning*. Thus, it belongs among the more sophisticated signs of intelligence, together with other abstract reasoning methods such as deduction or inductive inference. There are even differences between people; those who plan their actions carefully are perhaps more likely to succeed in whatever they are doing.

In the context of problem solving, a plan is a method of solution thought out in advance. Functionally, a planner is then a problem solver (typically general) capable of building a solution plan *before* acting. Structurally, the simplest planner will consist of:

- World model (problem representation).
- Operators (action representation).
- Solution strategy (search method).

Often, there is another part:

- Deductive mechanism (automatic deduction).

A *world model* is a symbolic abstraction of the outside world. It typically uses state-space description and the states with particular importance are the initial and goal states. The representation is handled by appropriate symbols, attributes and relations, often combined via predicate calculus or a similar scheme.

Operators, also called *action models*, are abstractions of the system itself, i.e., of the allowed transformations between states that the system can make, and their interactions with the outside world. Thus, each action description typically consists

The supervisor described thus far has been mainly concerned with the possibility of the plant and its environment going beyond the type and range of changes which the control system can tolerate. But even if the operating conditions fall within the acceptable limits, a supervisor's special abilities may be utilized to enable the system to follow far more complex and sophisticated objectives than those typically associated with basic control systems. This could take on the form of deciding on the most suitable control operation for a given task, or coordinating several individual controllers to manage a complex plant.

Supervision is then necessary if the control system:

- o Cannot meet all the objectives regarding the desired response of the plant.
- o Cannot ensure satisfactory response under all possible operating conditions.

The first case can be illustrated by an aircraft equipped with an automatic pilot (which qualifies as a low-level control system). Even though the autopilot may be controlling the plane for 90% of its flying time, a human pilot is necessary to monitor the conditions, manually change the settings of the autopilot to follow a given course and to take over the control task during each take-off and landing (not to mention emergencies).

The second case is the typical reason for having human operators supervise what are otherwise, fully automatic control systems. They are expected to notice and deal with problems before any damage is done.

2.1.3 Supervisory Tasks

Though the details of supervising different control systems can vary considerably, certain generic tasks can be identified as follows:

- o Interacting with the system.
- o Monitoring the system's performance.
- o Making decisions.
- o Taking appropriate actions.

3.5.4 Opportunistic Planning

Opportunistic planning is a very different approach in which appropriate actions are proposed when the right opportunity arises. The inference control strategy is far less structured than in the other approaches, mainly because it is distributed between many so called specialists, which work in an asynchronous fashion to provide a very flexible system. In order to recognize opportunities arising in the course of the planning process, the system has to consider low-level (detailed) actions from an early stage and that is where it differs markedly from hierarchical planning.

The idea behind this approach was to model certain characteristics present in human problem solving. The claim that human cognition displays opportunistic behaviour is not disputed here. However, it is viewed as emphasizing the less desirable features of human planning. Indeed, purely opportunistic planning is something of a 'jumping to conclusions' method, contrasting with the structured and calculated approaches of the other systems. Another drawback of this approach is that it requires a complex architecture—a blackboard control structure.

3.6 Advanced Representation Methods

3.6.1 Temporal Logic

The most significant development in the representation methods used by automatic planners has been the explicit treatment of time in temporal logic [Allen, 84]. In the typical representation scheme—the state-space description—world models describe the world at different instants in time. Actions effect changes in world models and therefore they can be seen as functions from one instantaneous world model to another. The limited expressive power of this approach (also called situation calculus) results in a sequential ordering of actions in the solution plan. While this is quite acceptable in many situations, there are cases where simultaneously occurring or overlapping actions must be represented.

The temporal logic maps actions and events into temporal intervals, instead of time points. This allows for more sophisticated ordering of actions: there are thirteen possible relationships in total:

X before Y

of a set of preconditions, which need to be satisfied in the current state for the action to apply, and a set of effects on the current state. The effects are typically described in terms of the state expressions which are eliminated and those that are added to the state description in the course of executing the action. The representation method used is the same as for the world model.

The *search strategy*, also called inference control, is the way in which the system goes about generating and testing new nodes in the search space, with the aim of finding a path joining the initial and goal states. Solution path is the necessary subspace of the entire search space (i.e., all theoretically possible combinations of operators), which must be expanded to find a solution. The nodes in this space correspond to the states the system plans to go through, while generating new nodes refers to applying different operators to the current node.

The size of the solution path is a measure of the efficiency of a particular strategy. A good strategy will only need to expand a small portion of the search space to arrive at a solution, while an inefficient method will result in the need to explore almost the entire search.

A *deductive mechanism* is only required if the representation used for state description allows complicated expressions which cannot be directly handled by the inference process. Its symbolic manipulation takes place within a single state only and for the case of predicate calculus representation, it assumes the form of a logic theorem prover.

The clear separation in the structure of automatic planners and insistence on generality of both the representation method and the search technique give rise to two major shortcomings:

1. Combinatorial explosion.
2. Limited expressive power

Combinatorial explosion is the consequence of using a 'weak' search strategy, which will necessarily conform to combinatorial principles. As the complexity of the task domain increases and more operators need to be admitted to the database, the search space of possible combinations of the operators grows exponentially. As a result, the performance decreases exponentially.

The fact that any non-trivial problem has a potentially huge search space, explains

Each of these tasks requires specific abilities. Interacting or communicating with the system implies the ability to work with real-time, continuous data. Performance monitoring applies to the plant as well as the control system itself, and it involves some complex issues. They include conversion of continuous, unprocessed data into high-level symbolic representation, testing for problems and diagnosing them.

The process of making decisions by using some form of reasoning is probably the most complicated part of supervision. The need for an agent capable of reasoning sets it apart from simple decision-making based on mathematical algorithms (which is the basis of low-level direct control). Qualitative rather than quantitative methods become necessary and the best ones implemented so far, try to model some aspect of human thought, for example, logic inferences, fuzzy logic, production rules, planning, qualitative causal reasoning and other AI techniques.

Finally, taking the decided-upon actions involves the reverse of data concentration, i.e., translation of symbolic representation into low-level continuous data, which can be processed by other parts of the system.

2.2 Automatic Process Control

The processing industry is an excellent example of an area characterized by a high degree of automation. This allows a clear separation of the basic control functions from the role of supervision. These differences are reflected in the fact that the low-level control is performed automatically, but most supervisory tasks rely on people (operators, engineers, management). It is also of particular importance to us, as a small plant which is a part of almost any processing production line is much more available for experimental evaluations than, for instance, an automated vehicle or aircraft etc.

The processing industry has experienced a historical shift from the very early manual control to the very latest digital control. A summary of the different approaches to automatic process control is shown below.

Automatic process control:

- o Analogue computer control
- o Digital computer control.

X equal Y
X meets Y
X overlaps Y
X during Y
X starts Y
X finishes Y

plus their inverses. Temporal logic is a very important enhancement of the state-space representation, it enables planners to deal with more complicated worlds and, therefore, solve a greater variety of problems. On the other hand, there are many problems adequately handled by the situation calculus and for which there is no need to introduce the extra computational overhead.

3.7 Conclusion

This chapter has provided the necessary information on planning. It has introduced planning as an important problem solving method. The fundamental concepts used in planning have been explained in detail, particularly the basic knowledge representation and search methods. A survey of the most significant new developments in these areas has also been included, with particular emphasis on the concept of reactive planning.

why even the earliest planners treated search methods with care and most research since then has been concentrated on dealing with combinatorial explosion.

The limitations in the *expressive power* of the representation used, follow from the assumption that it should be equally suitable for many different domains. Since even the most specialized representations used in AI are still far from ideal in terms of expressiveness, a universal representation is bound to be even less satisfactory. It will be 'good' for many cases but not particularly good in any one. This shortcoming is reflected in the typical difficulties encountered in mapping features from the real world to an abstract model.

3.3 Basic Search Strategies

Although this discussion is in the context of planners, many of the techniques described here are just as relevant for other problem solvers. Search techniques can be classified according to two major criteria: choice of the *starting point* and choice of the *next move*.

A *forward* directed search begins at the initial state and progresses toward the goal. Therefore, operators are examined by looking at their preconditions to see if these are present in the current state. Effects are used to create new nodes in the search space, until one of these new nodes is equal to the goal.

Forward chaining is suitable in those cases where the goal state is not as important as the initial state and, in fact, there does not have to be a goal at all at the beginning of search. An example would be monitoring and alarm triggering, where all that counts is the state of the process and whether an abnormal condition has been noticed. It is also known as bottom-up or data-driven approach.

A *backward* chaining method works from the goal state, looking for the operators whose effects coincide with it. Their preconditions are then compared with the initial state and if they are not satisfied, intermediate nodes are generated. Backward chaining is suited to situations where the goal state is known at the beginning of search. It is widely used in solving diagnostic type problems. In such cases, the fault is the starting node from which different paths are explored to find the cause. Alternative names are top-down or goal-driven.

Once the starting node has been decided on, the question remains whether to apply

- direct digital control
- self-tuning control
- adaptive control
- optimizing control
- advanced control
- AI-based control

Analogue computer control. This has been the norm for decades and is often called conventional control or just analogue control. Analogue computer is any information processing device which operates on continuous data. The classic control theory deals with continuous control systems and any control system is intrinsically a dynamic information processing device. The computational algorithms implemented by these systems are the well-known proportional (P), integral (I), derivative (D), phase lead/lag etc. The architecture of these systems is typically feedback and based on single-input, single-output, individual control loops.

Such devices continuously generate a control signal related by a simple mathematical function to the difference between the setpoint (the reference value) and the actual measured value of the controlled variable. However, more complex configurations, such as cascade feedback, are also possible. Early implementations were pneumatic and later gave way to analogue electronics (still widely used in industry). Analogue methods suffer from limited computational capability, inherent inflexibility and inability to deal with non-numeric data.

Digital computer control. This approach relies on the use of a digital computer which works in a discrete-time domain. This means that a continuous-time process needs to be sampled with a high enough frequency before any further signal processing can take place. Such systems can typically be divided into three large parts: input signal-processing, control processing (i.e., transforming the plant measurements and other inputs into a control signal) and output signal-processing.

The greatest advantage of digital computer control is the enormous flexibility of the computer and, consequently, the potential variety of applications. There is already a multitude of different approaches, from real-time direct control systems to off-line advisory expert systems, whose only common feature is the digital computer they are based on. This is probably only the beginning and there is a lot more to

Chapter 4

Design of the Planner

Every design involves certain compromises. In this case, the main trade-off is in deciding between a general-purpose system and a domain-specific one. This topic has already been discussed in relation to other problem solving systems. It is important to briefly look at the distinction again, this time from the point of view of the design objective.

This objective is to build an intelligent supervisory control system whose performance can be tested on an existing laboratory plant (level control plant). It would not be difficult to imagine a purpose built system to supervise this particular simple plant. There is little doubt that such a system would prove easier to develop, faster in performance and smaller in size.

However, the approach presented here relies on the 'brain' of the system (i.e., the planner) being sufficiently general, to make the system applicable to a range of problems encountered in control (and not just this particular plant). Yet, within the collection of generally applicable methods described in Chapter 3, some are more suited to the intended application than others. Those that fit best are chosen and modified or simplified, without taking away their generality.

the first suitable operator to create a new node and immediately shift the attention to the new node or, alternatively, first exhaust all options at each node.

The first method is called *depth-first search*, as it explores each branch of the search tree as far as possible (Figure 3.1). If it finds a solution, the search is complete, otherwise it will backtrack to the most recent node and try another operator. Depth-first is more suited to search spaces which are wide but not very deep, where alternative branches are of similar length and, particularly, where several solutions are possible, which is the case for many problems. If no solution is found, it simply degenerates to an exhaustive search.

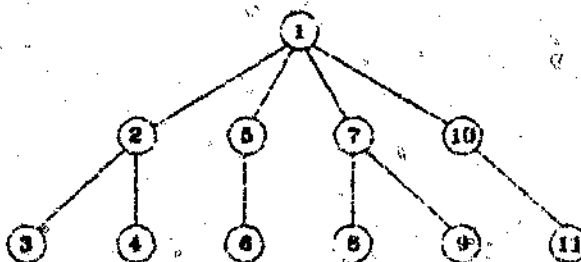


Figure 3.1: Depth-first search in a suitable space.

In a *breadth-first search*, all possible operators are tried before progressing to the next level (Figure 3.2). Therefore, breadth-first is more appropriate where some branches are short, while others are very long, and the entire space is not wide (i.e., few operators are possible at each node). It is especially advantageous where there is a possibility that a unique solution lies close to the starting node, surrounded by very long branches leading to nowhere. Again, it will degenerate to an exhaustive search, if no solution is encountered.

There are many other search strategies optimized for different applications, including least-cost and hill-climbing (which have an additional heuristic cost function), but one particularly worthy of mention here is the *means-ends analysis*. It was pioneered by GPS and subsequently used in many other benchmark planners (e.g., STRIPS).

Means-ends analysis is well suited to planning applications because it takes into account not only the starting state (whether initial or goal), but rather the *difference* between the initial and goal states. Different operators are ranked according to their usefulness in reducing a particular difference. If there is no operator which can eliminate the difference altogether (i.e., directly bridge the initial and goal states) the next best choice is made. If this operator is directly applicable in the initial

come with new advances in computer hardware and software (such as superconducting materials, parallel processing, symbolic languages, neural-network architectures etc.).

Direct digital control. This is the natural successor to analogue control. It uses the same control functions and algorithms as before, but the actual signal-processing is done using a digital rather than analogue computer. The theory of sampling ensures that a digital computer can fully replace an analogue one for continuous processes.

Rapid developments in microprocessor technology have resulted in progressively cheaper and more powerful and reliable microcomputers replacing analogue controllers. Advances in communication networks have led to distributed digital control systems providing new levels of reliability, in addition to the inherent flexibility and ease of configuration of software-based control algorithms. Other very important advantages of direct digital control (DDC) are the superiority of cathode ray tube (CRT) based displays over their predecessors in flexibility, accuracy and information capacity, the elimination of controller drifts (common in analogue systems), easier interfacing with other low-level or higher-level control systems and reserves in computational capacity.

On the other hand, pure DDC is something of a very hi-tech implementation of the old, conventional control methods. The digital computer is only required to mimic (only more accurately and reliably) its analogue counterpart, which uses only a small fraction of its potential capabilities.

Self-tuning and Adaptive control. These are related concepts, requiring increased processing abilities to provide on-line controller parameter estimation. Self-tuning control typically refers to the case where the process parameters are not known in advance, but it can be assumed that they do not change with time. Evaluation of the *process parameters* enables the system to estimate the *controller parameters* to achieve required performance.

Adaptive control assumes that there are changes in the environment, necessitating a continuous ability to adjust the controller tuning parameters in relation to the changes in the process parameters. Self-tuning and adaptive control algorithms have been used successfully in industry for tuning PID controllers. Their major shortcoming is that the structural model of the plant as well as the controller must

4.1 Proposed Representation

4.1.1 World Representation

The standard state-space method is used, where a world model (state) is a set of statements, describing the features needed to solve the problem, at a specific point in time. The state-space representation blends in well with the control component as it is convenient to represent a control process in terms of steady states and transitions between them. This should not be confused with the state space (state variable) description used in modern control, which is a systematic, numerical method for representing complex, continuous-time systems.

In contrast, the representation used by the planner is non-numerical (i.e., symbolic) and non-continuous (but not discrete either—discrete typically refers to digitized continuous-time functions conforming to the Nyquist sampling theorem). Sampling of the state is event-related instead of frequency-related and reflected in adding and deleting certain propositions, rather than constantly updating continuous state variables (this will be discussed in more depth in Chapter 5).

The propositions we use here, are a relaxed version of individual predicate calculus propositions. They have the following distinguishing characteristics:

- o Relaxed syntax.
- o No connectives: quantifiers are allowed.

Consider, as an example, a world model from the STRIPS planner:

```
ATROBOT(A)
AT(BOX1,B)
PUSHABLE(BOX1)
INROOM(ROBOT,ROOM1)
```

In our representation the same world model would be as follows:

```
Robot.at.A
Box1.at.B
```

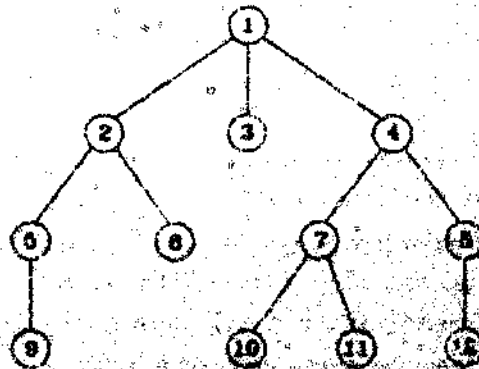


Figure 3.2: Breadth-first search in a suitable space.

state, then a new, intermediate state is formed and, consequently, the old difference reduced, by moving closer to the goal. The next step is to eliminate or reduce the new difference by repeating this process.

However, if the operator cannot be legally applied, the current process is temporarily suspended and a subgoal set up. Solving it is subject to the same method as the operator: preconditions, after which the main process is resumed as the current subgoal is solved.

Therefore, means ends is neither forward nor backward chaining, but could perhaps be best summarized as 'difference-driven'.

3.4 World and Action Representation Methods

The most widely used method of world representation in planning systems is the state-space description. The initial state contains all the propositions which hold just before planning commences. The goal state consists of all and only those propositions which have to be made to hold, before the problem can be said to be solved. Adding new propositions and deleting old ones is the consequence of trying out operators, i.e., abstract representations of actions that the system is capable of performing.

The state representation is a declarative way of describing the condition of the world, i.e., in which knowledge is explicit. Actions can also be represented declaratively or, sometimes, procedurally. Procedural representation is the natural one for most computer languages. Most computer programs (not just in AI) are collections of

be known in advance and must not change. In other words it is possible to 'adapt' to parameter changes (within certain limits), but not to structural changes.

Optimizing control. Optimizing control is typically concerned with large-scale economic implications of overall plant control. This is in contrast to a single-loop adaptive controller which, in a sense, also 'optimizes' its control action to achieve required performance. It is also just one of the more general optimization problems, the one concerned with plant operations. (The other two levels, as given in [Edgar and Himmelblau, 88], are the optimization in management decisions and optimization in process design and equipment specification, which are performed at the planning and design stages, respectively.)

In optimizing control, the entire plant (or at least a significant part of it) is taken into account and a complex numerical optimization technique implemented on a computer to calculate the best set of integrated control changes in order to maximize the profit at any time. The computational demands of optimizing control can, therefore, be very high, but not all processes require on-line optimization.

The main problem with optimizing control is that it requires a detailed plant or process model, as well as its economic model. In fact, the more accurate the mathematical representation, the better optimization can be achieved. The reality of large unpredictable variations in the operating conditions and consequent uncertainty in the mathematical model is the greatest drawback of optimizing control, often leading to sub optimal performance. In addition, the correctness of the plant and the control system is implicitly assumed and therefore structural changes cannot be handled.

Advanced control. This can take on many different forms, but it is typically characterized by its multi-input, multi-output capability, enhanced mathematical processing as well as inclusion of programmable logic functions. This enables an advanced controller to utilize an increased number of inputs from the process as well as from the operator console or other control systems (including digital inputs). It also makes possible switching between different control strategies (e.g., feedforward or feedback).

Advanced control allows an implementation of a much more complex overall control strategy than a low-level, single-loop controller. In principle, it can take over some of the higher-level control functions typically performed by human operators. It can be used for coordinated control, sequential batch control, processes which cannot be

Box1.is.pushable
Robot.in.room

Clearly, the changed syntax makes the propositions more difficult for machines to process, but easier for humans to understand. This is an intended effect, as the system designed here is aimed at people who may have no knowledge of predicate calculus. They should be able to enter initial and goal states, define new actions and occasionally monitor the supervisory process by looking at the state description; therefore, using simple English-like statements is preferable. The penalty of this more user-friendly approach lies in the need for more complicated string handling routines used for pattern-matching.

The decision to forgo the logic connectives (AND, OR, NOT, IMPLIES etc.) has two important consequences. The resulting representation is less expressive and it is no longer mathematically precise the way logic is (i.e., sound and complete). The first limitation is not a serious one, particularly in domains characterized by a number of propositions not related to one another. The second one means that a logic theorem prover cannot be used.

On the other hand, because only individual propositions are allowed, no theorem prover is necessary. All information is given explicitly and therefore straight pattern-matching can be used to compare states, operator preconditions or effects. This greatly simplifies the design of the planner and was, in fact, the main reason for using such representation.

4.1.2 Action Representation

Actions which the system is capable of executing are written as procedures. However, abstract action models (operators) which are used in the planning process are defined declaratively. As this section is only concerned with planning, the actual action procedures are discussed in later chapters. Each operator has a set of preconditions and effects associated with it. Our proposed structure for operators is shown in Figure 4.1.

POS CONDITIONS:	ACTION NAME	ADD EFFECTS:
NEG CONDITIONS:		DEL EFFECTS:

Figure 4.1: Logical structure of an operator

The preconditions are given in terms of positive conditions, i.e., all the propositions

procedures (sometimes called functions or routines) which are executed under certain conditions or often by default. A lot of information is encoded implicitly and therefore is difficult to see by anyone other than the programmer.

On the other hand, procedural representation is very efficient from the computational point of view. Encoding operators in a procedural way has a positive effect on efficiency for another two reasons. Firstly, it is generally much easier for such procedures to automatically update the world model with all the changes that follow from applying a particular operator. The difficulty of some (especially large) systems to propagate all side effects of proposed operators leads to the so called 'frame problem'. Secondly, the efficiency of the system is greatly influenced by including some domain-dependent heuristic knowledge. It is very easy to do that in procedural representation.

The disadvantages, in the context of general-purpose planners, centre around the issue of generality. Including any problem specific heuristics to influence the direction or order of evaluating possible solutions (except those explicitly allowed for in the search strategy) makes the system domain-dependent. Using implicitly encoded information (difficult for humans to understand) for the parts of the system which ought to be easily extensible (operators) is contradictory. An example of a planner with procedural operators is HACKER.

Probably the best known declarative way of representing knowledge is logic and its extensions. Most general-purpose planners use logic-based representation. A world model in STRIPS is defined as a collection of well-formed formulas (wffs) of first-order predicate calculus, where each wff describes something about the world objects and their relations.

Predicate calculus is an extension of propositional calculus (in which each logic statement is a simple proposition which cannot be broken down into parts and which has a value of either TRUE or FALSE) for handling more complex domains. Predicates are logic statements which allow a number of arguments to be evaluated. Specific objects, also called individuals, are substituted for the arguments and the result is either TRUE or FALSE, meaning that a given relationship between the arguments holds or does not.

First-order refers to the fact that predicates cannot be evaluated by other predicates the way that individuals are. Combinations of predicate calculus statements, using the logic connectives and quantifiers, are called well-formed formulas.

handled by conventional control, control algorithm selection as well as responding to structural failures. Many of these capabilities are due to the addition of logic, but sequential logic has definite limitations when applied to large-scale real-life control problems.

AI-based control. This is still in its infancy, but nevertheless it has already proved itself in some isolated industrial applications (mainly in the form of the expert systems). Although it is also based on the digital computer, its approach to information processing is radically different from all the other methods.

Every system which takes an action in response to certain conditions needs to make decisions. In the case of control systems, the decision determines what control action should be taken for a given set of operating conditions. All the approaches described above use a mathematical algorithm in their decision-making. In other words, the output is related to the input through a pre-designed and pre-entered mathematical function constituting a fully deterministic, fixed algorithm.

Advanced control adds to it a fully deterministic and fixed logic algorithm, which greatly enhances its capability. However, it is still numerically based decision-making, even if both analogue and digital numbers are used in the information processing.

In contrast, AI-based control puts emphasis on using *symbols* and not merely numbers in making decisions. As numbers form only a small subset of symbolic representation, they cannot capture all the available information (and particularly knowledge). On the other hand, numbers are well defined, precise, unambiguous and can be processed using standard and strict mathematical relations. Thus, they require little 'intelligence' in handling.

The more general symbols can be arbitrary and therefore much more 'intelligence' is needed to interpret them. Mathematical functions no longer apply, instead the AI techniques of pattern-matching and search have to be used. The relationship between the outputs and the inputs of an AI-based system can be nondeterministic and dynamic, as it is based on search rather than a fixed algorithm.

The symbolic nature of this decision-making bears much more resemblance to human problem solving. The unique advantages of AI based approach follow from its ability to capture some of the human-like intuitive or heuristic ways of representing knowledge. It can utilize qualitative knowledge, such as rules-of-thumb, which

which have to hold in a current state before the operator can be applied, and negative conditions which must not. The effects are divided in a similar fashion into add effects, i.e., all the propositions which have to be added to the current state after the operator has been executed, and delete effects which must be deleted.

It should be noted that positive conditions are logically ANDed, so that they all have to be satisfied, while negative conditions are logically ORed, so that not one must be present. Similarly, add effects are ANDed, so that all are added, while delete effects are ORed, so that only those present are deleted. For instance the equivalent of the STRIPS operator goto:

goto(M,N)
 Precondition: ATR(M)
 Delete list: ATR(M)
 Add list: ATR(N)

would be as follows:

POS CONDITIONS:	Go from M to N	ADD EFFECTS:
Robot_at_M		Robot_at_N
NEG CONDITIONS:		DEL EFFECTS:
		Robot_at_M

As can be concluded, the representation presented here is really just a simplified and more readable version, which has been found well suited to the intended application, while still preserving its general-purpose character. The type of application considered here does not require temporal logic and therefore does not justify the considerable extra effort it would involve in development and run-time processing.

4.2 Proposed Search Strategy

The system developed here can be summarized as general purpose, reactive, non-hierarchical, with no learning abilities and non-opportunistic. In other words, the inference control used is not domain-dependent and allows replanning to take place if necessary. Reactivity will be discussed Chapter 5 in the context of the entire system. This section only deals with the search strategy.

A simple example problem solved by STRIPS was as follows.

Initial state:

AtRobot(a)
At(Box1,a)
At(Box2,c)
At(Box3,d)

or in words:

The robot and the three boxes are all at different locations.

Goal state:

$(\exists x) \text{At}(\text{Box1},x) \wedge \text{At}(\text{Box2},x) \wedge \text{At}(\text{Box3},x)$

or in words:

The three boxes should be gathered together, at some location X.

The action representation is also based on what in the precondition part and individual predicates in the add and delete lists of effects.

For instance, operators applicable to the solution of the above problem were:

push(K,M,N)

Preconditions: $\text{At}(K,M) \wedge \text{AtRobot}(M)$

Delete list: $\text{AtRobot}(M), \text{At}(K,M)$

Add list: $\text{AtRobot}(N), \text{At}(K,N)$

or in words:

Provided both, object K and robot are at location M, the robot will push object K to location N, so that after the action is completed both object K and the robot will be at N.

goto(M,N)

Preconditions: $\text{AtRobot}(M)$

is commonly used by operators but totally wasted for numerically oriented control systems.

Symbolic reasoning creates enormous computational demands on the current computer architectures, resulting in limitations in the speed, size and capacity of such systems. However, it would be incorrect to view this as a permanent shortcoming. The rapid rate of development in computer technology has always exceeded the initial expectations and there is at present a concentrated effort going into designing efficient symbolic processing machines.

2.3 Automatic Supervision

Some of the techniques described above go so far beyond the conventional meaning of control, that it is probably more appropriate to classify them as supervisory techniques. In fact, an often found description of supervisory control is that it represents *everything above the low-level control loops*. Consequently, it enables achieving *better performance* or meeting *more sophisticated control objectives* than the basic controllers. It also follows, that a control system *can function without a supervisory control* (assuming normal operating conditions), albeit at a lower performance level.

It is for this last reason, that the earliest application of digital computer technology in the processing industry was in the field of supervision rather than direct control. These early implementations relied on a single, large and very expensive computer, which led to compromised reliability. However, these centralized supervisors (and their more recent distributed successors) did only a small part of a supervisor's job. They used complex algorithms to manipulate the setpoints to the lower-level analogue controllers and provided automatic start-up or shut-down sequences. They also enabled automatic logging of data, trend analysis and similar functions which only assist in supervising.

2.3.1 Supervisory Objectives

Consider the main goals for operating a plant, as defined in [Tsai *et al.*, 86]:

- o Maximize safety.
- o Adhere to governmental regulations, such as environmental standards.

Opportunistic planning has been ruled out as a method not appropriate to the eventual application. While opportunistic approach may be successful for planning a shopping trip, a control supervisor should only be directed toward fulfilling the goal as its main responsibility, rather than diverting its attention and looking for opportunities to perform some action of secondary importance.

Both hierarchical planning and including some learning capabilities are considered very useful extensions of the basic planner. And that is exactly how they are treated—as extensions—which can be added to the system when the need for higher speed and better efficiency arises, or when new techniques become available. For now, they are not included in the interest of focusing on more important issues.

The search technique we propose is forward chaining, depth-first but with an important feature of the means-ends analysis. A brief look at the area of application helps to explain and justify this choice. Automatic supervision of a control system is characterized by:

1. A limited number of applicable operators—due to the fact that there is a limited number of setpoints, overriding switches and emergency procedures, the supervisor (whether human or automatic) is restricted in the choice of actions he (or it) can take.
2. Data-driven approach—monitoring, emergency handling and alarm signalling are all extremely important actions in any supervisory system and they are triggered off in response to an abnormal condition in the current situation, but largely independent of the goal state.
3. Well specified operator preconditions and effects—as most actions are quite specific in when they can be applied and what changes they produce, both preconditions and effects can be used in guiding the search.
4. Branches of the search tree are quite uniform and not very long—supervision consists of taking the right few steps; there are no unduly long sequential operations.

It follows that the most suitable search method is forward chaining, which allows data-driven responses, and depth-first, which is well suited to the shape of the search space. In the interest of reducing the search space, it is modified to take into account not only operator preconditions, but also effects, in deciding on the next move.

Delete list: ATROBOT(M)

Add list: ATROBOT(N)

or in words:

Provided the robot is at location M, it will move to location N.

An example of a more complex STRIPS action is:

gothrudoor(K,L,M)

Preconditions: NEXTTO(ROBOT,K) $\wedge$ CONNECTS(K,L,M)

$\wedge$ INROOM(ROBOT,L) $\wedge$ ONFLOOR

Delete list: ATROBOT(\$), NEXTTO(ROBOT,\$), INROOM(ROBOT,\$)

Add list: INROOM(ROBOT,M)

or in words:

Provided the robot is in room L, close to the door K and K is the door between rooms L and M and all objects are on the floor, the robot will go through the door into room M.

Three conclusions follow from the approach used in STRIPS:

1. Because complicated wffs are used in goal descriptions and in action preconditions, STRIPS needs a logic theorem prover to extract all the information from them, as simple pattern matching is no longer sufficient.
2. For more complicated problems (in robot navigation), most wffs of the initial state describe static features of the problem (doors connecting rooms, stationary object locations etc.) and therefore they do not change throughout the planning process. To reduce data manipulation, STRIPS keeps all the initial wffs in a common database and newly generated states are characterized by their own lists of deletions and additions.
3. The strict form of predicates required by the theorem prover, makes it often difficult to understand the function of various operators and the meaning of state descriptions (particularly for someone not familiar with predicate calculus).

- o Maximize or control on-specification product production rate.
- o Minimize operating costs.

It is interesting to note that these goals apply just as well for planning and designing the plant as for operating it. Proper automatic supervision would be involved in all four aspects, but that is typically left to the human operators, the engineers and the management. The first two, in particular, do not easily conform to mathematical representation and may require complex problem solving, absent in the supervisory packages based on conventional software.

However, existing techniques can be used to improve the consistency in product quality as well as minimizing the operating costs. Supervisory control algorithms can reduce material and energy costs (by using various optimization methods to find the optimum operating conditions), maintenance costs (by providing early warnings) and labour costs (as fewer operators are needed due to decreased workload), [Tsai et al., 86].

It is quite clear that basic automatic control is necessary but not sufficient to fulfil the above goals. Conventional algorithm-based supervisory software can go a little further towards satisfying these goals, but perhaps not far enough to warrant the term 'supervisory'. Intelligent supervisory control seems to have significantly more potential of bridging that gap, as we will try to show in our case study.

2.3.2 Supervisory Actions

The generic tasks involved in supervising have already been discussed. The last of these is the ability to take appropriate actions. The specific actions at the supervisor's disposal will, again, depend on the actual problem at hand, but certain generic actions for the process control domain can be identified in view of the goals listed above.

For normal operating conditions:

- o Maximizing production and minimizing operating costs.
 - switch on the most effective controllers
 - provide the most effective setpoints for the controllers

This idea is borrowed from the means-ends analysis, with the important difference that if no operator with suitable preconditions and effects is found, the search process will degenerate to pure forward chaining and only preconditions will be looked at. In other words, the search is difference-driven whenever possible, but if necessary reverts to data-driven in an emergency, when achieving the goal becomes less important than preventing a disaster.

In case of a failure to reach a solution, backtracking takes place. Although there are methods more constructive than backtracking (e.g., nonlinear ordering) the approach taken here was to minimize the chances of backtracking occurring, rather than devising a separate mechanism for eliminating it altogether.

4.3 Organization and Processing of Knowledge

4.3.1 Data Structures

Problem specific knowledge, which includes initial and goal state descriptions and operator definitions, is organized in stand-alone units and therefore can be easily exchanged for another one when a different problem needs to be solved. The operators are stored declaratively in a single data structure which allows a very easy access to single operators or even single preconditions and effects. This is shown in Figure 4.2.

POS CONDITIONS:	OPERATOR 1	ADD EFFECTS:
NEG CONDITIONS:		DEL EFFECTS:
POS CONDITIONS:	OPERATOR 2	ADD EFFECTS:
NEG CONDITIONS:		DEL EFFECTS:
POS CONDITIONS:	OPERATOR n	ADD EFFECTS:
NEG CONDITIONS:		DEL EFFECTS:

Figure 4.2: Data structure for storing operators.

Domain-independent knowledge, which is the search strategy, is encoded in a recursive procedure which controls the inference process. This procedure explores paths in the search space by creating new intermediate states (through application of operators), or backtracking to previously traversed states if no operator can be

3.5 Advanced Planning Strategies

Sacerdoti provides a summary of various techniques used for automatic planning, in [Sacerdoti, 79]. Besides the basic search methods, he lists the following nine enhancements for improving planning efficiency:

- Hierarchical planning.
- Hierarchical plan repair.
- Bugging (or debugging).
- Special purpose subplanners.
- Constraint satisfaction.
- Relevant backtracking.
- Disproving.
- Pseudo reduction.
- Goal regression.

The *Handbook of Artificial Intelligence* [Cohen and Feigenbaum, 82] restricts the main approaches to the following:

- Nonhierarchical planning.
- Hierarchical planning.
- Script-based planning.
- Opportunistic planning.

Both of these taxonomies are a little outdated, because they do not cover the most recently researched area of *reactive planning*. A slightly different classification is proposed and explained below. It should be noted that the categories are not treated exclusively, i.e., a particular system can be a member of one or more of them (with the exception that an opportunistic planner probably has to be nonhierarchical).

1. Reactive versus non-reactive planning.

- provide the most effective controller parameter settings

In case of a safety risk:

o Preventing unsafe operation.

- switch on backup systems
- take over control
- shut down the plant
- issue warnings and alarms

This shows that safety is awarded the top priority status. Next priority is maximizing profits which is mainly done by maximizing the production rate and, secondly, by minimizing the costs involved in that production.

Thus, the main responsibility of the supervisor is to detect any changes which could lead to a safety risk. If that is the case, the supervisor must have some actions to prevent such situations. It should attempt to handle the emergency by itself, if it is within its capabilities. For instance, it could switch to backup systems, if such possibility exists, or take over the actual control of the plant and, if necessary, just lead the plant to an orderly and graceful shutdown. Quite often, the above actions would evoke a corresponding drop in performance (due to a switch to a less sophisticated control method) or even a stoppage in production (as in the case of a shutdown), if continuing may jeopardize the safety.

Another important type of action would be issuing warnings and alarms to get the attention of a higher authority (i.e., the person responsible for overseeing the system).

It should be stressed here, that a supervisor able to deal with all theoretically possible problems is an unrealistic ideal. Human supervisors also encounter problems which go beyond their expertise and training, and these have to be referred to the engineer or even the management. However, the supervisor's knowledge and ability to reason ensures that such problems are immediately identified and delegated higher up the hierarchy. This is in direct contrast to basic controllers which, no matter what happens, always assume their total competence.

When the system is operating well within its safety margin, economic considerations come to the fore. The supervisor should, therefore, have some actions which would

applied.

The working memory is provided by a stack data structure. Stack refers to a temporary storage of data which works on the last-in, first-out (LIFO) principle (i.e., items are accessible only on a one-by-one basis, starting with the last one entered). To be useful, stack must be accompanied by two procedures for storing and extracting its items, called *push* and *pop*.

In this case, the stack stores the path of the search space which is currently being explored. Therefore, the items on the stack correspond to the nodes in the search space. Each node is a state, i.e., a set of propositions. But it is also the explicit result of applying an operator, by adding the operator's add effects and deleting its del effects from the preceding node. Consequently, each item on the stack is a data structure itself. It contains all the propositions constituting an intermediate state, as well as the name of the operator whose application has resulted in this state.

In addition, because the planner must be able to backtrack and try new paths, another important piece of knowledge must be added to each node. To prevent the planner from trying again those operators which have been applied before and led to terminated paths, each node has to store them during backtracking. The stack data structure, an area of the search space and the corresponding stack item are shown in Figure 4.3. Figure 4.4 and Figure 4.5.

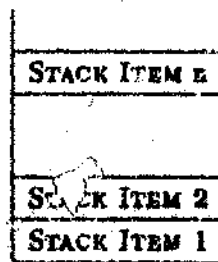


Figure 4.3: The stack data structure.

All currently existing intermediate states are stored on the stack. The initial state is stored at the very bottom of the stack. The goal state is stored separately in its own data structure. The stack always contains the currently searched path and when the planning process is completed (i.e., the goal is reached) the entire solution is on the stack.

2. Hierarchical versus non-hierarchical planning.
3. Planning with or without learning abilities.
4. Opportunistic versus non-opportunistic planning.

All these approaches have a different effect on the inference control strategy. While another category of temporal versus non-temporal planning can be identified, it has to do with the representation method, rather than the search strategy.

3.5.1 Reactive Planning

Reactive planning refers to the ability of the system to replan, if necessary. It seems an obvious requirement, especially as human planning relies heavily on this feature. However, early planners produced fixed solutions—plans that could not be retracted or modified, as there was no mechanism for monitoring plan execution.

Even though implemented only recently, interleaving of plan generation, execution and repair was recognized early on in [Sacerdoti, 74] and suggested as the focus of future research in [Sacerdoti, 79].

We agree with this opinion and consider reactivity as the most important characteristic of any automatic planning system. The immediate benefit of a system capable of execution monitoring and replanning is that plan generation is made much simpler, because it is no longer necessary to correctly predict all hypothetically possible future situations. But even more importantly, reactive planning is absolutely essential for dealing with real-world problems, where unpredictability and sudden changes are prominent.

Non-reactive planning is termed open-loop in [Smuts, 90], while a system capable of replanning is closed-loop. In the context of control, an intelligent supervisory system can be thought of as an abstraction of a basic control system.

An open-loop control system is illustrated in Figure 3.3. Such a system achieves a desired response R by passing a command C through a suitably designed prefilter P . An external disturbance d and a change in plant parameters Δp can be cancelled by modifying the command accordingly. This approach implies that all details about the process disturbance d and parameter variation Δp must be known to the system in advance. This is also the very reason why open-loop systems are used so rarely.

lead to an improvement in the overall performance of the plant, over that achieved by basic controllers. For example, it could switch on the most effective controllers for given operating conditions, if it has a choice of different control algorithms. Once the lower-level controllers are running, it could manipulate their setpoints to further improve performance. It could also directly manipulate the controller tuning parameters in accordance with some heuristic rules, totally independent of numerical tuning of adaptive algorithms (if such were used).

These examples of actions suggest that the supervisor should have total command of all lower-level systems, including analogue, direct digital, adaptive, optimizing and advanced controllers, as well as direct AI-based control methods, whichever are used in the system.

2.4 Conclusion

This chapter has defined the place of supervision in the context of control systems, particularly in process control. A logical succession has been shown to exist from automatic process control to intelligent supervision. The most important tasks of a supervisor and examples of general actions have been described.

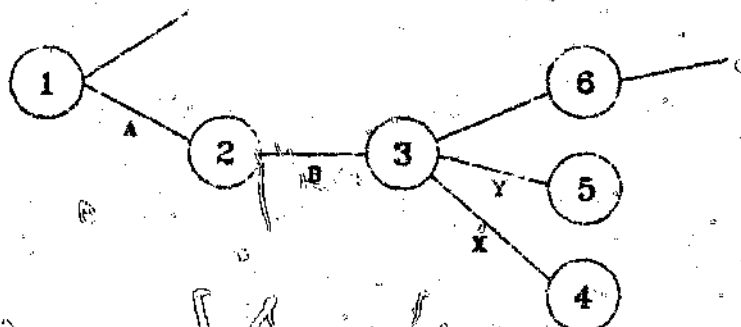


Figure 4.4: Currently explored search path.

STACK ITEM 3		
Preceding operator name: OPERATOR B	Intermediate state: STATE 3 TERMS	Inhibited operators: OPERATOR X OPERATOR Y

Figure 4.5: A single stack item.

4.3.2 Planning Process

Consider now the steps involved in the planning process. Before planning can begin, valid initial and goal states must be defined. If needed, many sequential goals can be entered to form a so called mission. If that is the case, the entire process described below is repeated for each remaining goal. Also, those operators which allow arguments, have to be initialized with constants.

When planning starts, the initial state is pushed onto the empty stack. The recursive search procedure checks if the state on top of the stack, i.e., the current state (or in this case the initial state) is equal to the goal state. If it is, a solution has been found (very trivial in this case, i.e., no action is required). If not, the available operators are taken one by one and put through a series of tests.

It is first checked if the operator being tested has already been tried from the same node, at an earlier stage. If it has, another operator is tried. Otherwise, the pos conditions are looked at first. A pattern matching procedure is used to check if all the pos conditions are present in the current state. If so, it is checked that no neg conditions are present. If these three tests are passed, the operator is applicable in the current state. Subsequently, the effects are checked, this time against the goal state.

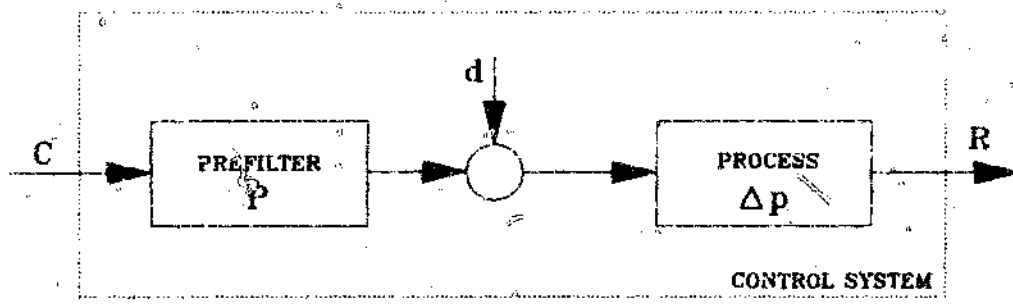


Figure 3.3: An open-loop control system.

The almost universal acceptance of feedback or closed loop systems (Figure 3.4) in real-world control applications can be traced down to the question of *uncertainty*. To assume total predictability of external disturbances or internal changes is simply unrealistic. But ignorance of the process or environment can be offset by constantly monitoring the response R through the feedback path with a sensor, and comparing it with a required setpoint S . The error (difference) E directs the actions of a compensator C .

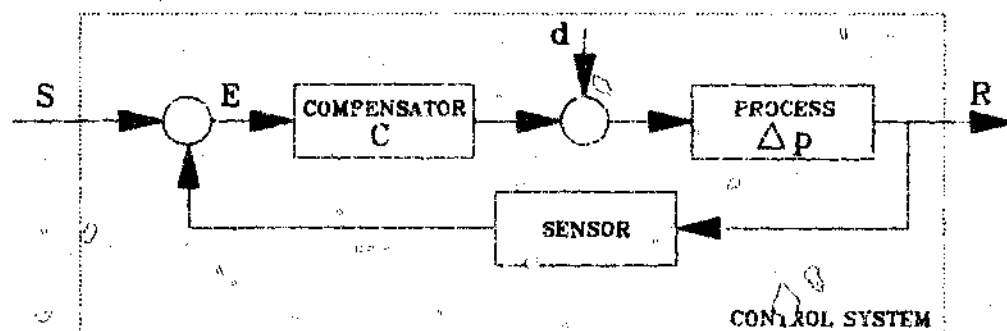


Figure 3.4: A closed-loop, feedback control system.

Similarly, a planner-based supervisory control system could be open-loop as in Figure 3.5, in which case all details about the disturbances D or process variations ΔP would have to be accurately predicted in advance (capital D and ΔP are used now to symbolize disturbances and parameter variations of great magnitude—beyond the capabilities of the controller—e.g., failure of the compensator). As can be seen from Figure 3.5, the supervisory system is at a higher level of abstraction; thicker arrows are now used to symbolize that the information exchange is no longer continuous or single-valued in nature.

Chapter 3

Introduction to Planning

This chapter contains the essential background on planning in the context of Artificial Intelligence. The existing planning techniques are explained and their advantages and disadvantages examined.

3.1 Approaches to Problem Solving

Although planning and problem solving are often regarded as synonymous in the AI literature, it is an imprecise and therefore potentially confusing view. Instead, planning should be thought of as only a subset of problem solving, as noted in [Cohen and Feigenbaum, 82], since it is just one of a variety of problem solving techniques belonging to AI. The distinguishing feature of planning is that of deciding on the course of action *before* acting.

3.1.1 Expert Problem Solvers

AI based problem solvers are usually divided into two large and distinct groups [Sussman, 75], [Sacerdoti, 74]. Those in the first group are called 'experts' as they are domain-specific systems, optimized for efficiency in their narrow field of expertise. Such systems have been developed in many areas of AI research, such as understanding natural language (e.g., SHRDLU), scientific applications (e.g., DEN-DRAL), medical diagnosis (e.g., MYCIN) and a multitude of game playing programs. More recently, expert systems have been applied to industrial problems, including

process control. They have proved to be successful in their performance. However, they all share the same disadvantages of being difficult to extend and practically impossible to use for tasks outside their limited scope.

The greatest strength of an expert problem solver, namely using shallow knowledge to dramatically prune the search space (thus increasing the efficiency), is also its major shortcoming: this pruning only occurs in the target problem space. The significant increase in efficiency, due to interleaving of problem related knowledge and the control of inference process, means that it is typically impossible to modify the system to adapt it to another domain.

3.1.2 General Problem Solvers

On the other hand, there is a group of generally applicable problem solvers which trade efficiency for extensibility. Systems such as GPS, QA3, STRIPS and many others, are characterised by making use of general heuristics, also called 'weak' methods, which by their nature are not particularly efficient in any specific domain. This is mainly because they are subject to combinatorial explosion in their search spaces as their databases grow, but also due to difficulties in encoding problem-specific knowledge using a general representation scheme. Each can, however, be applied (in theory at least) to a wide range of tasks from totally different domains (e.g., logic problems, blocks-world, robot navigation and game playing).

- The lack of problem-specific heuristics leads to a major shortcoming: enormous search spaces which are typical of general problem solvers. Well chosen search strategies can limit those, and there have been many other advances in this direction by GPS successors.

If we think of Artificial Intelligence as an attempt to create machines that act in an 'intelligent' way and if we take the human being as the best available example of intelligent behaviour (in the absence of a single, satisfying definition of 'intelligence'), then the two approaches can be looked at from a more familiar angle.

Domain-dependent methods resemble the human expert who has years of accumulated experience in his field and can quickly benefit from this precompiled body of knowledge in its ready-to-use form (shallow knowledge). Understandably, when faced with another type of problem from a different specialization, he will still be able to resort to general problem solving techniques, although his domain-specific

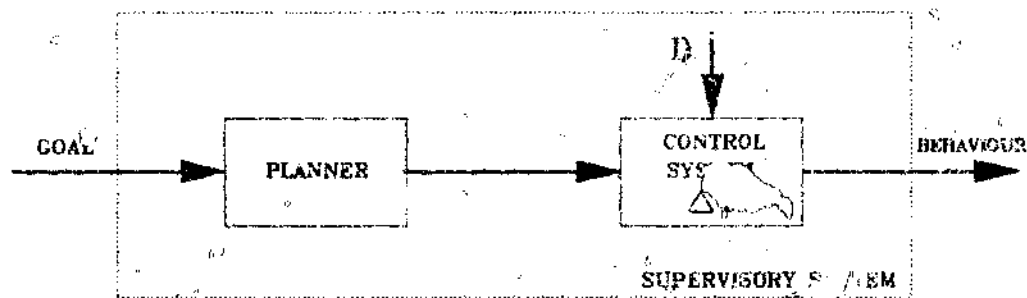


Figure 3.5: An open-loop planning system

Closing the loop in a supervisory system is illustrated in Figure 3.6. By monitoring the behaviour of the system and feeding it back, care is taken of unpredictable situations. The architecture of this system allows it to be directly interfaced with the real-world (as opposed to game playing or blocks-world, both highly predictable domains and typical targets for automatic planners). Whether the system has enough knowledge to be able to solve all potential problems and whether it is efficient enough to operate in real-time are another two issues which have to be considered at the design stage.

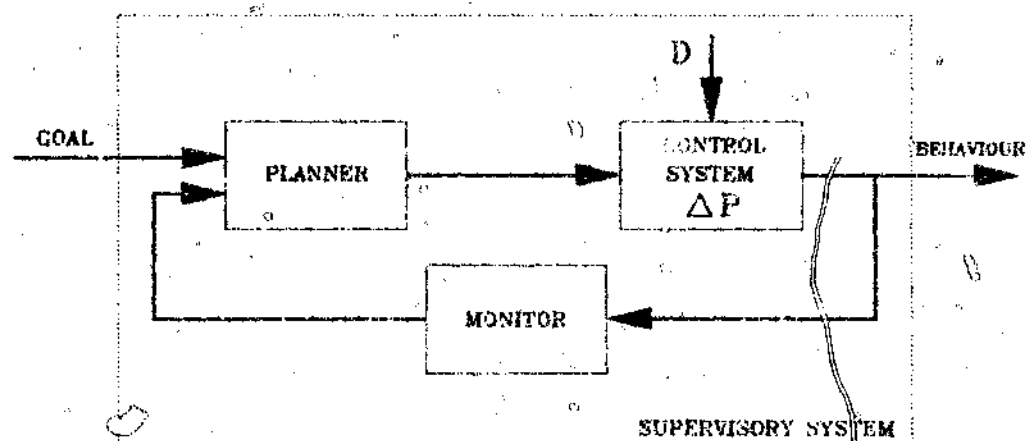


Figure 3.6: A closed-loop, reactive planning system.

It should be noted that the term reactive planner does not strictly refer to the planner itself. In fact, some existing non-reactive planners could be used in a reactive system, if suitably modified and equipped with the necessary subsystems. Instead, it refers to the entire system which has to include the feedback (monitoring) path and a number of extra modules (not explicitly shown in Figure 3.6 for clarity purposes). This topic will be covered in more depth in Chapter 5.

There are two distinct possibilities now, and each has a slightly different effect on the rest of the search. In the best case, the operator being checked has no effects corresponding to any of the propositions in the goal state and at least one of its add effects is also found in the goal state. If that is so, the operator is immediately applied to the current state. If none of its add effects features in the goal state, the operator is put on hold, while the remaining operators are tested. If none of these is any more applicable, the stored operator is applied to the current state.

Applying an operator to a state means that its add effects are added to the set of propositions in the state, while its del effects are deleted from it. Thus, a new intermediate state is created. Both the operator used to create the state and the state itself are then stored on top of the stack. The search procedure then calls itself.

If neither the first nor the second possibility occurs, i.e., no operator is found applicable in the current state, backtracking has to take place. First, the current state is taken off the top of the stack. Then, the name of the operator which had led to that state is written into the new current state and labelled as 'inhibited' (this tells the planner that this operator has already been applied to this state, but led to a dead-end). Finally, the search procedure is called.

The search procedure is outlined below:

Search

Begin

```

if (present state not equal to goal state)
  if (stack empty) push initial state on stack;
  while (operator set not empty)
    Take next operator;
    if (operator not inhibited)
      Check operator;
      if (preconditions satisfied and effects apply)
        Make new intermediate state;
        Push state and operator on stack;
        Call Search;
      end(if)
    if (preconditions satisfied)
      Put operator on hold;
    end(if)

```

knowledge will be useless. However, an automatic expert system based solely on task-dependent heuristics, will not function at all in such a case.

In contrast, the General Problem Solver (GPS) of Newell and Simon, evolved as a psychological theory aiming at explaining human problem solving. The clear separation between the domain-independent part of the system (which is the reasoning part) and the task-specific part (which is the problem description part) makes it applicable in many different areas.

In this aspect it is similar to the human novice, lacking in expertise in any field, but capable of intelligent dealing with a great variety of different problems, after being presented with the necessary information about the task environment. He will thus collect the necessary knowledge about the problem and use it with the help of one of the available general strategies. But again, he will adapt and after spending enough time in a single domain, he will start compiling his newly gained experience into ready-to-use chunks of knowledge (such as rules-of-thumb) to increase his efficiency.

Automatic general problem solvers are not yet capable of such behaviour, although there have been some interesting developments with a limited learning capacity (e.g., STRIPS and HACKER).

Since a hybrid system with combined advantages of both and no drawbacks of either method is not yet achievable, a choice has to be made and it seems most logical to base this choice on the eventual application. In our case, the intended application requires facing with the outside world, rather than staying within the bounds of theoretical considerations. Although the recent rise in popularity of specialized expert systems and their successful adaptation in many industries seem to indicate their proficiency in dealing with real-world situations, this is not necessarily so.

Indeed, the actual real-world performance of most specialized expert systems is extremely difficult to evaluate, as a typical expert-system shell is geared towards an indirect interface with the outside. The buffer in this case is provided by the human operator who may or may not act according to the recommendation given by the system.

In a true automatic supervisory system, there is no provision for the human operator, acting as the buffer. In fact, the main reason for constructing such a system, is to eliminate the human link. Besides, basic supervision depends more on general problem solving, such as pattern recognition, situation assessment and planning, than on very specific expertise. Recent developments in planning systems show

3.5.2 Hierarchical Planning

Hierarchical planning describes the process of plan refinement also called planning in a hierarchy of abstraction spaces. All it means is that not just one, but multiple versions of the solution plan are generated. At first, the planner works in the most abstract space (i.e., most devoid of details) and constructs a complete but very simple (vague) plan. Subsequently, this skeleton plan is refined by filling in all the missing details (detailed subplans) until the final stage is reached and the solution plan is given in terms of the available low-level operators.

A hierarchy of abstraction spaces thus refers to moving from the simplest to the most complex plan representation supported by the system. The actual method of abstracting or hiding irrelevant detail depends on the implementation and the following techniques have been used:

- o Problem representation abstraction, i.e., distinguishing between different problem representations according to their generality; first plan works with the most general problem representation (e.g., GPS).
- o Goal abstraction, i.e., distinguishing between subgoals according to their relative importance; first plan will try to satisfy the most important goals (e.g., ABSTRIPS).
- o Operator abstraction, i.e., distinguishing between operators according to their generality; early plans will be in terms of very abstract operators (e.g., NOAH; as goals are represented in terms of operators in NOAH, it could also be classified under goal abstraction).
- o Planning abstraction, i.e., distinguishing between different planning spaces according to their function. In the lowest space plan construction is controlled, in a higher space plan design is controlled and in the highest planning space design strategies are decided upon (e.g., MOLGEN).

The great advantage of hierarchical planning lies in the fact that it reduces the search space and therefore significantly improves planning efficiency. Its disadvantage is the increased complexity of the system. In the context of real-world problems, hierarchical planning is not necessary, but could be very helpful. It would be particularly suitable in those cases where a large number of operators applies, creating a huge search space and, at the same time, the speed of the system is critical.

```

    end(if)
  end(while)
  if (operator on hold)
    Make new intermediate state;
    Push state and operator on stack;
    Call Search;
  else
    Inhibit operator;
    Pop previous state off stack;
    Call Search;
  end(if)
end(if)
End.

```

4.4 Planning Example

To substantiate the claims of a general-purpose character of the proposed planner, a simple problem from a non-control domain is presented. The Nickel and Dime Problem (NDP) could be classified in the game playing category and is taken from [Marron and King, 85]. The purpose of this section is to help illustrate the basic forward planning strategy described earlier.

4.4.1 Problem Definition

Two nickels and two dimes (or any other distinguishable objects) are positioned in a row with an empty space between them, as shown in Figure 4.5. The goal is to exchange their positions, so that the two dimes replace the two nickels and vice versa. There are two legal kinds of moves, namely sliding one place forward or hopping over one place. There is also an added complication that moving backward (whether sliding or hopping) is not allowed.

This is a simple but not a trivial problem, as can be quickly asserted after trying a few haphazard moves.

INITIAL STATE

N	N	D	D
---	---	---	---

GOAL STATE

D	D	N	N
---	---	---	---

Figure 4.6: The Nickel and Dime Problem.

4.4.2 Problem Representation

To enable easy, declarative representation the squares are numbered as follows:

1	2	3	4	5
---	---	---	---	---

Consequently, four basic operators can be defined (four instead of two, because nickels can only slide or hop to the right, while dimes slide and hop to the left). They are as follows:

Pre Conditions: nickel_position	Slide nickel_position	Add Effects: nickel_position+1
Post Conditions: nickel_position+1 dime_position+1		Del Effects: nickel_position

Pre Conditions: nickel_position	Hop nickel_position	Add Effects: nickel_position+2
Post Conditions: nickel_position+2 dime_position+2		Del Effects: nickel_position

Pre Conditions: dime_position	Slide dime_position	Add Effects: dime_position-1
Post Conditions: dime_position-1 nickel_position-1		Del Effects: dime_position

Pos Conditions:	Hop.dine.position	Add Effects:
dine.position		dine.position-2
Next Conditions:		On 2.1.1
dine.position-2		dine.position
nickel.position-2		

Before the search can commence, these four operations are put in an initializing routine which substitutes the numbers from 1 to 5 for the argument 'position'. Overall, there are 14 moves, rather than 20, as some are impossible, e.g. to place a nickel at position 2 - the example of an 'illegal' state shown above.

Pos Conditions:	Hop.dine.position	Add Effects:
dine.1		dine.1
Next Conditions:		On 2.1.1
dine.2		dine.2
nickel.1		

The above example shows that a move which is impossible is not a move at all, and is not a move at all, and is not a move at all.

3.4.3. Solution

The initial state is (1) (2) (3) (4) (5)

nickel.1
nickel.2
nickel.3
nickel.4
nickel.5

and the goal state is:

GOAL STATE
dine.1
dine.2
nickel.4
nickel.5

The planner produces the following solution plan (actually it only produces descriptive outputs, more suitable in supervisory applications, but they are showed here in an equivalent graphical way for better clarity):

1: Slide nickel 2

N		N	D	D
---	--	---	---	---

2: Hop dime 4

N	D	N		D
---	---	---	--	---

3: Slide dime 5

N	D	N	D	
---	---	---	---	--

4: Hop nickel 3

N	D		D	N
---	---	--	---	---

5: Hop nickel 1

	D	N	D	N
--	---	---	---	---

6: Slide dime 2

D		N	D	N
---	--	---	---	---

7: Hop dime 4

D	D	N		N
---	---	---	--	---

8: Slide nickel 3

D	D		N	N
---	---	--	---	---

Because the domain specific knowledge contains little information to efficiently guide the search, the planner has to resort to backtracking to find the solution. This example is a good illustration of the flexibility of the search technique used and the consequences that follow from it. Consider now that the initial state is as follows:

N				
---	--	--	--	--

and the goal is:

				N
--	--	--	--	---

The planner arrives at the following solution:

1: Slide nickel 1

	N			
--	---	--	--	--

2: Slide nickel 2

		N	
--	--	---	--

3: Hop nickel 3

			N
--	--	--	---

This shows how the goal state cannot provide any useful information to choose the optimum move in the first two instances. Therefore, the first applicable operator is taken (in this case the nickel is slid, even though hopping would be better). Once the nickel is at position 3, enough information becomes available, so that the difference between the current and goal state can be used to choose 'hopp' over 'slide'.

4.5 Conclusion

Design of the AI-based part of the supervisory system has been the topic of this chapter. It is a reactive planning system drawing on the techniques described in Chapter 3, but characterized by a few important modifications in the areas of knowledge representation and search. These modifications take into account the eventual application to a system (real-time supervisory control), but do not change its general-purpose character. The approach taken in organizing the knowledge-base and in processing it, has been discussed in detail.

A simple example has been included to help explain the actual functioning of the planner and to show that it is a general-purpose system. The example is based on simulation and uses the planner in an open-loop configuration. This is typically the only kind of testing that most AI-based systems are subjected to. In the following chapters the planner will be incorporated into a closed-loop system and interfaced with a real-world plant.

Chapter 5

Design of the Supervisor

This chapter explains how the planning system can be used for control supervision. An architecture is proposed and each constituent part described. The issue of concurrent execution of the different parts on a single processor is also addressed.

5.1 Overall System Architecture

On the basis of the essential abilities required of a supervisor, we propose the basic structure by combining the different functional modules into a coherent, closed-loop system, as shown in Figure 5.1.

5.1.1 Read and Write Modules

The read and write modules form the system's interface with the plant which is to be supervised. They are necessary in any digital control application for translating the continuous-time data into discrete-time sampled data and vice versa. Therefore, they have to conform to the Nyquist sampling theorem, which states that the sampling frequency must be at least twice as high as the highest component frequency present in the continuous-time input.

- Physically, these modules consist of a hardware component, which is not a part of the digital computer, i.e., an analogue to digital (A/D) converter for the read module and a digital to analogue (D/A) converter for the write module (these two

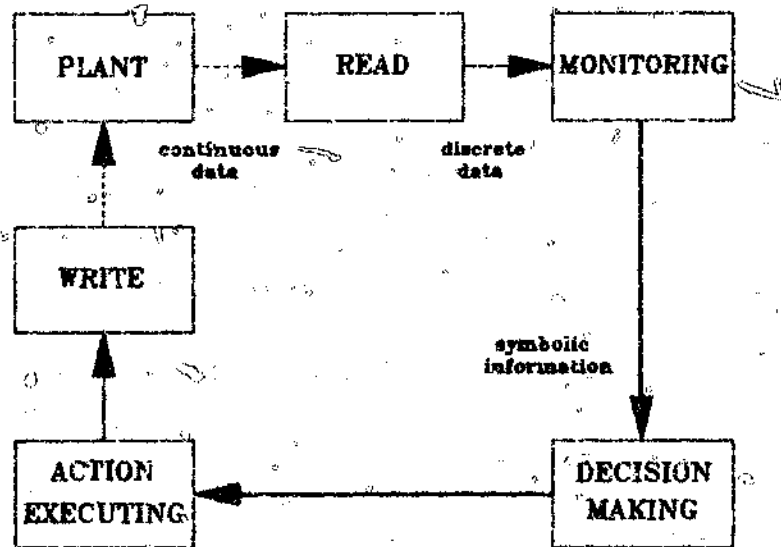


Figure 5.1: Basic structure of the supervisory system.

are often combined into a single ADAC). They also need a software program within the computer, to interface with the external hardware. Thus, if a boundary line were to be drawn to separate the computer from the external world, it would cut through the read and write modules. Figure 5.2 illustrates this difference between the physical and the functional structures.

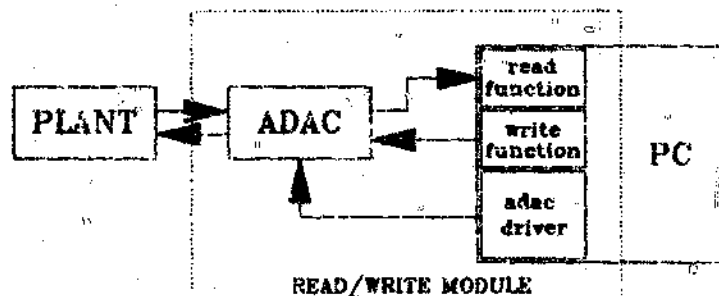


Figure 5.2: The read and write module.

5.1.2 Monitoring Module

The *monitoring* module receives discrete real-time data and it uses this data to create a symbolic representation of the state of the plant. This process can be regarded as data 'concentration', because large amounts of signals, each individually carrying

little information, are transformed into few abstract symbols, rich in information. Monitoring is responsible for detecting any kinds of critical or abnormal conditions, equipment status changes and failures and any other important changes. It is used for evaluating the effectiveness of the actions performed, diagnosing problems as well as creating a condensed, high-level representation of the state of the system, which can be used in the decision-making.

In order to perform such varied and complex tasks, the module has to make use of the kind of knowledge and inference that human operators and process engineers use. The functions involved in monitoring are analytical in nature. The signals coming in from the plant, the controllers and other subsystems are analysed for any unusual patterns which could be symptoms of an impending problem. They are also analysed for any other characteristic patterns which need to be translated into symbols to assess the situation.

A representation which is very well suited to analytical tasks and, therefore, by far the most common for diagnostic systems, is based on production rules. Production rules are in the form of IF-THEN clauses, where the IF part determines what conditions have to be satisfied before the THEN part can be applied. Such rules can embody the knowledge necessary for the monitoring functions in a very convenient, ready-to-use manner.

It should be noted that monitoring is domain-dependent. The rules applicable to one system may be totally unsuitable for another. The number, contents and ordering of rules within the monitoring module will all be specific for each different application. For illustration purposes, consider the following simple example:

IF *temperature* > 80
THEN *High.temperature*

This rule will create a symbolic output *High.temperature* whenever the numeric variable *temperature* exceeds 80 (e.g., degrees Celsius). *High.temperature* will then be added to any other symbols representing the current state of the plant and subsequently processed by the decision-making module. The second example shows a rule which contains some diagnostic knowledge:

IF *temperature* > 120 and *pressure* > 100
THEN *Cooling.system.failure*

This time there is enough information to diagnose the faulty subsystem.

These types of rules are often software copies of the heuristic rules used every day by human operators and based on their experience. There may be more sophisticated rules, based on deeper knowledge of the system, and encoded with the help of the process engineer.

There may also be more sophisticated representations, for instance, rules based on *fuzzy logic* which allow an even better approximation of the real situation. However, that in itself is a broad area of research which is not central to this work. For the purposes of this research, the monitoring module is looked at mainly from the functional point of view and our implementation is based on the simple but effective *IF-THEN* rules.

Perhaps the most important function of the monitoring module lies in its ability to invoke the decision-making process in reaction to a critical change in the system. The monitoring module must also create a symbolic representation of the current state of the system (including that latest change which is the reason for replanning) and make it available to the decision-making module.

It is important to realize that there can be, in principle, three different operating frequencies for the three different modules. The highest frequency is determined by the Nyquist sampling theorem and applies to the read module. In theory, there is no requirement on the monitoring module to operate at such high frequency or any other specific frequency, for that matter.

Even more significantly, in practice it may be impossible for the decision-making module to operate at anything near the sampling frequency (due to its computational intensity), nor is there any need for that. In the proposed architecture the monitoring function determines when the decision-making module is to run, i.e., it has full control over the operating frequency of the decision-making module. This issue will be revisited in the context of multitasking.

3.1.3 Decision-Making Module

The *decision-making* module, based on AI principles and techniques, forms the central part of the proposed system. Two separate chapters have been set aside to discuss the details of, firstly, different ways of approaching this issue and, secondly, the approach taken by us, and its consequences. The general-purpose reactive plan-

ning system developed in Chapter 4, forms the mechanism by which our supervisory system decides what actions to take and when.

Decision-making is a creative task since a decision or solution has to be *synthesized* from the available information. Planning is ideally suited to this task, as it is a naturally creative activity. Every plan constitutes a new creation built up from many smaller pieces (actions) combined in a unique way.

Every action which the supervi has at its disposal has a corresponding entry in the planner's knowledge base. These entries are called operators and they are abstract representations of the actual actions. Each operator has the same name as the action it represents. It also contains a set of preconditions and effects associated with that action. They are used in the planning process to determine if a given operator is applicable in the current state and what effects it will have on that state, if applied. The state of the system, as discussed here, refers not to the numerical state description that is used by low-level controllers, but to its abstract representation.

Another input to the planner is the triggering signal received from the monitoring module. This is the basis of the reactive behaviour of the planner. Finally, the planner must know what the initial and goal states are. When the system is first started, the initial state in the planner is equal to the initial state of the plant. When the system is already running and replanning is ordered, the current state of the plant becomes the new initial state in the planner. The monitoring function takes care of determining that state.

The goal state, however, has to be entered by the user, before any solution plan can be generated and any action taken. It forms the only run-time user input. The goal state determines the major objectives of the supervision and control. Once it has been entered, the system can proceed to control, monitor and supervise the plant, *without any further assistance from the user.*

The planning module uses the abstract model of the world (initial state), the abstract models of the actions (operators) and the abstract model of the objective (goal state) to try out different sequences of the available actions, and find the right one for the given conditions. It thus reasons in abstract terms to arrive at a solution to the real problem. This solution is produced as a sequential list of actions and is guaranteed to take the system to the required objective (or as close as possible), provided the current conditions remain unchanged.

5.1.4 Action-Executing Module

The *action-executing* module coordinates the physical actions of the system, in accordance with the solution plan it receives from the planner. This module has an associated knowledge-base where all the actions available to the system are stored in the form of ready-to-run procedures. The task of managing these actions at run-time is a complex operation. Different computers and operating systems will impose different restrictions on the software implementation of this task.

In addition, just like the monitoring module, it is application dependent. The action procedures will depend on the characteristics of the various elements in the system (e.g., what actuators and controllers are used). This is to be expected as the module, together with the actions it commands, can be thought of as the reverse of data concentration involved in monitoring. The process of implementing physical actions according to an abstract plan, involves a translation of the high-level symbolic information into low-level discrete-time signals. These can then be transformed into continuous time signals by the write module.

The signals coming out of the actions may be used to switch on and off controllers, provide setpoints and tuning parameters to the controllers, directly drive plant actuators or communicate with other subsystems.

Consider the following example of a simple control action procedure:

Switch_to_backup_control

```
backup_controller.setpoint = main_controller.setpoint
backup_controller.gain = main_controller.gain
main_controller.status = OFF_LINE
backup_controller.status = ON_LINE
```

When this procedure is executed by the action-executing module, it first sets the parameters of the backup controller to those of the main controller. It then switches the main controller off (presumably because a fault has been detected in its functioning) and switches the backup controller on-line. Real actions will typically be more complex, particularly those responsible for control tasks.

5.1.5 Structure of the Supervisor

To summarize the above points, Figure 5.3 shows the detailed structure of the proposed supervisory system. As can be seen, this is a closed-loop system with the monitoring module being the feedback path. Because of this, the system can evaluate the effects of its previous actions and react if these effects turn out to be other than expected. The proposed system makes full use of this feature, to ensure fully automatic supervision of a dynamic plant with arbitrary and unpredictable changes.

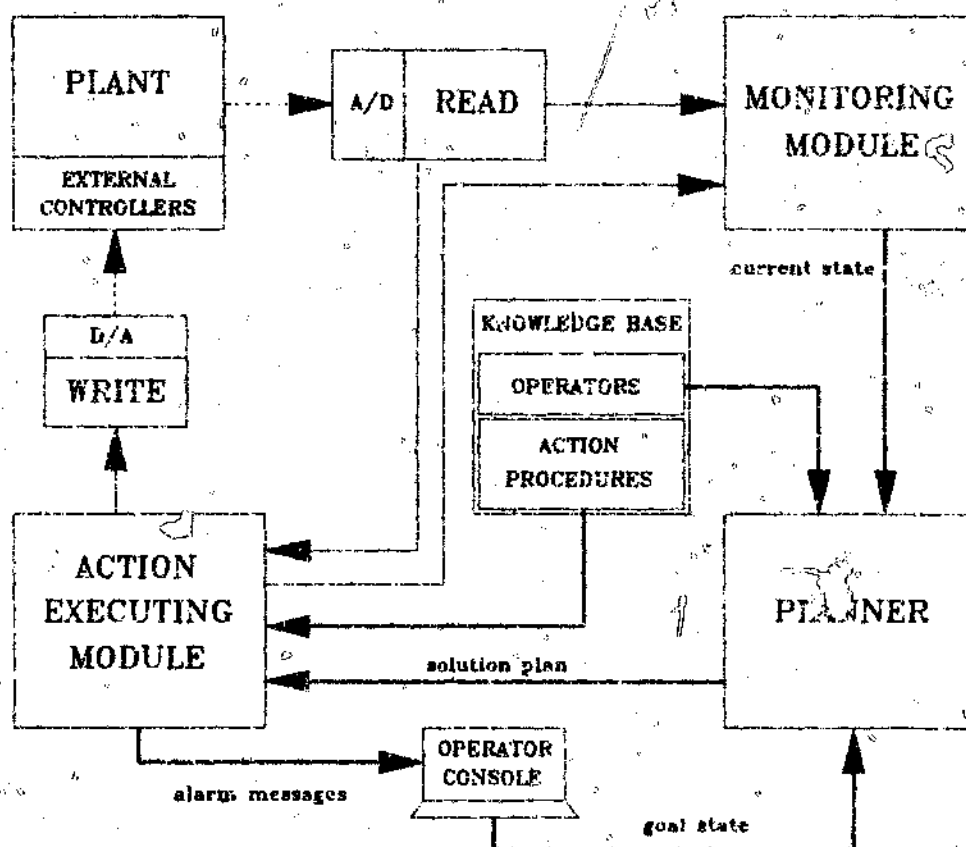


Figure 5.3: Detailed structure of the supervisory system.

As there may be some actions performing low-level feedback control, Figure 5.3 shows a standard feedback path for analogue signals, going from the read module to the action-executing module.

5.2 Run-Time Requirements of the System

Supervision is a complex process requiring concurrent execution of multiple activities. This gives rise to the need for a multiprocessing environment. Because the proposed system does not have to be implemented on a distributed architecture (i.e., it may use a single microprocessor), the issue of *apparent* multiprocessing must be addressed explicitly.

5.2.1 Scheduling of the CPU Time

Apparent multiprocessing occurs when two or more processes are competing for the attention of the central processing unit (CPU). To enable concurrent execution of these processes, a CPU scheduler is used and it can take one of two forms.

Preemptive scheduling refers to the case where a higher priority task (process) can interrupt the currently active task, without returning control to it. Alternatively, the currently active task will be interrupted when its time is up (i.e., its CPU time-slice is used up). Preemptive scheduling is advantageous in situations where certain tasks have definite priorities and, particularly, where certain tasks are critical and cannot wait for the current process to run to its completion.

Non-preemptive scheduling means that the currently active task decides when to yield control to another task. Different tasks may still have different priorities which are used to order them in the queue (to determine the order of execution). Non-preemptive CPU scheduling is significantly less complicated to implement and it makes the resulting behaviour totally deterministic. On the other hand, it has no mechanism for interrupting the active process.

A round-robin non-preemptive scheduling is the simplest type of CPU scheduling as it allows no interrupts and treats all tasks with uniform priority. The tasks are stored in a circular list and run sequentially, always yielding to the next task on the list.

5.2.2 Round-Robin Non-Preemptive System Executive

A multitasking system executive for concurrent programming is developed and explained in [Lun *et al.*, 90]. It is significant, because it runs under the DOS operat-

ing system on an IBM-PC compatible computer and has been successfully tested in control applications. It is implemented entirely in the C programming language. It can concurrently run different routines within one program using round-robin non-preemptive scheduling.

Intertask switching is implemented with the ANSI C standard functions *setjmp()* and *longjmp()*. The *longjmp()* function can be thought of as a far or nonlocal goto. The *setjmp()* function is used to save the task state (task context) just before a *longjmp* is performed, so that it is possible to return to the original task later. The task context consists of the contents of the CPU registers, the CPU flags and the stack and base pointers stored in a special data structure called *jmp_buf*.

A call to the *setjmp()* function returns zero. If the *longjmp()* function is later used to jump to this task (restoring the previously saved context), it will reenter the code just after the *setjmp()* and it will appear as if the *setjmp()* function has just returned with a nonzero value. A combination of *setjmp()* and *longjmp()* is typically used by C programmers for exception handling.

To enable multitasking within a program, the tasks are defined as C functions and arranged in a circular list. Each task has an infinite loop inside, so that it runs forever. However, just before the end of that loop the *yield()* function makes sure that control will be passed on to the next task on the list. Because all the tasks have infinite loops inside, they are all active concurrently, although only one is receiving CPU's attention at any time. The *yield()* function is defined below:

```
yield()
{
    if ( !setjmp(current_task) )
        longjmp(next_task)
}
```

When put at the end of a task, it does the following:

If the task has just executed, then jump to the next task;
else if a jump from the preceding task has just been made,
then execute the current task.

Figure 5.1 illustrates the fixed and deterministic nature of the system executive. The circular list, which is created first, defines the order in which the tasks will be

run and that order remains static. Therefore, the number and the order of the tasks must be known in advance.

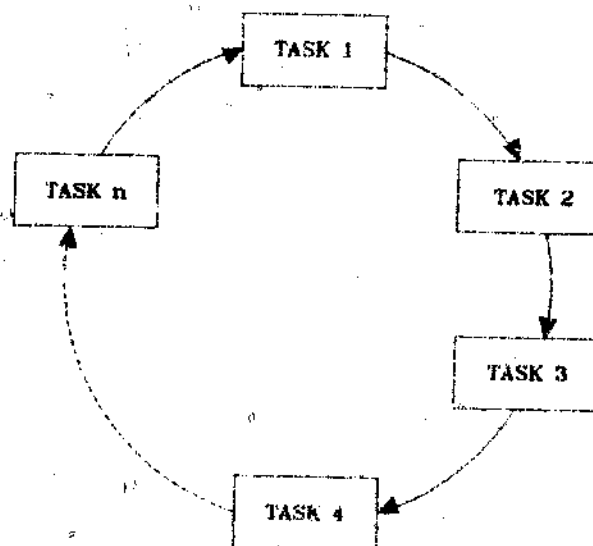


Figure 5.4: Round-robin non-preemptive scheduling using the system executive.

Once the system is started, all the tasks will appear to run in parallel although, in reality, each will run periodically with a period equal to the time it takes to go round the entire list. Therefore, all the tasks will run with the same frequency. If any task within the list uses a variable time-slice, then the period of the entire list will change. The lack of explicit time management in the system executive means that it may be impossible to determine the frequency with which the tasks will execute.

5.2.3 Multitasking Requirements of the Supervisor

Consider now the multitasking requirements of the supervisory system. If low-level controllers, the monitoring module and the planner run on a single computer, a careful approach to scheduling must be adopted.

As mentioned before, there could be three different operating frequencies for the above tasks. A digital controller contains a discrete-time equivalent of a continuous-time control function. The mathematical derivation of this discrete function assumes that the period with which the incoming signals are sampled is constant and conforms to the Nyquist theorem. This period must be calculated for a given plant and then strictly adhered to, to ensure correct functioning of the controller.

The monitoring module does not have to use the same sampling period. In fact, there is very little theory to guide one's choice in this aspect. It is still a real-time operation which needs to scan its inputs often enough to detect any important changes. We chose to make the monitoring module run at the Nyquist frequency, i.e., to use the same data as the controllers. This ensures that it receives all theoretically available information and, therefore, is the safest solution.

The planning module works in a totally different dimension. It does not sample the continuous-time variables of the system. It uses the symbolic state description made available to it by the monitoring module. A digital controller uses a sample-and-hold circuit to gather data by frequency-based sampling. The planner could be described as using the monitor to gather information by event-based sampling.

There is no need for the planner to run with a fixed frequency. Replanning is totally unpredictable, because it occurs in response to critical changes in the system, which are themselves unpredictable. Additionally, the planner is a few orders of magnitude more computationally intensive than the other modules. In general, it may easily take much longer to execute than the sampling period of the low-level controllers.

If the AI-based part of the system takes too much time in relation to the basic sampling period, then it cannot be allowed to run to its completion, as it would disturb the operation of the controllers. In such a case, the tasks of sampling new data, calculating a control signal and writing it to the plant should be awarded higher priority and the right to interrupt the slower planning module. Generally, the complexity involved in the process of supervising would require a preemptive CPU scheduler (if parallel processing is unachievable). However, design of a preemptive scheduler is a complicated issue, while DOS presents many serious limitations further increasing the level of difficulty [Lun *et al.*, 90].

5.2.4 The Multitasking Kernel

Here we take a different approach, one which satisfies the needs of the proposed supervisor, while being much simpler than a preemptive scheduler. It is based on the multitasking system executive described earlier and shares much of its simplicity, but it has three important additions. They are: explicit management of time, task priorities and dynamic reordering of tasks. This method works only for a system where the planning module runs fast enough to fit within one sampling cycle.

The tasks of sampling the plant, performing low-level control, writing the control signal to the plant and monitoring the incoming information all run in this fixed order, at a frequency determined by the sampling period. Therefore, they are all awarded a higher priority status. All the other tasks, including the planner as well as most supervisory actions, do not have to run at a fixed frequency and hence are treated with lower priority.

A special function, residing at the beginning of all the lower priority tasks, decides if a task should run or be bypassed. It does so by comparing the current time with the time of the next sample, to check if there is enough time for the task to run. If the next sample is almost due, the task and all subsequent low priority tasks will be bypassed, so that control is returned to the sampling task in good time.

The read (sampling) task has a different function at its beginning. This function is used for frequency synchronization. It compares the current time with that of the next sample and if the two are close enough, it causes the system to wait for a period equal to the calculated difference. Thus, it ensures that sampling is always done at precisely the same frequency. The controlling, writing and monitoring tasks can then follow (in this order) to utilize the newly sampled data.

The other shortcoming of the basic system executive is the static nature of its task ordering. Every cycle, the same tasks are run in the same order. The supervisory system is, in contrast, a highly dynamic one. Each time the planner comes up with a new solution plan, a reconfiguration of the active actions (which like tasks from the point of view of the scheduler) is called for.

To extend the system executive to allow for dynamic reordering, the pre-entered list of tasks includes the full set of actions, in addition to the planning, monitoring, reading and writing tasks. When control is passed to an action from the preceding task on the list, the action checks the latest solution plan to see if it is supposed to execute. It then either executes or immediately yields control to the next task.

Furthermore, there are some actions which require continuous running and others which only execute once. They are distinguished by the means of dedicated flags. A sequential action (which executes once) takes itself off the solution plan as soon as it has run, so that it will not run in the next cycle. A parallel action (which executes continuously) also does that, but at the same time it sets its dedicated flag to enable it to run on every subsequent cycle, for as long as the flag is set.

These extensions of the system executive result in a multitasking kernel capable of

handling the supervisory process. It is still relatively simple, can run under DOS and is deterministic, though no longer predictable. To help explain how it works in practice, a simple example follows.

Consider a system with the read, write, monitor and plan tasks, in addition to three actions. These are: use_control_A, use_control_B and alert. Then, the circular list will contain all these tasks, with the low-level control actions located between the read and write tasks. This is shown in Figure 5.5.

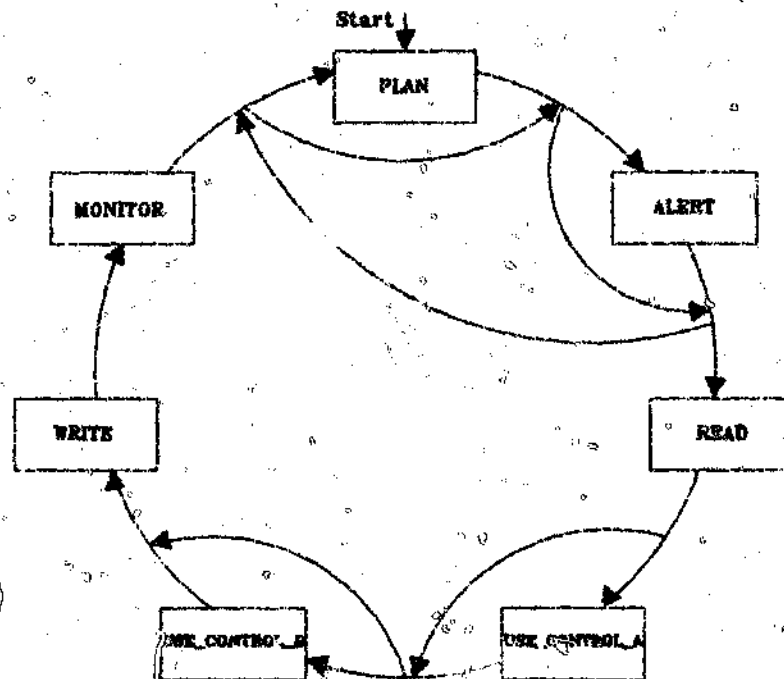


Figure 5.5: An example application using the multitasking kernel.

When the system is started, it goes through initialization routines which include obtaining the initial and goal states while it is still off-line. The planning task is the first one on the list and it uses these states to find a solution plan. Assume that the first solution plan has just one item: use_control_A. Therefore, the next task on the list, alert, would be bypassed as it does not feature in the plan. The read task would execute without any need for synchronizing (as it is the first sample) and set an associated flag to be used by the control actions (e.g., new_read_flag).

The use_control_A action would execute next, as it finds itself in the plan and also, because newly sampled data is available (new_read_flag is set). This task would calculate a control signal on the basis of the sampled data. It won't set an associated

new_write_flag to indicate that new data is available for writing to the plant, and a flag to be used by the monitor task (e.g., new_data_flag). As it is a parallel action, it would also set another flag (e.g., control_A_flag) to ensure its continuous running.

The next task on the list is use_control_B which will not execute, because it is not in the solution. The write task will run, as the write_flag is on. Similarly, the monitor task will run, since the new_data_flag is on. The new_data_flag, new_read_flag and new_write_flag are all reset by the respective task, right after it executes. The monitor task, based on its situation assessment, can trigger replanning. In this case it will not, as it is impossible to perform a serious situation assessment after only one sample. Thus, the plan task would be bypassed and so would be the alert action. Next time the read task is entered, it checks the time and if the next sample is due, it synchronizes its timing and executes (otherwise the entire part of the system which works on sampled data would be bypassed).

The sequence of events does not change until, at some stage, the monitor task notices that controller A is not performing well and it orders replanning by setting a special flag (e.g., plan_flag). The plan task checks if it has enough time to execute and if so, it creates a new plan. This could be to first alert of the change and then switch to controller B. If there is still time before the next sample, the alert action will execute (it is in the solution plan), but only once as it is a sequential action. After the next read, the use_control_B action would run instead of use_control_A.

5.3 Conclusion

This chapter has shown how the planner can be connected to other modules, to form a real-time intelligent supervisory control system. The run-time requirements of the proposed architecture have been discussed with a detailed look at the issues of apparent multitasking and scheduling.

Chapter 6

Outline of Implementation

This chapter contains a systematic high-level description of how the entire supervisory system is implemented in software, and how the program functions. Some of the major parts of the system have already been described in the context of planning and supervision. The domain-dependent parts will be discussed in the next chapter. Here, the system will be looked at from the software point of view, with emphasis on the algorithm design and data flow between different routines. A listing of the source code appears in the appendices.

6.1 Data Structures

First, the most important data structures are described. The relevant code appears in Appendix A. Names of data structures and their types are given in *italics*.

6.1.1 Global Data Structures for Intertask Communication

Communication between concurrent tasks can be achieved by means of either message passing or global data tables. Message passing increases the complexity of the system, but helps to protect the integrity of the data. Global data is the simpler solution as it does not require a special mechanism for passing messages (with a corresponding overhead). However, it is more difficult to ensure the correctness of global data.

We use global data structures to communicate between concurrent tasks, for the reason of simplicity. To ensure data integrity, access to each structure is tightly controlled. Writing rights are severely limited, to just one task in most cases, while other tasks only have reading rights. There are four global data structures, each consisting internally of many slots with predetermined access rights.

Plant variables

This data structure is of type *numeric data*, which means that it contains all the numerical variables needed by the supervisor. The internal details of this data will depend on the actual plant and actions used, but it can be divided into two main parts. One of them serves as a storage place for all the incoming data, i.e., the unprocessed data (e.g., plant measurements). Writing rights to this part are granted exclusively to the read task.

The other part stores the remaining numerical data which has already been processed and waits to be sent to the plant (e.g., actuating signal) or further processed by another task (e.g., setpoint). The writing rights to each variable are limited to one or a few related tasks, depending on the problem (e.g., there could be a few control tasks writing to a single variable, but only one at a time). The *plant variables* structure is used by the monitoring, read, write and control tasks.

State of plant

This data structure is of type *symbolic data*. It contains the entire symbolic description of the current state of the system, as needed by the planner. Again, the exact contents of this structure are domain-dependent. The monitoring task has exclusive writing rights to most of this structure (e.g., variables representing equipment failures). A few slots in this structure are updated by other tasks (e.g., a variable for representing the currently active controller). This data is used to create an abstract model of the system, for use in the planning process.

Task flags

This is a structure of type *flag table* and is the storing place for the flags used in dynamic reconfiguring of concurrent tasks. Only the read task is time synchronized

and totally independent of the other tasks. All the other tasks, including some actions, are enabled or disabled with these flags.

Plan_stack

This structure is of type *search_space_node*. This is where the solution plan is stored. The planning task has, therefore, exclusive writing right to this data. *Plan_stack* is used extensively at each stage of the search process. All the actions read this data to check if they are the next action to be executed. Whereas the supervisor communicates with the planner by means of the *state_of_plant* structure, *plan_stack* is used to transfer information in the reverse direction.

6.1.2 Other Global Data Structures

Operator_definition and operator_buffer

These are both of type *operator*. They form the knowledge base for the AI part of the system, containing the definition of uninitialized and initialized operators respectively. Only the programmer can change the *operator_definition* structure by providing a different set of operators for a different problem. *Operator_buffer* contains an initialized copy of these operators, which is actually used in planning. The scope of both of these structures is limited to the planning task (by use and not by declaration).

6.1.3 Local Data Structures

Present_state

This structure is only defined within the planning task. It contains an abstract representation of the current state of the plant, just before planning begins (i.e., the initial state). During planning, it is used to store temporarily all the intermediate states created by the search algorithm.

Goalstate

Like *present state*, *goal state* is only used by the planning task. It has to be entered by the user, as the only run-time input. Its contents are used in the planning process and do not change until another goal state is entered by the user.

6.2 Top-Down Analysis of the Software Routines

In the following sections, the most significant routines of the program are analysed. To keep the discussion free from the terminology of a specific computer language, each important routine (subprogram) is defined in terms of its inputs, outputs and a descriptive algorithm. Names of routines are given in SMALL CAPS.

The routines described in this section show how each of the system's tasks is implemented and how they are all initialized. The tasks can be listed according to the previously developed architecture for the supervisory system.

- o Actions
- o Monitor
- o Read
- o Write
- o Plan

Plan is the only truly general-purpose task. Read, write and monitor are all problem-specific in their internal implementation, but their main features can be generalized. They are always active and their major functions do not change. Actions are entirely problem-specific.

6.2.1 Actions

Actions are all those tasks that implement supervisory actions for a given control problem. Their number, requirements, abilities and functioning are totally domain-dependent. The only aspect that does not change, is their overall task-like structure.

On the other hand, actions have a relatively simple logic structure (though they may have very complex mathematics). Therefore, a generic action can easily be represented if its function is disregarded. For the actual code, refer to Appendix B.

Generic Action

Inputs

- Plant variables
- State of plant
- Task flags
- Plan stack

Outputs

- Plant variables
- State of plant
- Task flags
- Order outputs

Algorithm

1. Check plan stack

- if next action on stack is this action then continue
- else goto 5

2. TIME UP

- if next sample due then goto 5
- else continue

3. Perform the functions of this action

4. Point to the next action in *plan_stack*

5. YIELD

6. Goto 1

An action will always use the *plan_stack* structure and optionally the *task_flag* structure to determine if it should execute. It can also use *plant_variables* and *state_of_plant*. It calls the *TIME_UP* routine to check if there is enough time for it to execute. If the action is not supposed to run, because it is not the next item in the solution plan or a necessary flag is not set, or there is simply not enough time, it will immediately YIELD to the next task.

If the action does execute, it performs its functions by modifying one or more of *plant_variables*, *state_of_plant* and other outputs (e.g., a direct screen output). It then ensures that next action is pointed to, on the *plan_stack* and, finally, calls the YIELD routine.

TIME_UP

This routine returns a boolean value depending on how much time is left before the next sample is due. If there is ample time for the calling action to execute to its completion without disturbing the sampling period, it will return *TRUE*.

YIELD

The YIELD routine has already been described in detail in the previous chapter. It is the mechanism by which a task can be put on hold while the rest of the circular list is executed.

6.2.2 Monitor

The functional aspect of monitoring will be treated in the next chapter; the relevant source code is given in Appendix B.

MONITOR

Inputs

- Plant variables
- State of plant
- Task flag

Outputs

- State of plant
- Task flag

Algorithm

1. Check new data flag
 - If it has changed
 - Go to step 2
2. Perform the monitoring functions
3. Reset new data flag to zero
4. Yield
5. End 1

MONITOR uses the available information about the system from the plant variables and state of plant structures. It also checks if the new data flag in the task flag structure has been set (by the read task). If so, the monitoring functions are executed. This is usually a complex procedure, performing situation assessment and diagnosis. At the end of it, the state of plant structure is updated. The MONITOR routine is also the only one which can set the plan flag (to trigger replanning). It does so in response to certain undesirable conditions. After executing, it resets the new data flag and YIELDS.

6.2.3 Read

The code for this task appears in Appendix B, with some of its lower-level functions in Appendix D.

READ_PLANT

Outputs

- *Plant variables*
- *Task flags*

Algorithm

1. SYNCHRONIZE SAMPLING

- If sampling not yet due then goto 5
- else continue

2. READ_GAS into plant variables

3. Write time into plant variable

4. Set new read flag in task flags

5. YIELD

6. Goto 1

This routine does not have any inputs but it writes into both the *plant variables* and the *task flags* structures. It starts off by calling the *SYNCHRONIZE SAMPLING* routine which checks the time and synchronizes next sampling if necessary. If there is still time for other tasks to run, the body of the routine is bypassed. Otherwise, the *READ_GAS* routine samples the plant and writes the results into *plant variables*. The exact time of the sampling is also written there, to eliminate compound errors. Finally, the new read flag in the *task flags* structure is set and the routine exits through *YIELD*.

Both sensors are based on a standard industrial differential-pressure transducer. Their output is a current signal in the 4-20mA range. For the level sensor, its current is directly proportional to the level. For the flowrate sensor, its current is proportional to the *square* of the actual flowrate. The inlet valve is a standard industrial diaphragm valve, supplied with 140kPa pressurized air. It is activated through an electro-pneumatic converter which accepts an input current within 4-20mA and produces a proportional pressure. This is a low-precision piece of instrument, characterized by large nonlinearities and hysteresis.

manual outlet valve can be used to inject disturbances into the plant, by varying the load. When it is fully opened, the pump has too little throughput to maintain an equilibrium. There is also an electric signal converter which converts a 4-20mA current into a 0-10V voltage signal and vice versa.

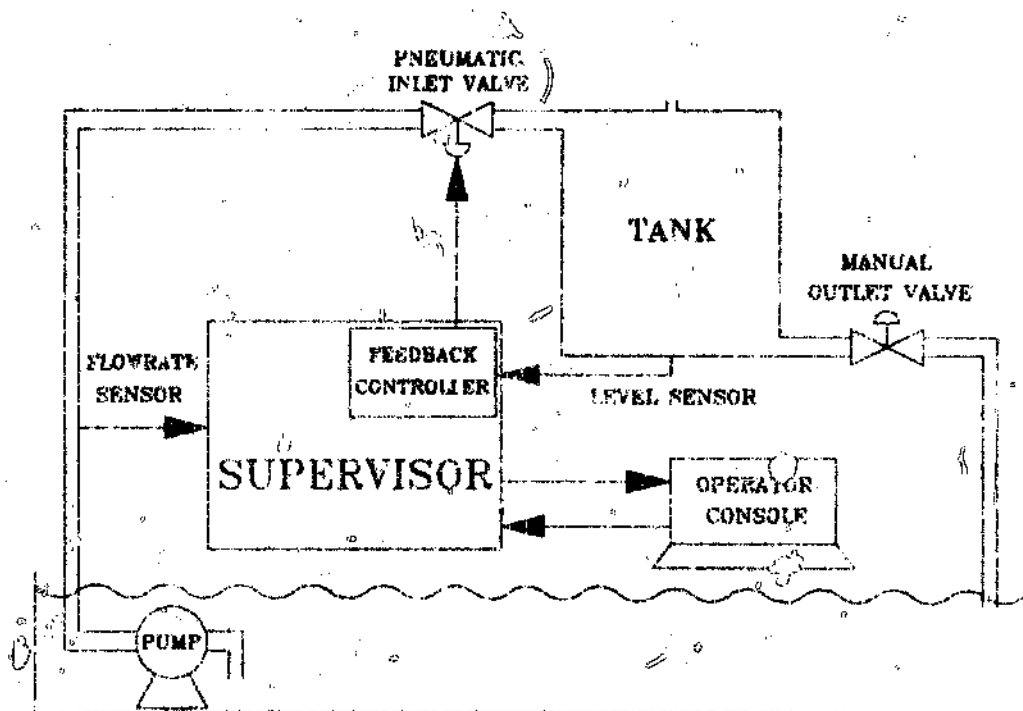


Figure 7.1: The plant and the supervisory system.

The plant was initially controlled by an analogue electronic PI controller. The process of replacing the analogue control system with a distributed direct digital PI controller, is described in [Zagnoev, 91]. The first stage of that process was the analysis of the plant by a step test.

The equation of a first-order plant is:

CHRONIZE SAMPLING

This routine gets the current time and compares it with the time when the next sample is due. If the difference is large enough to accommodate another task, zero is returned. Otherwise, the process is suspended until the moment when sampling has to occur. This ensures that samples are always taken at equal intervals.

READ CARD

This routine has to access the hardware modules responsible for analogue to digital conversion. It first sets up the A/D card to perform sampling and when samples are ready it stores them in the plant variables structure.

3.2.4 Write

The code for this task also appears in Appendix B and, in part, in Appendix D.

WRITE TO PLANT

Inputs

- o *Plant variables*

- o *Task flags*

Outputs

- o *Task flags*

Algorithm

1. Check new.write_flag
 - o if set then continue

Wait_for_steady_state

Pos CONDITIONS: active monitoring setpoint_goal	Wait_for_steady_state	ADD EFFECTS: level_goal
NEG CONDITIONS: alarm_condition fatal_failure level_goal large_disturbance high_gain low_integral derivative_action		DEL EFFECTS: level_pres overload valve_failure

This action is similar in principle to wait_for_proximity, but is always the last action in a solution plan. It can only be executed if the controller is set for the low-gain, smooth control mode. It cannot start if there is a large disturbance or a safety risk or, like before, if there is an alarm condition.

It uses the monitoring function to determine when a steady state condition is reached. At that moment, it informs the user that the level has stabilized at the setpoint.

Sound_alarm

Pos CONDITIONS: alarm_condition	Sound_alarm	ADD EFFECTS:
NEG CONDITIONS:		DEL EFFECTS: alarm_condition

This action is activated to send a visual and audible message to the console, to get the attention of the user. The preconditions are formulated in such a way, as to give it the highest priority.

$$P(s) = \frac{k}{\tau s + 1}$$

The values derived for the open-loop system were:

- Time constant (τ) = 572 seconds
- Gain (k) = 1.67

The chosen values for the digital PI controller were:

- Controller gain (K_p) = 2
- Integral time constant (T_i) = 300

As the analysis will not be repeated here, the above values will be referenced.

Figure 7.1 illustrates all the major parts of the plant and how a feedback controller and the supervisor are connected to it. It clearly shows that the basic controller works in a single-input, single-output configuration. It receives the measured level of water in the tank from the level sensor, finds the error between it and the preentered reference value (setpoint), and calculates a control signal according to its algorithm. This control signal is used to drive the inlet valve.

In contrast, the supervisor is a multi-input, multi-output device. It reads the flowrate in the inlet pipe, in addition to the level in the tank, and also receives information via the user console. Its outputs are used to drive the inlet valve and to send signals and messages to the screen. In other words, the supervisor utilizes all the available channels to gather the maximum amount of information about the system. It also uses all the available channels to influence the system in the most effective way.

7.2 Control and Supervisory Objectives

The basic control objective is clear and simple—keeping the level in the tank at a predetermined value. This makes it a typical example of regulatory control which is the most common type of control in processing plants. The supervisory objectives are more arbitrary. Consider the following modes of operation for the plant equipped with a feedback controller.

- o else goto 4
- 2. WRITE_TO_CARD from *plant_variables*
- 3. Reset new_write_flag in *task_flags*
- 4. YIELD
- 5. Goto 1

This routine reads both the *plant_variables* and the *task_flags* structures and writes back into *task_flags*. The availability of new data for writing (e.g., a new actuating signal) will depend on some action whose function is to provide such output. That action will also be responsible for setting the new_write_flag. WRITE_TO_PLANT executes by calling the WRITE_TO_CARD routine, which is the reverse of READ_CARD, and then resetting the new_write_flag.

WRITE_TO_CARD

WRITE_TO_CARD is used to set up the D/A card and then to drive it, in order to translate some of the discrete values from *plant_variables* into continuous-time analogue voltage signals. This, together with the READ_CARD routine, is the interface between the digital computer and the physical plant.

6.2.5 Plan

Planning is by far the most complex and computationally demanding part of the supervisory process. The routines given below, and many other lower-level functions, are included in Appendix C.

PLAN

Inputs

- o *State_of_plant*
- o *Task_flags*

Emergency shutdown

Pos CONDITIONS: fatal_failure	Emergency shutdown	ADD EFFECTS: plant_down
NEG CONDITIONS: alarm_condition plant_down active_standard		DEL EFFECTS: active_* setpoint_* high_gain low_integral derivative_action large_disturbance supervisory_control

The supervisor takes this action only in a real emergency, when *fatal_failure* is reported. This means that continuing with the plant operation could be unsafe. The only alternative for the supervisor is to take over control, force the inlet valve to close and then disable all other actions. Thus, the plant and the control system are effectively shut down. This action is accompanied by a suitable message sent to the user, to inform him of the events.

7.3.3 Other Tasks

Besides the above actions and the basic tasks of planning, reading and writing there is another task called *abort_all*. It allows the user to switch off the system at any time and return to DOS, or to suspend the multitasking process temporarily (by pressing any key). It has another important function, as it allows the user to simulate a failure of the active controller (by pressing the F key). This task works by regularly scanning the keyboard of the user console for any buffered inputs.

7.4 Test Results

7.4.1 User Interface

Our supervisory system produces a comprehensive real-time display via the console screen. It must be emphasized that such comprehensive information is necessary, as it is an *experimental* system whose performance has to be thoroughly recorded and

- o Normal operating conditions.

- small disturbances
- large disturbances

- o Abnormal operating conditions.

- overload
- valve failure
- sensor failure
- controller malfunction

Normal operating conditions refer to the case in which the basic controller is equal to the task of regulating the plant. Abnormal operating conditions are beyond the capabilities of the controller. Therefore, the control objective will not be achieved and instead a safety risk may result. Overload or valve failure (unpowered valve closes) will lead to the level dropping below the setpoint or even to the emptying of the tank. More serious is a sensor or controller failure, which could lead to an overflow of the tank which is regarded as an unsafe condition. In theory, there could be many more potential problems, but the above ones represent the most likely cases.

Recall that the aims of supervision should be, firstly, to maximize safety and, secondly, to maximize production rate. Specific objectives should be set up in such a way, as to test the performance of the intelligent supervisory control system under both normal and abnormal conditions. We define them as follows:

1. Ensure safe operation at all times.
2. Keep the plant running despite failures (unless it endangers safety).
3. Provide better quality of control than a conventional controller.
 - (a) better disturbance rejection
 - (b) better steady state performance

The last objective needs a little explanation. In terms of PID control, a disturbance can be dealt with more effectively by increasing the controller gain (proportional term) and also by using derivative action to introduce some phase-lead (speed up response).

- o *Operator_definition*
- o *Operator_buffer*
- o *Present_state*
- o *Goal_state*

Outputs

- o *Plan_stack*
- o *Task_flags*

Algorithm

1. Define *terminal_state*
2. GET_GOAL into *goal_state*
3. Check *plan_flag*
 - o If set then continue
 - o else goto 9
4. TIME_UP
 - o if next sample due then goto 9
 - o else continue
5. MAKE_ABSTRACT_STATE into *present_state*
6. INITIALIZE_OPERATORS
7. SEARCH from *present_state* to *goal_state* or *terminal_state*
8. Reset *plan_flag* in *task_flags*
9. YIELD
10. Goto 3

evaluated. The idea of intelligent supervisory control suggests a minimum interface with the user. Unlike most expert systems, the supervisor needs no consultation with the user at run-time, other than signalling alarms. Intelligent advisory systems have elaborate user interfaces taking up anything between a third and a half of all code, as pointed out in [Bobrow et al., 88].

An example of a run-time screen (all results were obtained with a screen-saver) is shown in Figure 7.3. The top part of it is devoted to the supervisory functions, while the bottom part shows the low-level control. At the very top, a rectangular area is reserved for special messages. These include alarm, steady state and shutdown messages. The first of the four windows shows the terms of the starting state. It is a symbolic representation of the state of the system at the moment when planning was started. The next window contains the terms of the goal state, as entered by the user.

The third window is called the expected state, because it describes the state of the system that *should* result from the implementation of the solution plan. Normally, the expected state differs from the goal state only in some unimportant details (it uses more terms to describe the same situation). But having two separate windows is deliberate. There may be cases where the actual final state will be totally unlike the goal state. That happens if the goal turns out to be unreachable and the planner reverts to a plain data-driven approach.

The last window contains a numbered sequence of actions which the system needs to take in order to achieve the expected state. This is the solution plan. To show which action the system is currently busy with, an arrow is provided, pointing to its number.

The large window in the lower part of the screen shows the continuously changing analogue data. Its top line gives the time that has passed since the first sample was taken, the number of samples taken so far and the sampling period used. Both graphs are calibrated in percents versus time in minutes. The top graph shows the measured level in the tank (in bold) and the setpoint, as functions of time. The graph below shows the output of the controller, i.e., the actuating signal (in bold) and the measured flowrate in the inlet pipe, as functions of time.

The graph at the very bottom of the screen shows whether the controller is operating in the high gain or the low gain mode. This graph is only used by the advanced and backup controllers. Finally, every time a new plan is generated a small letter P is printed on the x-axis of the top graph.

The **PLAN** routine first defines the *terminal_state* and gets the user to fill the *goal_state* by calling the **GET_GOAL** routine. This happens only the first time that **PLAN** executes. Next, the routine checks if the *plan_flag* has been set by **MONITOR**. If so, it checks further if there is enough time for it to run, by calling **TIME_UP**. If it decides to run, it first uses **MAKE_ABSTRACT_STATE** to fill the *present_state* structure. Then, the **INITIALIZE_OPERATORS** routine is called. The call to **SEARCH** is what effectively starts the planning process. It carries on until a solution plan is found or all possibilities exhausted. At the end of the **PLAN** routine, *plan_stack* contains the solution. The *plan_flag* is then reset and **YIELD** called.

GET_GOAL

GET_GOAL allows the user to enter a goal state at run-time. It is stored in the *goal_state* structure and used by the **SEARCH** routine to find a solution plan.

MAKE_ABSTRACT_STATE

This routine is used by **PLAN** to fill the *present_state* structure with the current state of the plant, to become the initial (starting) state for the searching algorithm. The state is created, based on the information in the *state_of_plant* structure.

INITIALIZE_OPERATORS

Inputs

- *Operator_definition*
- *Present_state*
- *Goal_state*

Outputs

- *Operator_buffer*

On the other hand, high gain is undesirable near the setpoint, as it can cause unnecessary wear of the actuator for even very small variations in the controlled variable. High gain also tends to introduce oscillatory behaviour. Similarly, the derivative term is often turned off completely, as it has a tendency to amplify high frequency sensor noise. It is not a purpose of this work to investigate the conventional PID controller, but it should be realized that tuning such a controller involves a large degree of compromise.

7.3 The Supervisory System

7.3.1 Monitoring

The monitoring module is entirely problem-specific and would have to be redesigned for each new problem. This is to be expected as each different plant/conditions and malfunctions in a specific way, depending on the range, quality and even age of the hardware used. For the level control plant, the following equipment is constantly monitored and diagnosed:

- Level sensor.
- Feedback controller.
- Inlet valve.
- Level in tank.

Level sensor. This is probably the most crucial element of the system. The plant does not have a second level sensor, so all its information about the level comes from one source only. Its failure renders the entire basic control loop useless. But even more importantly, faulty readings may very easily lead to an overflow of the tank, if the basic controller is left on its own. Therefore, the monitoring function continuously checks the signal arriving from the level sensor.

If the signal is found to be beyond its standard allowable range, then a failure is reported. The rate of change of the signal is also monitored and if it exceeds the maximum rate at which the tank can be drained, then the sensor is assumed faulty. In either case, two new terms are added to the abstract description of the state of the plant: *sensor failure* and *fatal failure*. The first one is informative, while the

7.4.2 Standard PI Control

In this test, the conventional PI controller, which was used to control the plant previously, is evaluated. Its parameters are left unchanged, gain at 2 and integral time constant at 300. Figure 7.3 shows a record of the entire testing session.

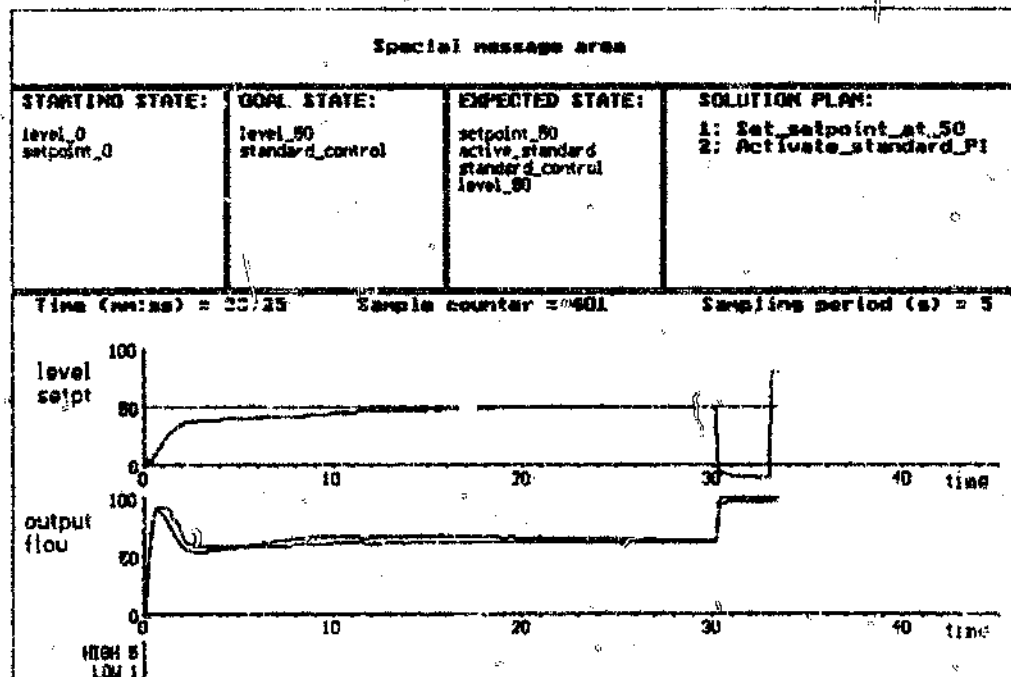


Figure 7.3: Standard PI control.

The process begins with empty tank, closed inlet valve and the outlet valve at 90%. When prompted, the following goal state is entered:

- o level_50
- o standard_control

The solution plan is, in this case, quite trivial. It takes the system about 4 minutes to get within 10% of the setpoint and about 15 minutes to reach a steady state. The control action (on the lower graph) is smooth and quick. The difference between the actuating signal and the actual flowrate is caused by the inconsistent and low precision behaviour of the pneumatic valve. This happens despite the fact that a special procedure is used to linearize the valve. However, it does not eliminate the inherent hysteresis.

second one indicates that there is a safety risk. In addition, the *plan_flag* is set to trigger immediate replanning.

Feedback controller. The supervisor has two controllers at its disposal. The main one is called the *advanced* controller and the other one is the *backup* controller. The failure of either one is acknowledged by adding *advanced_failure* or *backup_failure* respectively, to the state description. However, the monitoring function knows that there are only two controllers, so if both of them fail (a very unlikely case) it will also add *fatal_failure*. Controller malfunction is the only case which needs to be simulated. An actual fault could easily occur with an external controller, but cannot physically happen with the software implemented controllers. In case of a controller failure, replanning is triggered.

Inlet valve. The most fragile part of the pneumatic valve is the pressurized air supply. The valve itself is spring loaded to automatically close, should the air supply be broken. Therefore, even if a failure occurs, there is no danger to safety. To detect valve failure, the monitoring function needs to make use of an extra sensor—the flowrate sensor—which is not a part of the basic control loop. This illustrates a more general principle, that a supervisor needs more sensory inputs, because it is responsible for so much more.

This extra sensor allows continuous monitoring of the flowrate in the inlet pipe (which is a function of the valve position). Simultaneously, the valve actuating signal is monitored. If the controller output is driving the valve open, but there is no flow in the pipe, then the valve is diagnosed faulty and the term *valve_failure* added to the state. Replanning is then requested.

Level in tank. To enable the system to improve both the disturbance rejection and the steady state performance, the monitor translates the level in the tank into symbolic descriptions. It recognizes three important cases, based not only on the actual level, but also on the rate of change in the level. These zones are illustrated in Figure 7.2, for an arbitrary setpoint.

(B-C) Level is in the proximity zone, i.e., it is approaching a steady state condition. The monitoring function will add the term *proximity* to the state description if either the measured level is less than 10% higher than the setpoint and

Algorithm

1. RESET_OPERATORS
2. EXPAND_STATE in *goal_state*
3. For each operator in *operator_buffer*
 - INITIALIZE_POS_CONDITIONS
 - INITIALIZE_NEG_CONDITIONS
 - INITIALIZE_ADD_EFFECTS
 - INITIALIZE_DEL_EFFECTS
 - INITIALIZE_OPERATOR_NAME

This routine initializes the *operator_buffer* structure once the initial and goal states become available. Therefore, it uses *operator_definition*, *present_state* and *goal_state* as inputs and *operator_buffer* as its output. It first copies the contents of *operator_definition* to *operator_buffer* by calling RESET_OPERATORS. It then uses EXPAND_STATE to complete the goal state, if necessary. Finally, each operator is initialized (its name, preconditions and effects).

SEARCH

Inputs

- *Present_state*
- *Goal_state*
- *Terminal_state*
- *Plan_stack*
- *Operator_buffer*

Outputs

- *Plan_stack*

After 30 minutes a sensor failure occurs (sensor signal is cut off). The A/D converter passes on an erroneous (negative) value to the control system. The controller, unaware of the real level in the tank, tries to compensate and fully opens the valve. It does not have enough intelligence to recognize that a negative level is meaningless. A few minutes later, as the tank is about to overflow, the level sensor is reconnected to show the true level.

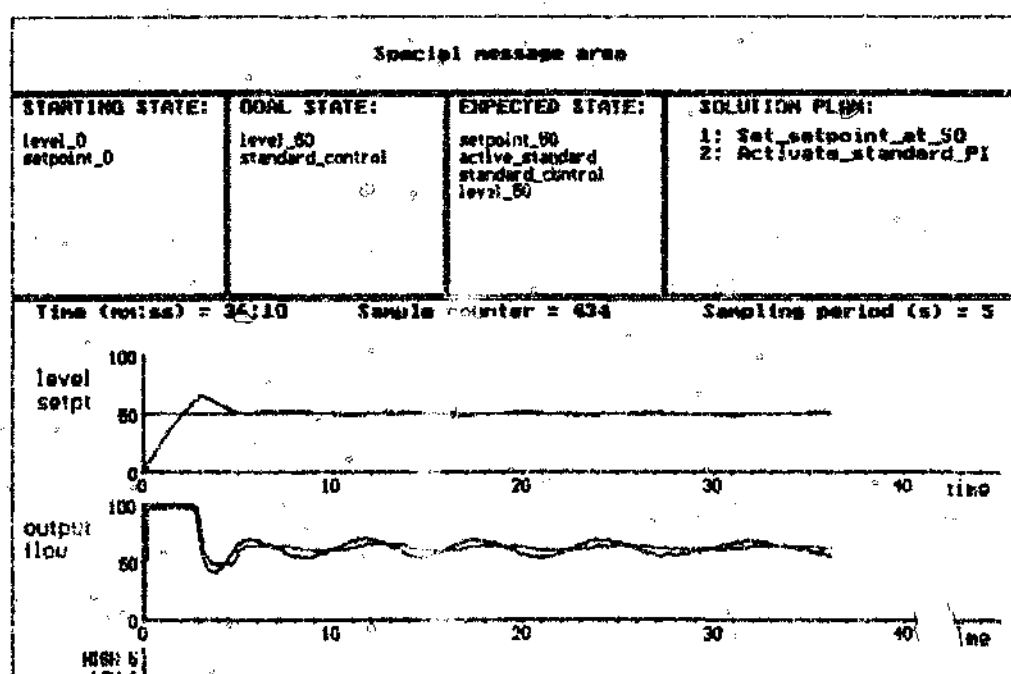


Figure 7.4: High gain PID control.

For comparison, the same test is done on a fixed parameter PID controller with the gain of 5, derivative term of 5 and integral time constant of 100 (see Figure 7.4). It is clear that as the response of the controller is speeded up, unacceptable overshoot and oscillations occur.

7.4.3 Supervisory Control

In this section, our supervisory system is put to the test. The advanced controller it uses, has a standard PID algorithm, but its parameters are now switched. To deal with large disturbances the gain is set at 5, the derivative term at 5 and the integral time constant at 100. Near the setpoint the gain is 1, the integral time constant 200

Algorithm

1. Check *plan_stack*
 - if full then goto 8
 - else continue
2. END_OF_SEARCH for *present_state*, *goal_state*, *terminal_state*
 - if end reached then goto 8
 - else continue
3. Check *plan_stack*
 - if empty then PUSH_ON_STACK *present_state*
4. For each operator in *operator_buffer*
 - CHECK_INHIBITED_OPERATORS
 - if operator is inhibited then goto 4
 - else continue
 - CHECK_OPERATOR
 - if preconditions violated then goto 4
 - if preconditions satisfied then put operator on hold
 - if preconditions satisfied and effects apply then goto 5
5. If operator found then
 - UPDATE_PRESENT_STATE in *present_state*
 - PUSH_ON_STACK *present_state*, *operator*
 - goto 7
6. Else check *plan_stack*
 - if empty then
 - UNINHIBIT_OPERATORS
 - change mode
 - goto 7
 - else continue
 - INHIBIT_OPERATOR
 - POP_OFF_STACK

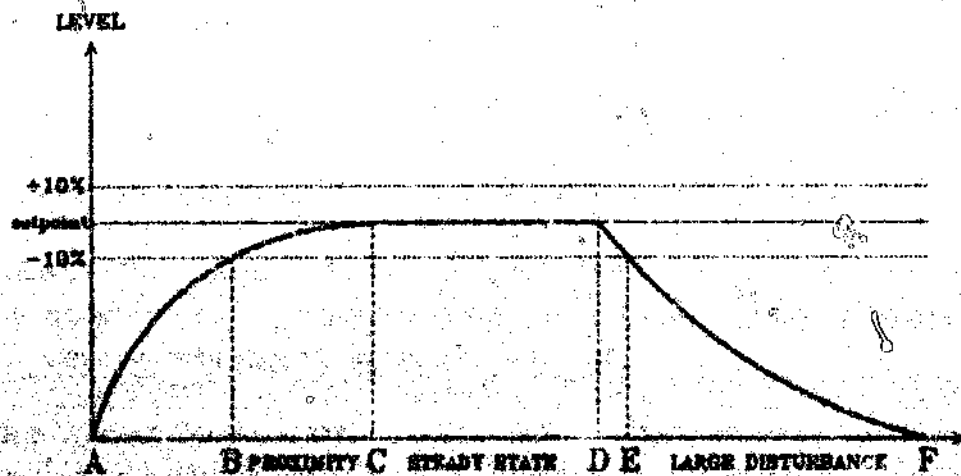


Figure 7.2: Symbolic representation of the level in tank.

dropping or if it is less than 10% lower than the setpoint and increasing. In this case something out of the controller action must start taking place.

C-D Level is at the setpoint and steady, i.e., its derivative is zero. This is the condition which the supervisor tries to maintain at all times. The term *steady state* is added to the state.

E-F Level is beyond the proximity zone and getting further away from the setpoint. If the measured level is more than 10% higher than the setpoint and not decreasing or if it is more than 10% lower than the setpoint and not increasing (the case shown), then the monitoring fraction adds *large disturbance* to the state description. This signifies a serious condition which should prompt the system to respond with urgency.

These three qualitative descriptions show the power of symbolic representation. Each is a single expression containing multifaceted information about both the level and the changes in level, relative to the current setpoint. These three expressions are mutually exclusive, but not exhaustive, as can be seen in Figure 7.2.

7.3.2 Supervisory Actions

An action is stored in the form of its own abstract model and a procedural attachment. The abstract model is called the operator and consists of pos conditions, neg conditions, add effects and del effects. It is used in the planning process only and is

and the derivative term is off.

It must be explained here that, according to the control theory, the integral term should not play a major role in initial disturbance rejection. This task should be left to a high gain and the derivative action which helps to stabilize the system. The high value of gain should produce almost a bang-bang (on-off) action to quickly get rid of major disturbances.

However, it was found during the tests, that the incremental algorithm used in the digital controllers does not tolerate high gains. When set in a high gain proportional or proportional plus derivative mode, it results in erroneous output. It was found that the best improvement in the response can be achieved by speeding up the integral action.

Normal Operating Conditions

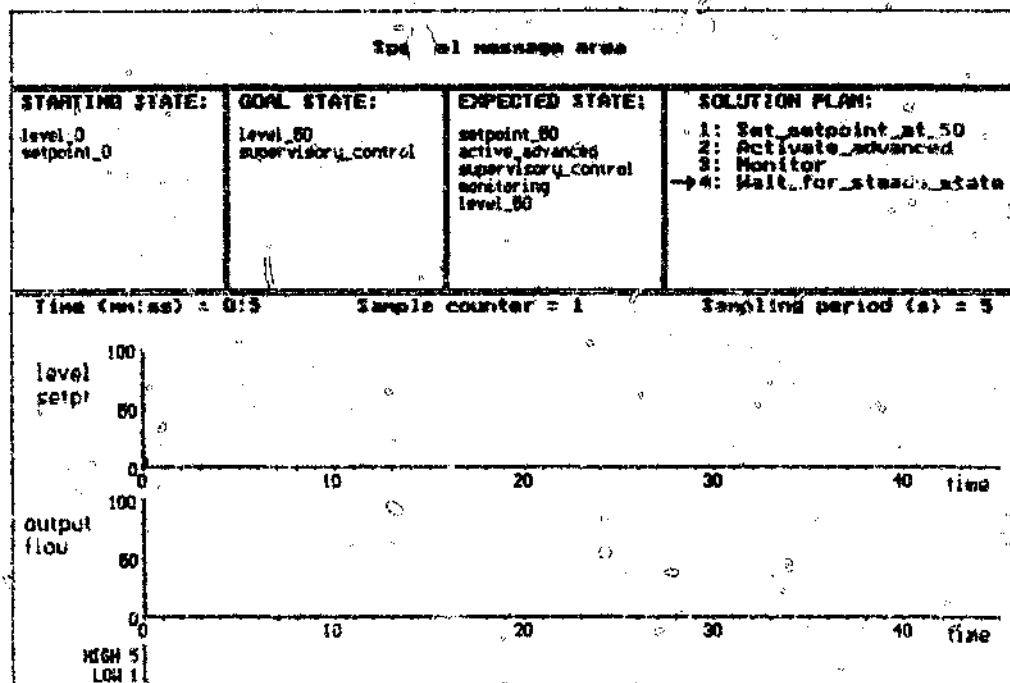


Figure 7.5: Supervisory control: initial plan.

Figure 7.5 shows the first solution plan, just after the system is initialized in the same manner as before. The goal state is now entered as:

7. SEARCH

8. Return

This is an extended version of the recursive algorithm given in Chapter 4. The extensions make it far more suited to the real-time supervisory application, while taking nothing away from its general-purpose character.

Besides the present and goal states, the *terminal_state* structure is passed on to the SEARCH routine. The *terminal_state* structure must be defined in advance, depending on the problem and it is an alternative or *undesired* goal. As the search can revert to a simple data-driven approach in unfavourable conditions, it may come across the terminal state which is the signal that it should stop. The terminal state implies that the condition of the system will not allow it to reach any other goal.

The other extension is a two-mode operation of the SEARCH routine. It starts off working in the normal mode, where it tries to find a solution plan that will satisfy all the terms of a given goal state. The user can assign higher priority to certain terms of the goal state. Then, if search in the normal mode fails, the entire process is repeated in the second mode. Now the planner will try to find a solution plan that satisfies only the high priority terms of the goal. This extension makes the algorithm even more general.

The SEARCH routine only executes if the *plan_stack* is not full and if END_OF_SEARCH indicates that the end has not yet been reached. Then, if the *plan_stack* is empty, the *present_state* (with the initial state) is pushed on the stack using PUSH_ON_STACK. The search begins and each operator in the initialized *operator_buffer* is first checked if it has not been tried before and failed. The CHECK_INHIBITED_OPERATORS routine is used for this. If it has not, CHECK_OPERATOR is called to choose the most applicable operator for the given present and goal states.

Any operator whose preconditions are violated in the present state, is rejected straight away. If an operator is chosen, it will be applied to the present state by UPDATE_PRESENT_STATE. This new intermediate state is still stored in the *present_state* structure and is put on top of the stack using PUSH_ON_STACK. Finally, SEARCH is called.

Alternatively, if no operator qualifies, backtracking is attempted. If *plan_stack* is empty, no further backtracking can take place. This is when a fresh start occurs, by calling UNINHIBIT_OPERATORS and switching to the second mode (for high priority

formulated in such a way that it represents the major characteristics of the actual action it is based on.

The procedural attachment is the action itself, coded as a ready-to-run procedure. This procedure is executed to cause some change in the system, whether it changes something within the supervisor or the external part of the system (plant).

There are fifteen actions in this case. Different plants and control systems will need different actions. Each action is described by giving the full definition of its operator and then examining what the action procedure does.

There is one difference in the way that actions and tasks are interpreted here, as compared with Chapter 5 and Chapter 6. Reading, writing, monitoring and planning were treated before as tasks only. Tasks do not have corresponding operators, so that the planner cannot reason about them (they are always active). Actions are also tasks (i.e., they run in the multitasking kernel just like ordinary tasks) but in addition they can be enabled or disabled at the right time by the supervisor (according to the solution plan).

However, we will now treat monitoring as an action rather than a simple task, so that it does not always have to run. The supervisor needs monitoring to perform its functions; indeed, without monitoring there could be no supervision. The reason behind doing this is that we want to be able to compare the performance of the supervisory system with that of a simple feedback control system. Therefore, we can request the supervisor to disable its monitoring, switch on a standard PI controller and by doing so, disable itself as well.

Set_setpoint

PRE CONDITIONS:	Set_setpoint_at_goal	ADD EFFECTS:
NEG CONDITIONS:		DEL EFFECTS:
setpoint_goal		setpoint_goal
fatal_failure		setpoint_pres

This action adjusts the setpoint of the currently active controller from the present value (pres) to the goal value (goal). The presence of a variable in this operator's name, preconditions and effects means that it has to be initialized before it can be used. The variable pres is initialized with the actual setpoint at the initial state.

- o level.50
- o supervisory_control

The graphs are empty because only one sample has been taken so far. The most obvious and important difference in the plan is that monitoring is now performed. As the controller is always initialized with low gain and slow integral, the initial response is very slow. Consequently, the monitoring action almost immediately signals a large disturbance. In response, after two samples, the planner comes up with a new solution plan which can be seen in Figure 7.6.

The supervisor makes full use of the advanced controller to get to the setpoint more quickly. It turns up the gain, decreases the integral time constant and switches on the derivative term initially, and then reverses the settings when it gets close to the setpoint. Thus, it gets within 10% of the setpoint in 2 minutes now, and reaches the setpoint in less than 3 minutes. This response should be compared with the responses of the fixed-parameter controllers in Figure 7.3 and Figure 7.4.

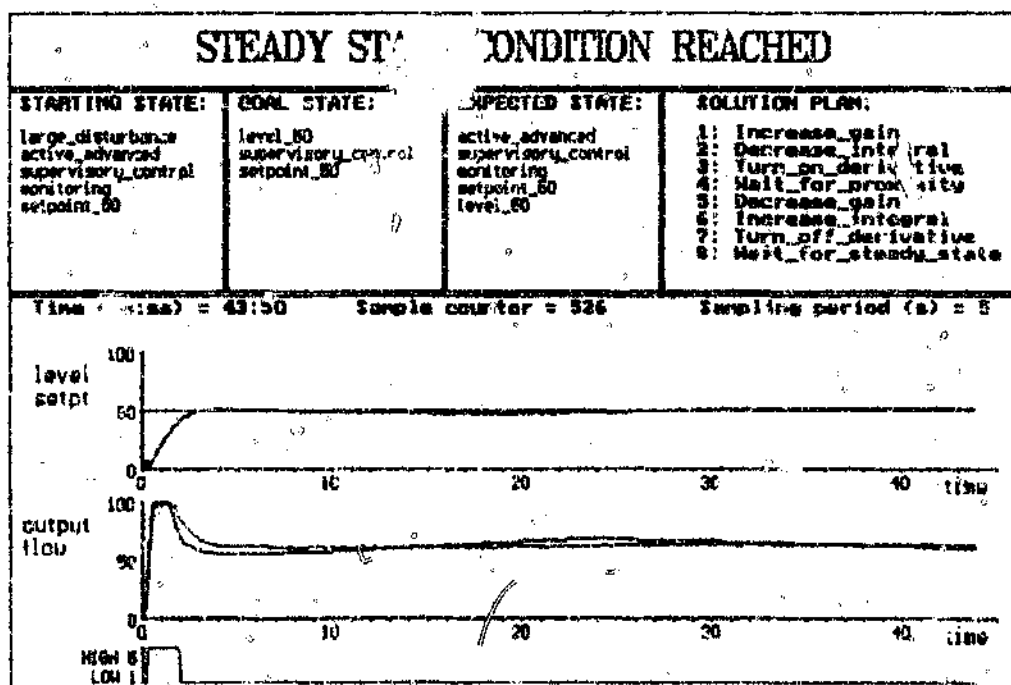


Figure 7.6: Supervisory control: second plan.

The variable `goal` is initialized with the value of the level which is entered by the user as a part of the desired goal state.

This differentiation between the desired level and the setpoint may seem confusing and needs explanation. Thinking in terms of a simple feedback controller, the desired level is the setpoint. However, for a complex supervisory system the two may be totally different. Such a system can use the setpoint to the low-level controller in an arbitrary way, to bring about the required effect, i.e., desired level. Adjusting controller setpoints is typically one of the main methods of exercising supervisory control.

For the level control plant with only one control loop and one setpoint, there is little point in varying the setpoint during the operation. Therefore, the setpoint is initialized to the desired (goal) level straight away. But if it were advantageous, it could be initialized to any other value (calculated by a special algorithm). The deliberately explicit treatment of the controller setpoint helps to illustrate this point.

This action has to be executed before any controller can be activated, unless a fatal failure is reported.

Activate_standard_PI

Pos CONDITIONS: setpoint_goal	Activate_standard_PI	ADD EFFECTS: active_standard standard_control level_goal
NEG CONDITIONS: active_* fatal_failure alarm_condition		DEL EFFECTS: active_advanced active_backup supervisory_control level_pres

The asterisk in the active_* condition stands for any character string (like in DOS operating system).

This action is not a part of the supervisor. It is only included here for illustration purposes, so that non-supervisory PI control can be implemented. It will provide a yardstick for measuring the supervisor's performance. This function activates a direct digital PI controller with fixed parameters; it does not use monitoring. It

goal terms). But if the *plan_stack* is not empty, *INHIBIT_OPERATOR* is called to mark the operator which has produced the present state. *POP_OFF_STACK* is then used to copy the preceding state into the *present_state* structure. Finally, *SEARCH* is called as before.

END_OF_SEARCH

This routine determines whether the search should be terminated. It checks if the *present_state* contains the *goal_state* or the *terminal_state*. If not, it returns zero. If the *SEARCH* routine is running in its second mode, it checks if the *present_state* contains the high priority terms from the *goal_state*. It uses the *COMPARE_STATES* routine.

PUSH_ON_STACK

This routine pushes the given item on the current *plan_stack*. It puts both the contents of the *present_state* and the current operator, on top of the stack. It uses the *COPY_STATES* routine.

CHECK_INHIBITED_OPERATORS

Given an operator and the current *plan_stack*, this routine checks if the operator has been marked as inhibited. If not, it returns zero. It uses *COMPARE_STATES*.

CHECK_OPERATOR

The *CHECK_OPERATOR* routine needs an operator and uses the *present_state* and the *goal_state* structures. It calls four routines in a sequence: *CHECK_NEG_CONDITIONS*, *CHECK_POS_CONDITIONS*, *CHECK_DEL_EFFECTS* and *CHECK_ADD_EFFECTS*. These are used to check if the preconditions of the given operator are satisfied in the present state and whether the effects agree with the goal state. Depending on the combination, it returns an integer code (rating) of the operator.

Overload

After 10 minutes an overload occurs (outlet valve is opened to 120%). Again, re-planning occurs in response, as the monitoring action diagnoses an overload. This is illustrated in Figure 7.7, together with the alarm message. After another minute, the outlet valve is reset to its previous position and the tank starts filling again.

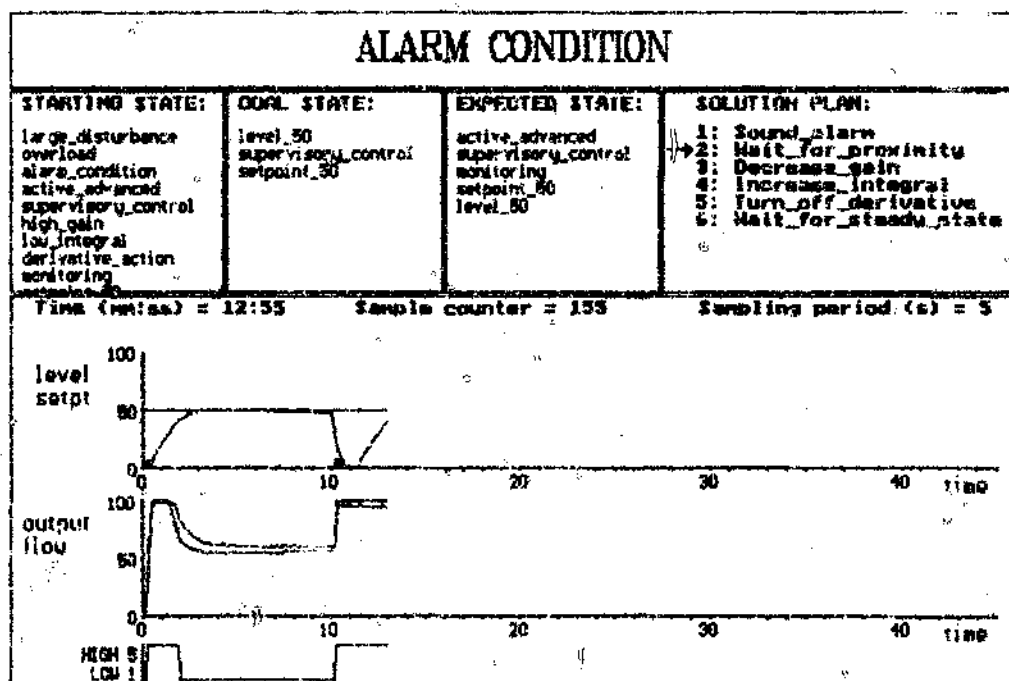


Figure 7.7: Supervisory control: overload.

Valve Failure

Figure 7.8 shows the response of the supervisory system to a failure in the valve. This occurs after 2 minutes (when the pressurized air supply is cut off) and is immediately diagnosed by the monitor. As it is the only actuator in the plant, there is actually nothing that the supervisor can do, except to alert the operator and continue. The air supply is reconnected before the next test.

UPDATE_PRESENT_STATE

The UPDATE_PRESENT_STATE routine first calls DEL_FROM_STATE which deletes all the del effects of a given operator from the *present_state* structure. It then uses ADD_TO_STATE, to include all the add effects of the operator in *present_state*. In effect, it applies the operator to the present state, to create a new state.

UNINHIBIT_OPERATORS

This routine clears all the operators that have previously been marked as inhibited. This is done to allow a second search to start anew. This second search is only tried if the initial search is unsuccessful. It has a better chance of succeeding because it only considers high priority goal terms.

INHIBIT_OPERATOR

This routine is used in backtracking to mark a given operator as inhibited. This ensures that the same operator will not be tried again from this node of the search tree.

POP_OFF_STACK

This routine is the reverse of PUSH_ON_STACK. It is the mechanism for backtracking. The top of the *plan_stack* is cleared and the preceding state is copied into the *present_state* structure and becomes the new top of the stack. The COPY_STATES routine is used in the process.

6.2.6 Task Initialization Routines

The four routines described below, are used to activate the basic multitasking kernel. Refer to Appendix D for the code.

can only be executed once the setpoint has been adjusted, but not if any other controller is already active. Once switched on, it carries on until the system is forcefully stopped; the supervisor relinquishes all its control over the plant to it.

Active advanced

Main Controller: advanced active	Active advanced	And Errors: active advanced supervisory control Det. Error active backup
--	-----------------	--

This action activates the backup controller, but only if a fault is detected in the main controller. If the controllers were implemented in hardware, then the backup controller would most probably be a simple one, in the interest of lower costs. In this case, it costs nothing extra to have two high performance, advanced controllers.

Active backup

Main Controller: advanced failure active	Active backup	And Errors: active backup supervisory control Det. Error active advanced
--	---------------	--

This action activates the backup controller, but only if a fault is detected in the main controller. If the controllers were implemented in hardware, then the backup controller would most probably be a simple one, in the interest of lower costs. In this case, it costs nothing extra to have two high performance, advanced controllers.

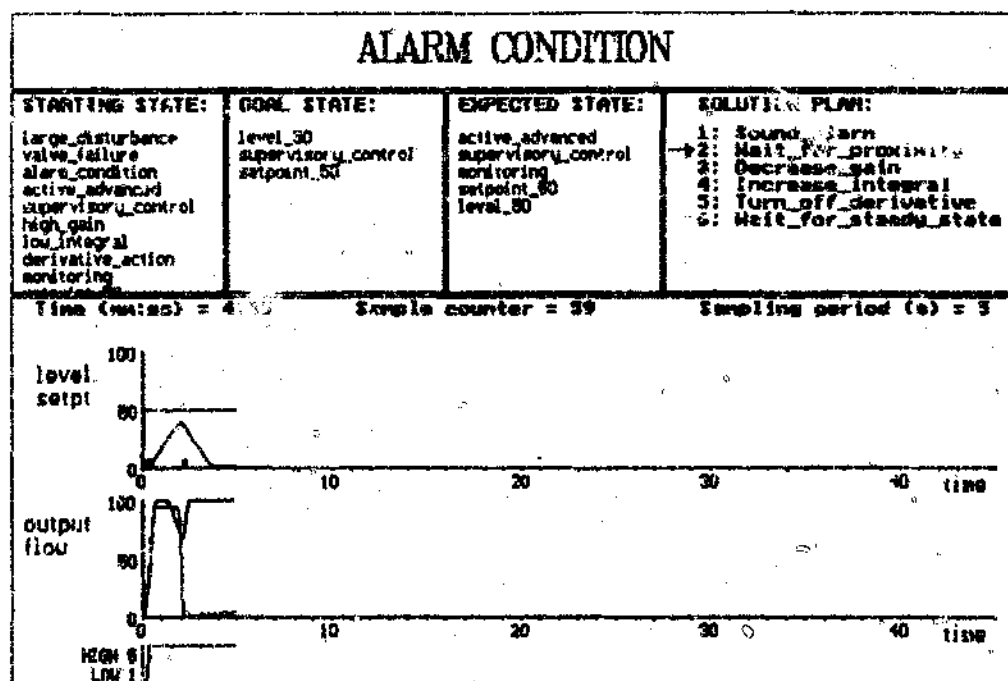


Figure 7.8: Supervisory control: valve failure.

Controller Failure

After 10 minutes a controller failure is simulated. The new solution plan tells the system to switch to backup control and to alert the operator. Therefore, despite the failure, the system carries on with very little negative effect. This allows it to maximize the production rate. The solution plan is illustrated in Figure 7.9.

Switching to another controller has a side-effect of introducing a large disturbance into the system. Therefore, another replanning follows, which puts the backup controller in the high gain mode.

Sensor Failure

Finally, after 35 minutes, when the system has recovered again, a sensor failure occurs (sensor signal is cut off). The supervisory system detects that fault and Figure 7.10 shows its reaction.

MAIN

Algorithm

1. INITIALIZE all tasks
2. SYSTEM_GO to first task control block

This is the top-level (compilable) routine and therefore has to be called MAIN in the C language. It has no proper inputs or outputs, although it does use information in the form of a preentered list of tasks, to dynamically allocate an area in memory in which they can run concurrently. Functionally it is very simple, providing only the first step of the run-time activity.

It first calls the INITIALIZE routine for each task in the preentered list. It then calls the SYSTEM_GO routine with the control block of the first task on the list, thus beginning the multitasking process. Control of the CPU is then passed directly from one task to the next and does not return to MAIN, unless the process is forcefully stopped.

INITIALIZE

Inputs

- o task name

Algorithm

1. Allocate a stack area for the task
2. Allocate a task control block for the task
 - o set the instruction pointer to point to the address of the task's routine
 - o set the stack pointer to point to the stack area
 - o save the task context
3. INSERT_ITEM

Monitor

Pos Conditions: setpoint.goal active.*	Monitor	ADD EFFECTS: monitoring
Neg Conditions: monitoring fatal.failure active.standard		DEL EFFECTS:

The reason for classify this task as an action has already been explained. Monitoring is only used with the supervisory control. It can only be switched on once the advanced (or backup) controller is running, but it then runs periodically for as long as the plant is operating. The range and type of functions performed by this action have been described in the previous section.

Increase gain

Pos Conditions: active.* large.disturbance	Increase gain	ADD EFFECTS: high.gain
Neg Conditions: high.gain fatal.failure active.standard		DEL EFFECTS:

This action allows the supervisor to change the gain of the controller (only advanced or backup) from low to high. To do that, a large disturbance must be reported by the monitor.

The values of low and high gains are preentered constants in our system, but they could also be calculated on-line.

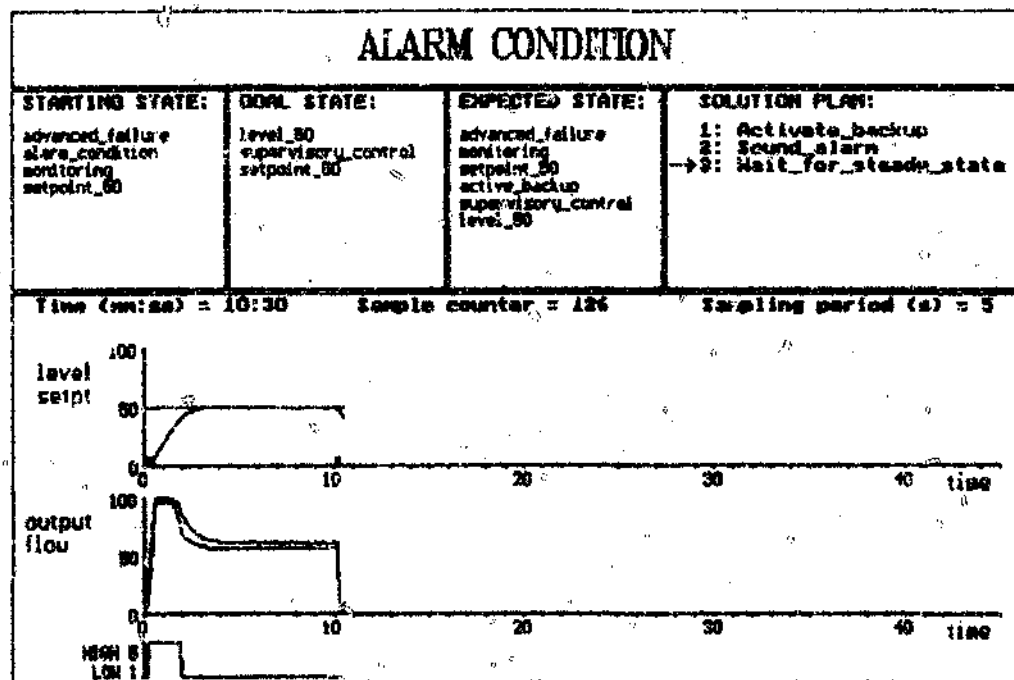


Figure 7.9: Supervisory control: controller failure.

The newly created solution plan takes into account the seriousness of this latest change and tells the system to shut down in order to avoid a safety risk. The sensor is reconnected after another 3 minutes, to verify that the level has indeed dropped.

7.5 Conclusion

The concepts and ideas developed in chapters 1-3, then used to design the supervisory system in chapters 4-5 and encoded as a software program (described in Chapter 6) have been tested in this chapter.

Although a very simple laboratory plant has been used, the real significance of this case study lies in the fact that it is real-world. Unlike simulation-based testing, it explicitly involves issues such as sampling, situation assessment, multitasking and real-time constraints.

The tests have shown that the intelligent supervisory control system exhibits a performance advantage over a conventional feedback controller, under normal op-

This routine initializes a task, given its name. To enable concurrent running of multiple tasks, a separate stack area for each task is required. Thus, INITIALIZE first allocates a new memory area for the given task as its stack.

Similarly, each task needs to have its essential information stored in a separate area in memory. This information, stored in a structure called task control block, includes the instruction pointer (which points to the actual routine of the task), the stack pointer, pointing to the newly allocated stack area, as well as the task context. The latter is saved in the *jmp.buf* data structure and consists of the CPU registers and flags and the stack and base pointers. Finally, the INSERT_ITEM routine is called to include the task control block on the circular list.

INSERT_ITEM

A task control block created and initialized with the INITIALIZE routine, needs to be linked with all the other tasks to form a circular list. This circular list is the basis of non-preemptive round-robin scheduling. INSERT_ITEM takes a given task control block and puts it at the back of the circular list, updating pointers to and from its neighbours.

SYSTEM_GO

SYSTEM_GO is called for the first task control block on the circular list. It performs a nonlocal jump to the routine associated with the given task and effectively transfers control of the CPU to the first task. This activates the multitasking process.

6.3 Conclusion

In this chapter, the software implementation of the intelligent supervisory control system has been presented, without focusing on any specific computer language. The main routines of the program have been described in terms of the flow of data and the internal algorithms.

The complete source code for all of them, and many other routines which play a supportive role (e.g., string handling, input-output, graphics), is given in the appendices.

Decrease_gain

Pos CONDITIONS: active.* high_gain	Decrease_gain	ADD EFFECTS:
NEG CONDITIONS: large_disturbance fatal_failure active_standard		DEL EFFECTS: high_gain

This is the reverse of the previous action. It can only take place once the disturbance has vanished. The controller gain is changed to a lower value to reduce demands on the actuator, by producing a smoother control action.

Increase_integral

Pos CONDITIONS: active.* low_integral	Increase_integral	ADD EFFECTS:
NEG CONDITIONS: large_disturbance fatal_failure active_standard		DEL EFFECTS: low_integral

This action is used to increase the integral time constant from low to high. A higher value of the time constant means that the integral action is slower. Again, it is only applicable for the advanced and backup controllers.

Decrease_integral

Pos CONDITIONS: active.* large_disturbance	Decrease_integral	ADD EFFECTS: low_integral
NEG CONDITIONS: low_integral fatal_failure active_standard		DEL EFFECTS:

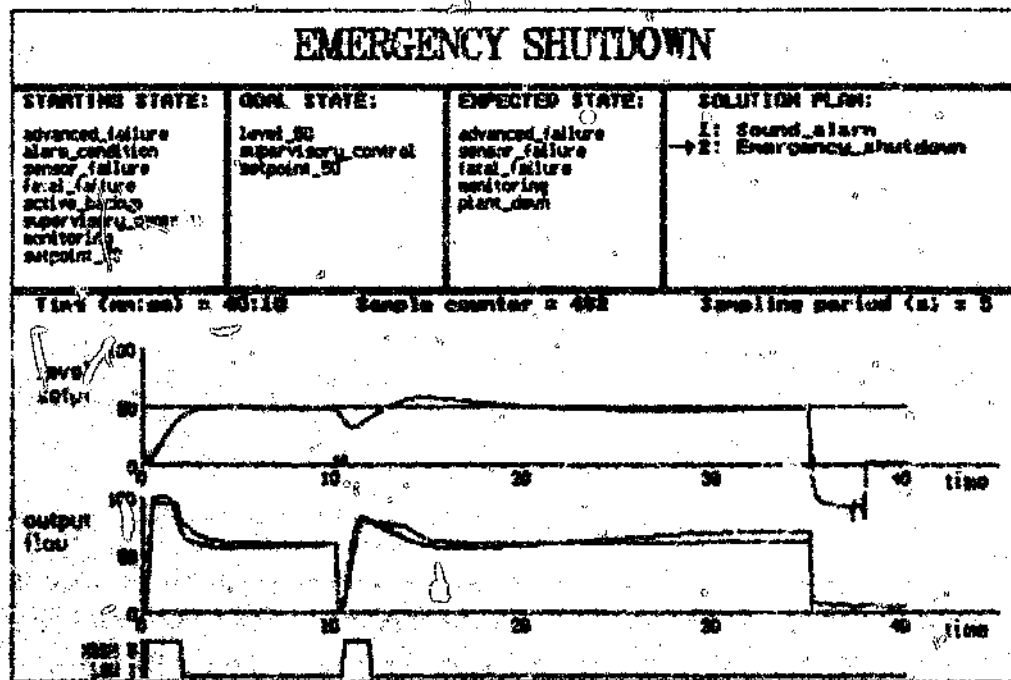


Figure 7.10. Supervisory control: sensor failure.

erating conditions. Even more significantly, the supervisory system is capable of dealing with abnormal operating conditions such as overloads and equipment failures. A low-level controller cannot even recognize such conditions, let alone prevent potentially dangerous situations which could thus arise.

()

The program used in the case study was written in the C language using Turbo C compiler (ver. 2.0). It contains over 5300 lines of code (including header files) in over 70 functions and compiles to a 160Kbyte executable file.

To reverse the effects of the above action, the integral time constant can be decreased. This will result in the response due to the integral term being faster.

Turn_on_derivative

Pos CONDITIONS: active large disturbance	Turn_on_derivative	ADD EFFECTS: derivative action
Neg CONDITIONS: derivative action fatal failure active standard		DEL EFFECTS:

This action can be used by the supervisor to reconfigure the advanced or backup controller, by switching from a PI mode to a PID control algorithm. This is done for large disturbances, to offset the destabilizing effects of a higher gain and a faster integral.

Turn_off_derivative

Pos CONDITIONS: active derivative action	Turn_off_derivative	ADD EFFECTS:
Neg CONDITIONS: large disturbance fatal failure active standard		DEL EFFECTS: derivative action

This action will change back to a PI algorithm, if there is no serious disturbance in the load. It will do so by setting the derivative term of the advanced or backup controller to zero.

Chapter 8

Conclusions

It has been found, during this research, that there is a need for an effective supervisory control system for use in process control and possibly many other applications. It should fill the gap above the existing low-level automatic controllers, to provide a more autonomous, better performing and safer control system. There is a very strong argument that such a system should be intelligent.

This dissertation has shown how the planning techniques developed in Artificial Intelligence can be used to provide that intelligence. The design of a reactive planner has been described and a closed-loop supervisory system built around the planner. The supervisor has been implemented in the C language and then evaluated on an actual laboratory plant. The tests have shown very positive results and have proved the developed theory in practice.

8.1 AI in Control

Better quality of control can be achieved by analysing the controlled plant more thoroughly and then devising more sophisticated control algorithms, on the basis of this analysis. However, there is a limit as to how far this can go. It is theoretically impossible to analyse a dynamic and unpredictable system (such as most controlled plants) to the lowest level of detail. Often, very little is known about the plant and there are many real-world cases which do not lend themselves to mathematical analysis.

Chapter 7

Case Study

It is a common failing of most systems employing AI methods, that they seldom get tested in real-world situations. Though they may be impressive under simulated conditions, their real performance is never properly evaluated. This is probably the main reason why many engineers approach AI-based systems with a high degree of scepticism. The few systems that have found their way to industry are typically off-line and advisory in nature.

Therefore, the development of an AI-based supervisory control system which could be interfaced with a real plant and operate in real-time, on-line mode was, from the very outset, one of the main objectives of this research. This chapter will illustrate how the ideas developed earlier on, can be used to regulate and supervise an existing laboratory plant.

7.1 The Laboratory Plant

The plant used for this experiment can be found in any processing factory. It is a simple, single tank, level control plant. The tank of about 10 litres capacity is supplied from and drained into a single sump filled with water. The supply part consists of a constant-speed pump submerged in the water, a pneumatic diaphragm valve, a flowmeter and an inlet pipe terminated at the top of the tank. The drain part of the circuit starts at the bottom of the tank and includes a manually operated valve and an outlet pipe going back to the sump. The tank is equipped with a level sensor. The plant is shown in Figure 7.1.

Wait_for_proximity

Pos CONDITIONS: active* setpoint_goal monitoring large_disturbance high_gain low_integral derivative_action	Wait_for_proximity	ADD EFFECTS:
NEG CONDITIONS: alarm_condition fatal_failure		DEL EFFECTS: large_disturbance overload valve_failure

This is a passive action which is used by the supervisor to evaluate its own performance in the face of a large disturbance.

The controller must already be set to high gain, low integral with derivative on, before this action can be executed. It ends as soon as the proximity zone is entered, i.e., when the level comes to within 10% of the setpoint and keeps approaching it.

When this action ends, the system is close to its operating point. A change in the control algorithm can then take place to modify the character of the control action from fast and decisive to smooth and accurate.

Waiting for the system to reach its proximity zone may take some time and therefore this action will not execute if there is an alarm condition. Raising an alarm has higher priority and cannot afford to wait.

Chapter 8

Conclusions

It has been found, during this research, that there is a need for an effective supervisory control system for use in process control and possibly many other applications. It should fill the gap above the existing low-level automatic controllers, to provide a more autonomous, better performing and safer control system. There is a very strong argument that such a system should be intelligent.

This dissertation has shown how the planning techniques developed in Artificial Intelligence can be used to provide that intelligence. The design of a reactive planner has been described and a closed-loop supervisory system built around the planner. The supervisor has been implemented in the C language and then evaluated on an actual laboratory plant. The tests have shown very positive results and have proved the developed theory in practice.

8.1 AI in Control

Better quality of control can be achieved by analysing the controlled plant more thoroughly and then devising more sophisticated control algorithms, on the basis of this analysis. However, there is a limit as to how far this can go. It is theoretically impossible to analyse a dynamic and unpredictable system (such as most controlled plants) to the lowest level of detail. Often, very little is known about the plant and there are many real-world cases which do not lend themselves to mathematical analysis.

The proposed alternative is to give the control system more intelligence and freedom to act, without discarding the usefulness of lower level control algorithms. This gives rise to the concept of intelligent supervisory control. The intelligent part supervises automatic controllers (unlike direct AI-based control), although it can take over direct control if necessary. Planning is chosen over the other AI techniques (particularly rule-based expert systems) because of its clear separation between domain-dependent knowledge base and general-purpose inference. It is also seen as the more suitable tool for synthesis (whereas most expert systems are analytical).

8.2 Approach to Planning

An easy-to-follow version of knowledge representation is proposed, in keeping with the idea that a supervisory system should not require AI specialists to understand

A difference/data driven search is used (based on enhanced forward chaining), as it is more applicable to creative tasks such as planning. The most important feature of the proposed planner is its reactivity which allows its interfacing with the real-world.

Two characteristics which the proposed planner does not have, but which are considered very useful, are hierarchical planning and learning ability. When people plan, associative memory ensures that only the applicable actions (a very small subset of all possible actions) are considered. Out of these, the actions involving much detail are not considered until a later stage. Learning implies a continuous improvement in performance, also characteristic of humans.

8.3 Software Implementation

The entire system, including the AI part, was implemented in the C language. This approach has shown a disadvantage: C is not an AI language (the best known AI language is Lisp) and therefore some extra effort was involved in creating a library of string handling routines (needed for pattern matching). On the other hand, C is a powerful and fast language which allows easy interfacing with hardware modules, and is often used for real-time applications. It is for these reasons that C was chosen.

Consequently, the whole system, including digital controllers, A/D card drivers and the scheduler, could be implemented in one language. Although non-preemptive

scheduling was successfully used, it has been found that a preemptive scheduler would generally be more suitable. Ultimately, parallel processing should be aimed at.

The resulting multitasking system has proved to be fast enough for real-time applications, a requirement which is often the main stumbling block for AI-based systems used in on-line control. The efficiency of this system is underlined by the fact that the tests were performed on a single IBM-AT compatible computer, with a 12MHz 286 processor. This class of computers is hardly the ideal tool when concurrent programming and speed are critical. There are already personal computers up to ten times as fast and more expensive workstations even faster.

8.4 Experimental Results

Within the limitations of the simple laboratory plant used for testing, the case study has validated the theoretical concepts developed in this research. A model input, multi-output, variable gain, fault tolerant, reconfigurable controller was effectively implemented with the supervisor. The system has shown its advantages under both normal and abnormal operating conditions. An important strength of the proposed supervisor is its ability to recognise when the full goal cannot be reached, but still achieving valid results. The most significant improvement over a conventional feedback controller has been shown in reacting to structural failures.

8.5 Future Scenarios

There is a definite trend toward more intelligent systems, particularly in the control domain. Whether the current AI techniques can be called intelligent is debatable, but they are, without doubt, more intelligent than conventional computing methods. However, AI is registering a steady, if slow, progress. New, better techniques in knowledge representation and inference can be expected and when combined with new generations of symbolic, parallel-processing computers, the potentials are enormous.

Appendix A

Data definitions

This appendix lists the files defining the most important global data in the form of macros, structures and variables.

A.1 Sup-def.c

```
/*
-----
filename:    SUP-DEFS.C

description: Definition of global data and data structures used by the
            supervisor.
-----
*/

/*
----- Global constants -----
*/

#define ON      1
#define OFF    0
#define TRUE   1
#define FALSE  0
#define LOW    1
#define HIGH   2

/*
----- Constants used in data structures definitions -----
*/

#define STACKSIZE 20    /* max no of items stored on stack */
```

```

#define LETTERS 22 /* max no of letters in char string */
#define TERMS 12 /* max no of propositions in states */
#define INH_OPERATORS 12 /* max no of inhibited operators */

```

```

#define USER_TERMS 10 /* max no of state terms entered by the user */
#define END 'O/C' /* string used to end states (character 'O') */

```

```

/*
----- Global data structures -----
*/

```

```

struct symbolic_data
{

```

```

    int active; /* 1 */
    int gain; /* 2 */
    int integral; /* 3 */
    int derivative; /* 4 */
    int large_disturbance; /* 5 */
    int overload; /* 6 */
    int valve_failure; /* 7 */
    int advanced_failure; /* 8 */
    int backup_failure; /* 9 */
    int alarm_condition; /* 10 */
    int sensor_failure; /* 11 */
    int fatal_failure; /* 12 */
    int steady_state; /* 13 */
    int proximity; /* 14 */

```

```

};

struct symbolic_data state_of_plant =

```

```

{ 0, LOW, HIGH, LOW, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
/*
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14.

```

Structure state_of_plant of type symbolic_data defined above is used to store the qualitative knowledge about the plant and the controllers. This knowledge is used in the planning process.

```

*/

```

```

struct numeric_data
{

```

```

    int setpoint;
    double level;
    double flowrate;
    double level_derivative;
    double controller_output;
    double last_sample_time;

```

```

};

```

```
struct numeric_data plant_variables =
```

```
{ 0, 0.0, 0.0, 0.0, 0.0, 0.0 };
```

```
/*
```

Structure plant_variables of type numeric_data defined above is used to store the numerical (quantitative) information about the plant and the control system.

```
*/
```

```
struct flag_table
```

```
{
```

```
int plan_flag;
int monitor_flag;
int new_read_flag;
int new_write_flag;
int new_data_flag;
int new_diagnosis_flag;
```

```
};
```

```
struct flag_table task_flags =
```

```
{ 0, OFF, OFF, OFF, OFF, OFF };
```

```
/*
```

Structure task_flags of type flag_table defined above is used in reconfiguring active tasks in the multitasking kernel.

```
*/
```

```
struct search_space_node
```

```
{
```

```
char operator_name[LETTERS];
char intermediate_state[TERMS][LETTERS];
char inhibited_operators[INH_OPERATORS][LETTERS];
```

```
};
```

```
struct search_space_node plan_stack[STACKSIZE];
```

```
/*
```

Structure plan_stack of type search_space_node defined above is used during the planning process to store the intermediate states and at the end of planning it contains the solution plan.

```
*/
```

----- Constants used in control -----

*/

```
#define STANDARD_CONTROL 1
#define ADVANCED_CONTROL 2
#define BACKUP_CONTROL 3
```

```
#define STANDARD_GAIN 2 /* or 5 */
#define STANDARD_INTEGRAL 300 /* or 100 */
#define STANDARD_DERIVATIVE 0 /* or 5 */
```

```
#define LOW_GAIN 1
#define HIGH_GAIN 5
#define LOW_INTEGRAL 100
#define HIGH_INTEGRAL 300
#define SMALL_DERIVATIVE 0
#define LARGE_DERIVATIVE 5
```

----- Other constants -----

```
#define SAMPLING_PERIOD 4
#define TIME_MARGIN 1
#define CLOCK_RESOLUTION 0.001
```

----- Constants used for the user interface and graphics -----

```
#define TEXT_FONT 0
#define LABEL_FONT 2
#define SPECIAL_FONT 1
```

```
#define TEXT_SIZE 1
#define LABEL_SIZE 5
#define SCALE_SIZE 4
#define SPECIAL_SIZE 3
```

```
#define TEXT_CHAR_WIDTH 8
#define TEXT_CHAR_HEIGHT 9
#define LABEL_CHAR_WIDTH 8
#define LABEL_CHAR_HEIGHT 10
#define SCALE_CHAR_WIDTH 8
#define SCALE_CHAR_HEIGHT 8
```

```
#define TEXT_COLOUR WHITE
#define TEXT_BKG BLACK
#define GRAPE_COLOUR WHITE
#define GRAPE_BKG BLACK
#define PLOT1_COLOUR WHITE
#define PLOT2_COLOUR LIGHTMAGENTA
```

```
#define PLOTS_COLOUR WHITE
#define SPECIAL_COLOUR BLACK
#define ARROW_COLOUR WHITE
```

```
#define X 0
#define Y 130
#define X1 270
#define X2 414
#define X3 630
#define Y1 0
#define Y2 41
#define Y3 147
#define Y4 340
```

```
#define GRAPH1_X 84
#define GRAPH1_Y 730
#define GRAPH2_X 84
#define GRAPH2_Y 213
#define GRAPH3_X 84
#define GRAPH3_Y 340
```

```
/*
----- Global variable definition and initialization -----
*/
```

```
int stack_top = 0; /* used to store current stack size */
int next_action = 1; /* indicates next action to be executed */
int task_counter = 1; /* counts total no of tasks executed */
int sample_counter = 0; /* counts total no of samples taken */
```

A.2 Plan-def.c

```
/*
```

```
filename: PLAN-DEF.C
```

```
description: Definition of problem independent data and data
structures used only by the planner.
```

```
*/
```

```
#define NB '!' /* first char of high priority goal terms */
#define RECORDSIZE 1 /* max size of record for tracing search */
```

```
/*
```

```
----- Data structures -----
```

```
*/
```

```
struct operator
```

```
{  
    char name[LETTERS];  
    char pos_condition[TERMS][LETTERS];  
    char neg_condition[TERMS][LETTERS];  
    char add_effect[TERMS][LETTERS];  
    char del_effect[TERMS][LETTERS];  
};
```

```
/*  
    Defines structure type operator used later in the declaration  
    of the operator_definition and operator buffer structures.  
*/
```

```
struct entry
```

```
{  
    char operator_name[LETTERS];  
    int operator_rating;  
};
```

```
struct entry record[RECORDSIZE];
```

```
/*  
    Structure record of type entry defined above can be optionally used  
    for tracing the search procedure by storing every single step of  
    the search (including backtracking).  
*/
```

```
/*  
    ----- Global variable initialization -----  
*/
```

```
int second_mode_flag = OFF; /* used to change the mode of search */  
int current_bottom   = 1;   /* used only for multiple goal missions */  
int record_top       = 0;   /* stores current size of record */
```

A.3 Pl-co-df.c

```
/*
```

```
-----  
filename:    PL-CO-DF.C
```

```
description: Definition of operators and other data specific to the  
             control problem (case study).  
-----
```

```
*/
```

```
#define OPERATORS 16
#define MIN 3
```

```
/* max no of operators for the problem */
/* min no of chars needed for matching */
```

```
struct operator operator_definition[OPERATORS] =
```

```
{
  { "Activate_standard_PI",
    { "setpoint_goal", "0" },
    { "active", "fatal_failure", "alarm_condition", "0" },
    { "active_standard", "standard_control", "level_goal", "0" },
    { "active_advanced", "active_backup", "supervisory_control",
      "level_pres", "0" }
  },
  { "Activate_advanced",
    { "setpoint_goal", "0" },
    { "advanced_failure", "active", "fatal_failure", "0" },
    { "active_advanced", "supervisory_control", "0" },
    { "active_backup", "0" }
  },
  { "Activate_backup",
    { "setpoint_goal", "advanced_failure", "0" },
    { "backup_failure", "active", "fatal_failure", "0" },
    { "active_backup", "supervisory_control", "0" },
    { "active_advanced", "0" }
  },
  { "Set_setpoint_at_goal",
    { "0" },
    { "setpoint_goal", "fatal_failure", "0" },
    { "setpoint_goal", "0" },
    { "setpoint_pres", "0" }
  },
  { "Wait_for_steady_state",
    { "setpoint_goal", "active", "monitoring", "0" },
    { "level_goal", "high_gain", "large_disturbance", "fatal_failure",
      "alarm_condition", "low_integral", "derivative_action", "0" },
    { "level_goal", "0" },
    { "level_pres", "valve_failure", "overload", "0" }
  },
  { "Monitor",
    { "setpoint_goal", "active", "0" },
    { "monitoring", "fatal_failure", "active_standard", "0" },
    { "monitoring", "0" },
    { "0" }
  },
  { "Increase_gain",
    { "active", "large_disturbance", "0" },
    { "high_gain", "fatal_failure", "active_standard", "0" },
    { "high_gain", "0" },
    { "0" }
  }
}
```

```

{ "Decrease_gain",
  { "active_+", "high_gain", "0" },
  { "large_disturbance", "fatal_failure", "active_standard", "0" },
  { "0" },
  { "high_gain", "0" }
},
{ "Increase_integral",
  { "active_+", "low_integral", "0" },
  { "large_disturbance", "fatal_failure", "active_standard", "0" },
  { "0" },
  { "low_integral", "0" }
},
{ "Decrease_integral",
  { "active_+", "large_disturbance", "0" },
  { "low_integral", "fatal_failure", "active_standard", "0" },
  { "low_integral", "0" },
  { "0" }
},
{ "Turn_on_derivative",
  { "active_+", "large_disturbance", "0" },
  { "derivative_action", "fatal_failure", "active_standard", "0" },
  { "derivative_action", "0" },
  { "0" }
},
{ "Turn_off_derivative",
  { "active_+", "derivative_action", "0" },
  { "large_disturbance", "fatal_failure", "active_standard", "0" },
  { "0" },
  { "derivative_action", "0" }
},
{ "Wait_for_proximity",
  { "setpoint_goal", "active_+", "high_gain", "large_disturbance",
    "monitoring", "derivative_action", "low_integral", "0" },
  { "fatal_failure", "alarm_condition", "0" },
  { "0" },
  { "large_disturbance", "overload", "valve_failure", "0" }
},
{ "Sound_alarm",
  { "alarm_condition", "0" },
  { "0" },
  { "0" },
  { "alarm_condition", "0" }
},
{ "Emergency_shutdown",
  { "fatal_failure", "0" },
  { "plant_down", "alarm_condition", "active_standard", "0" },
  { "plant_down", "0" },
  { "active_+", "setpoint_+", "high_gain", "low_integral",
    "derivative_action", "large_disturbance", "supervisory_control",
    "0" }
},
{ "0"
}

```

};

/*

Structure operator_definition of type operator is used to define abstract representations of the actions that the supervisor can take. These operators are initialized and used in the planning process.

*/

struct operator operator_buffer[OPERATORS];

/*

Structure operator_buffer of type operator is used to store an initialized copy of the above operators.

*/

Appendix B

Supervisory tasks and actions

This appendix contains the source code for the supervisory tasks and actions, as well as the lower level actions used in their implementation.

B.1 Sup-acts.c

/*

filename: SUP-ACTS.C

description: Supervisory tasks and actions corresponding to each operator are defined in this file (except the planning task).

written by: A. Korpala - 3/91

*/

```
#include <bios.h>
#include <math.h>
#include <stdlib.h>
#include <process.h>
```

```
#include "sup-ifac.c"
#include "sup-fcns.c"
#include "sup-adac.c"
```

```
void increase_gain(void)
```

```
/*
```

```
    This action increases the gain of the currently active
    controller by updating the shared state_of_plant structure.
```

```
*/
```

```
{
```

```
    for (;;)
    {
```

```
        if ( compare_terms(plan_stack[next_action].operator_name,
                           "Increase_gain") == TRUE &&
            time_up("Increase_gain") == FALSE )
        {
```

```
            state_of_plant.gain = HIGH;
            next_action++;
            show_next_action();
        }
```

```
    }
    yield();
}
```

```
void decrease_gain(void)
```

```
/*
```

```
    This action decreases the gain of the currently active
    controller by updating the shared state_of_plant structure.
```

```
*/
```

```
{
```

```
    for (;;)
    {
```

```
        if ( compare_terms(plan_stack[next_action].operator_name,
                           "Decrease_gain") == TRUE &&
            time_up("Decrease_gain") == FALSE )
        {
```

```
            state_of_plant.gain = LOW;
            next_action++;
            show_next_action();
        }
```

```
    }
    yield();
}
```

```
void increase_integral(void)
```

```
/*
```

```
    This action increases the integral term of the currently active
    controller by updating the shared state_of_plant structure.
```

```
*/
```

```
{
```

```
    for (;;)
    {
```

```
        if ( compare_terms(plan_stack[next_action].operator_name,
```

```

        "Increase_integral") == TRUE &&
        time_up("Increase_integral") == FALSE )
    {
        state_of_plant.integral = HIGH;
        next_action++;
        show_next_action();
    }
    yield();
}

```

```

void decrease_integral(void)

```

```

/*
   This action decreases the integral term of the currently active
   controller by updating the shared state_of_plant structure.
*/
{
    for (;;)
    {
        if ( compare_terms(plan_stack[next_action].operator_name,
                           "Decrease_integral") == TRUE &&
            time_up("Decrease_integral") == FALSE )
        {
            state_of_plant.integral = LOW;
            next_action++;
            show_next_action();
        }
        yield();
    }
}

```

```

void turn_on_derivative(void)

```

```

/*
   This action turns on the derivative term of the currently active
   controller by updating the shared state_of_plant structure.
*/
{
    for (;;)
    {
        if ( compare_terms(plan_stack[next_action].operator_name,
                           "Turn_on_derivative") == TRUE &&
            time_up("Turn_on_derivative") == FALSE )
        {
            state_of_plant.derivative = HIGH;
            next_action++;
            show_next_action();
        }
        yield();
    }
}

```

```
void turn_off_derivative(void)
```

```
/*
```

```
This action turns off the derivative term of the currently active
controller by updating the shared state_of_plant structure.
```

```
*/
```

```
{
```

```
for (;;)
{
```

```
if ( compare_terms(plan_stack[next_action].operator_name,
                  "Turn_off_derivative") == TRUE &&
    time_up("Turn_off_derivative") == FALSE )
```

```
{
```

```
state_of_plant.derivative = LOW;
```

```
next_action++;
```

```
show_next_action();
```

```
}
```

```
yield();
```

```
}
```

```
}
```

```
void wait_for_proximity(void)
```

```
/*
```

```
This action waits until the monitoring action signals that
the level is now sufficiently close to the setpoint.
```

```
*/
```

```
{
```

```
for (;;)
{
```

```
if ( compare_terms(plan_stack[next_action].operator_name,
                  "Wait_for_proximity") == TRUE &&
    time_up("Wait_for_proximity") == FALSE &&
    task_flags.new_diagnosis_flag == ON )
```

```
{
```

```
if ( state_of_plant.proximity == TRUE )
```

```
{
```

```
sound(100);
```

```
sleep(1);
```

```
nosound();
```

```
next_action++;
```

```
show_next_action();
```

```
}
```

```
task_flags.new_diagnosis_flag = OFF;
```

```
}
```

```
yield();
```

```
}
```

```
}
```

```

void wait_for_steady_state(void)
/*
    This action waits until the monitoring action signals that
    the level is now at the setpoint and notifies the user.
*/
{
    for (;;)
    {
        if ( compare_terms(plan_stack[next_action].operator_name,
                           "Wait_for_steady_state") == TRUE &&
            time_up("Wait_for_steady_state") == FALSE &&
            task_flags.new_diagnosis_flag == ON )
        {
            if ( state_of_plant.steady_state == TRUE )
            {
                display_steady_state();
                sound(100);
                sleep(1);
                nosound();
                next_action++;
                show_next_action();
            }
            task_flags.new_diagnosis_flag = OFF;
        }
        yield();
    }
}

```

```

void sound_alarm(void)
/*
    This action sounds alarm and displays an alarm message in response
    to the signal from the monitoring action.
*/
{
    for (;;)
    {
        if ( compare_terms(plan_stack[next_action].operator_name,
                           "Sound_alarm") == TRUE &&
            time_up("Sound_alarm") == FALSE )
        {
            display_alarm_message();
            sound(440);
            sleep(1);
            nosound();
            next_action++;
            show_next_action();
        }
        yield();
    }
}

```

```
void emergency_shutdown(void)
```

```
/*
```

```
This action causes a graceful shutdown of the system, it shuts the
valve by taking over the physical control of the plant and it
disables most other actions (except write).
```

```
*/
```

```
{
```

```
for (;;)
{
```

```
if ( compare_terms(plan_stack[next_action].operator_name,
"Emergency_shutdown") == TRUE )
```

```
{
```

```
    plant_variables.controller_output = 0.0;
    display_shutdown_message();
    next_action = stack_top;
    state_of_plant.active = 0;
    task_flags.new_write_flag = ON;
```

```
}
```

```
yield();
```

```
}
```

```
void set_setpoint(void)
```

```
/*
```

```
This action sets the setpoint in the plant_variables structure
for further use by the control tasks.
```

```
*/
```

```
{
```

```
char buffer1[20];
char buffer2[20];
char string[5];
int setpoint;
void copy_term_end();
```

```
for (;;)
{
```

```
if ( compare_terms(plan_stack[next_action].operator_name,
"Set_setpoint") == TRUE &&
time_up("Set_setpoint") == FALSE )
```

```
{
```

```
    copy_term_end(buffer1, plan_stack[next_action].operator_name);
    copy_term_end(buffer2, buffer1);
    copy_term_end(string, buffer2);
    setpoint = atoi(string);
    plant_variables.setpoint = setpoint;
    next_action++;
    show_next_action();
```

```
}
```

```
yield();
```

```
void activate_standard_PI(void)
```

```
/*
```

This action switches on and then performs periodically the fixed PI control algorithm. All three control actions used here are based on the same incremental discrete PID algorithm.

```
*/
```

```
double measured = 0.0;
double setpoint = 0.0;
double error = 0.0;
double filtered_error = 0.0;
double filtered_difference = 0.0;
double previous_filt_diff = 0.0;
double output = 0.0;
double previous_output = 0.0;
```

```
static double max_output = 9.99;
static double min_output = 0.0;
static double filter_constant = 0.3;
```

```
double controller_gain = STANDARD_GAIN;
static double integral_term = STANDARD_INTEGRAL;
static double derivative_term = STANDARD_DERIVATIVE;
```

```
for (;;)
{
```

```
if ( ( state_of_plant.active == STANDARD_CONTROL ||
      compare_terms(plan_stack[next_action].operator_name,
                    "Activate_standard_PI") == TRUE ) &&
    ( task_flags.new_read_flag == ON ) )
```

```
{
    if ( compare_terms(plan_stack[next_action].operator_name,
                      "Activate_standard_PI") == TRUE )
    {
        state_of_plant.active = STANDARD_CONTROL;
        next_action++;
        show_next_action();
    }
}
```

```
measured = plant_variables.level;
setpoint = plant_variables.setpoint / 10;
error = setpoint - measured;
filtered_difference = filter_constant * (error - filtered_error);
filtered_error = filtered_error + filtered_difference;

output = previous_output + controller_gain * ( filtered_difference
```

```

        + ( SAMPLING_PERIOD / integral_term ) * filtered_error +
        ( derivative_term / SAMPLING_PERIOD ) *
        ( filtered_difference - previous_filt_diff ) );

previous_output = output;
if ( output > max_output ) output = max_output;
else if ( output < min_output ) output = min_output;
previous_filt_diff = filtered_difference;

plant_variables.controller_output = output;

task_flags.new_read_flag = OFF;
task_flags.new_write_flag = ON;

    }
    yield();
}
}

```

```

void activate_advanced(void)

```

```

/*
    This action switches on and then performs periodically the advanced
    control algorithm.
*/

```

```

{
    double measured = 0.0;
    double setpoint = 0.0;
    double error = 0.0;
    double filtered_error = 0.0;
    double filtered_difference = 0.0;
    double previous_filt_diff = 0.0;
    double output = 0.0;
    double previous_output = 0.0;

    static double max_output = 9.99;
    static double min_output = 0.0;
    static double filter_constant = 0.3;

    double controller_gain = LOW_GAIN;
    static double integral_term = HIGH_INTEGRAL;
    static double derivative_term = SMALL_DERIVATIVE;

    for (;;)
    {
        if ( ( state_of_plant.active == ADVANCED_CONTROL ||
              compare_terms(plan_stack[next_action].operator_name,
                           "Activate_advanced") == TRUE ) &&
            ( task_flags.new_read_flag == ON ) )
        {
            if ( compare_terms(plan_stack[next_action].operator_name,

```

```

        "activate_advanced") == TRUE )
    {
        state_of_plant.active = ADVANCED_CONTROL;
        next_action++;
        show_next_action();
    }

    if ( state_of_plant.gain == LOW && controller_gain == HIGH_GAIN )
    {
        controller_gain = LOW_GAIN;
        if ( previous_output > max_output )
            previous_output = max_output;
        if ( previous_output < min_output )
            previous_output = min_output;
    }
    else if ( state_of_plant.gain == HIGH &&
               controller_gain == LOW_GAIN )
        controller_gain = HIGH_GAIN;

    if ( state_of_plant.integral == LOW &&
          integral_term == HIGH_INTEGRAL )
        integral_term = LOW_INTEGRAL;
    else if ( state_of_plant.integral == HIGH &&
               integral_term == LOW_INTEGRAL )
        integral_term = HIGH_INTEGRAL;

    if ( state_of_plant.derivative == LOW &&
          derivative_term == LARGE_DERIVATIVE )
        derivative_term = SMALL_DERIVATIVE;
    else if ( state_of_plant.derivative == HIGH &&
               derivative_term == SMALL_DERIVATIVE )
        derivative_term = LARGE_DERIVATIVE;

    measured = plant_variables.level;
    setpoint = plant_variables.setpoint / 10;
    error = setpoint - measured;
    filtered_difference = filter_constant * (error - filtered_error);
    filtered_error = filtered_error + filtered_difference;

    output = previous_output + controller_gain * ( filtered_difference
        + ( SAMPLING_PERIOD * integral_term ) * filtered_error +
        ( derivative_term / SAMPLING_PERIOD ) *
        ( filtered_difference - previous_filt_diff ) );

    previous_output = output;
    previous_filt_diff = filtered_difference;

    if ( output > max_output ) output = max_output;
    else if ( output < min_output ) output = min_output;

    plant_variables.controller_output = output;

    task_flags.new_read_flag = OFF;

```

```

    task_flags.new_write_flag = ON;
    task_flags.new_data_flag = ON;

}
yield();
}
}

void activate_backup(void)
/*
    This action switches on and then performs periodically backup of
    the advanced control algorithm.
*/
{
    double measured = 0.0;
    double setpoint = 0.0;
    double error = 0.0;
    double filtered_error = 0.0;
    double filtered_difference = 0.0;
    double previous_filt_diff = 0.0;
    double output = 0.0;
    double previous_output = 0.0;

    static double max_output = 0.99;
    static double min_output = 0.0;
    static double filter_constant = 0.3;

    double controller_gain = LOW_GAIN;
    static double integral_term = HIGH_INTEGRAL;
    static double derivative_term = SMALL_DERIVATIVE;

    for (;;)
    {
        if ( ( state_of_plant.active == BACKUP_CONTROL ) )
            compare_terms(plan_stack[next_action].operator_name,
                "Activate_backup") == TRUE ) &&
            ( task_flags.new_read_flag == ON ) )
        {
            if ( compare_terms(plan_stack[next_action].operator_name,
                "Activate_backup") == TRUE )
            {
                state_of_plant.active = BACKUP_CONTROL;
                next_action++;
                show_next_action();
            }

            if ( state_of_plant.gain == LOW && controller_gain == HIGH_GAIN )
            {
                controller_gain = LOW_GAIN;
                if ( previous_output > max_output )

```

```

        previous_output = max_output;
        if ( previous_output < min_output )
            previous_output = min_output;
    }
    else if ( state_of_plant.gain == HIGH &&
              controller_gain == LOW_GAIN )
        controller_gain = HIGH_GAIN;

    if ( state_of_plant.integral == LOW &&
          integral_term == HIGH_INTEGRAL )
        integral_term = LOW_INTEGRAL;
    else if ( state_of_plant.integral == HIGH &&
              integral_term == LOW_INTEGRAL )
        integral_term = HIGH_INTEGRAL;

    if ( state_of_plant.derivative == LOW &&
          derivative_term == LARGE_DERIVATIVE )
        derivative_term = SMALL_DERIVATIVE;
    else if ( state_of_plant.derivative == HIGH &&
              derivative_term == SMALL_DERIVATIVE )
        derivative_term = LARGE_DERIVATIVE;

    measured = plant_variables.level;
    setpoint = plant_variables.setpoint / 10;
    error = setpoint - measured;
    filtered_difference = filter_constant * (error - filtered_error);
    filtered_error = filtered_error + filtered_difference;

    output = previous_output + controller_gain * ( filtered_difference
        + ( SAMPLING_PERIOD / integral_term ) * filtered_error +
        ( derivative_term / SAMPLING_PERIOD ) *
        ( filtered_difference - previous_filt_diff ) );

    previous_output = output;
    previous_filt_diff = filtered_difference;

    if ( output > max_output ) output = max_output;
    else if ( output < min_output ) output = min_output;

    plant_variables.controller_output = output;

    task_flags.new_read_flag = OFF;
    task_flags.new_write_flag = ON;
    task_flags.new_data_flag = ON;
}
yield();

```

```

#define AT(variable, constant, deviation) \
    ((variable > (constant - deviation)) && (variable < (constant + deviation)))
#define GREATER(variable, constant, deviation) \
    (variable > (constant + deviation))
#define LESS(variable, constant, deviation) \
    (variable < (constant - deviation))
#define NOTGREATER(variable, constant, deviation) \
    (variable < (constant + deviation))
#define NOTLESS(variable, constant, deviation) \
    (variable > (constant - deviation))

```

```

void main r(void)

```

```

/*

```

This section starts and performs periodical monitoring and diagnosis of the buffer-tank plant.

```

*/

```

```

{

```

```

    double level          = 0.0;
    double previous_level = 0.0;
    double flowrate       = 0.0;
    double controller_output = 0.0;
    double previous_output = 0.0;
    double level_derivative = 0.0;
    double setpoint        = 0.0;

```

```

    static double level_deviation      = 2.5;
    static double flowrate_deviation   = 7.0;
    static double derivative_deviation = 2.5;
    static double output_deviation     = 2.5;
    static double proximity_band       = 10.0;

```

```

    static double max_output      = 100.0;
    static double min_output      = 7.5;
    static double lower_level_bound = 0.0;
    static double upper_level_bound = 100.0;
    static double zero_flowrate    = 0.0;
    static double large_derivative = 80.0;
    static double abnormal_derivative = 400.0;
    static double overflow         = 90.0;

```

```

    static max_filling_rate = 1;

```

```

    for (;;)
    {

```

```

        if ( ( task_flags.monitor_flag == ON ) )
            compare_terms(plan_stack[next_action].operator_name,
                          "Monitor") == TRUE ) &&
            ( task_flags.new_data_flag == ON ) )
        {
            if ( compare_terms(plan_stack[next_action].operator_name,

```

```

        "Monitor") == TRUE )
    {
        task_flags.monitor_flag = ON;
        next_action++;
        show_next_action();
    }

    previous_level = level;
    level = plant_variables.level * 10;
    flowrate = plant_variables.flowrate * 10;
    setpoint = plant_variables.setpoint;
    previous_output = controller_output;
    controller_output = plant_variables.controller_output * 10;

    if ( sample_counter > 1 )
    {
        if ( previous_level != 0.0 )
            level_derivative = (level - previous_level) / SAMPLING_PERIOD
                               * 100 / var_filling_rate;
        plant_variables.level_derivative = level_derivative;

        if ( level < lower_level_bound || level > upper_level_bound ||
            level_derivative >= abnormal_derivative ||
            level_derivative <= -abnormal_derivative )
        {
            state_of_plant.sensor_failure = TRUE;
            state_of_plant.fatal_failure = TRUE;
            state_of_plant.alarm_condition = TRUE;
            state_of_plant.steady_state = FALSE;
            state_of_plant.proximity = FALSE;
            state_of_plant.overload = FALSE;
            state_of_plant.large_disturbance = FALSE;
            state_of_plant.gain = LOW;
            state_of_plant.integral = HIGH;
            state_of_plant.derivative = LOW;
            task_flags.plan_flag = ON;
        }
        else if ( state_of_plant.advanced_failure == TRUE &&
                 state_of_plant.backup_failure == TRUE )
        {
            state_of_plant.fatal_failure = TRUE;
            state_of_plant.alarm_condition = TRUE;
            state_of_plant.active = 0;
            task_flags.plan_flag = ON;
        }
        else if ( GREATER(level,overflow,level_deviation) )
        {
            state_of_plant.fatal_failure = TRUE;
            state_of_plant.alarm_condition = TRUE;
            task_flags.plan_flag = ON;
        }
    }

```

```
else
```

```
{
```

```
state_of_plant.alarm_condition = FALSE;
```

```
state_of_plant.steady_state = FALSE;
```

```
state_of_plant.proximity = FALSE;
```

```
if ( state_of_plant.advanced_failure == TRUE at
state_of_plant.active == ADVANCED_CONTROL ||
state_of_plant.backup_failure == TRUE at
state_of_plant.active == BACKUP_CONTROL )
```

```
{
state_of_plant.active = 0;
```

```
state_of_plant.alarm_condition = TRUE;
```

```
task_flag.plan_flag = ON;
```

```
if ( AT(level, setpoint, level_deviation) at
GREATER(previous_output, old_output, output_deviation) )
```

```
{
```

```
if ( state_of_plant.valve_failure == FALSE )
```

```
{
```

```
state_of_plant.valve_failure = TRUE;
```

```
state_of_plant.alarm_condition = TRUE;
```

```
task_flag.plan_flag = ON;
```

```
}
```

```
}
```

```
else
```

```
state_of_plant.valve_failure = FALSE;
```

```
if ( AT(level, setpoint, level_deviation) )
```

```
{
```

```
if ( AT(level_derivative, 0.0, derivative_deviation) )
```

```
{
```

```
state_of_plant.steady_state = TRUE;
```

```
state_of_plant.overload = FALSE;
```

```
state_of_plant.large_disturbance = FALSE;
```

```
}
```

```
}
```

```
else if ( AT(level, setpoint, proximity_band) )
```

```
{
```

```
if (( GREATER(level, setpoint, level_deviation) at
LESS(level_derivative, 0.0, derivative_deviation) at
GREATER(level_derivative, -large_derivative,
derivative_deviation) ) ||
```

```
( LESS(level, setpoint, level_deviation) at
```

```
GREATER(level_derivative, 0.0, derivative_deviation) at
```

```
LESS(level_derivative, large_derivative,
```

```
derivative_deviation) ))
```

```
{
```

```
state_of_plant.proximity = TRUE;
```

```
state_of_plant.large_disturbance = FALSE;
```

```
state_of_plant.overload = FALSE;
```

```
}
```

```

    }
    else if ((GREATER(level, setpoint, proximity_band) &&
              NOTLESS(level_derivative, 0.0, derivative_deviation))
             || (LESS(level, setpoint, proximity_band) &&
                 NOTGREATER(level_derivative, 0.0,
                             derivative_deviation)))
    {
        if ( state_of_plant.large_disturbance == FALSE )
        {
            state_of_plant.large_disturbance = TRUE;
            task_flags.plan_flag = ON;

            if ( state_of_plant.gain == HIGH &&
                  AT(previous_output, max_output, output_deviation) &&
                  state_of_plant.valve_failure == FALSE )
            {
                if ( state_of_plant.overload == FALSE )
                {
                    state_of_plant.overload = TRUE;
                    state_of_plant.alarm_condition = TRUE;
                    task_flags.plan_flag = ON;
                }
            }
            else
            {
                state_of_plant.overload = FALSE;
            }
        }

        task_flags.new_data_flag = OFF;
        task_flags.new_diagnosis_flag = ON;
    }
}

yield();
}
}

```

```
void read_plant(void)
```

```
/*
```

This task samples the plant by setting up the A/D card and reading the converted values into the corresponding global variables. It also sends the sampled values to the screen for real-time plots (but that is of secondary importance).

```
*/
```

```

{
    static int level_channel    = 5;
    static int flowrate_channel = 6;
    static int gain_code       = 0;
    double level;
    double setpoint;
    double output;
}

```

```

double flowrate;
int gain;

output = 0.0;
gain = LOW;

for (;;)
{
    if ( synchronize_sampling() == TRUE )
    {
        level = plant_variables.level;
        setpoint = plant_variables.setpoint;
        flowrate = plant_variables.flowrate;
        readcard(gain_code, level_channel, flowrate_channel);
        sample_counter++;
        linearize_flowrate();
        plant_variables.last_sample_time = biostime(0L)*CLOCK_RESOLUTION;
        plot_variables(level, setpoint, output, flowrate, gain);
        output = plant_variables.controller_output;
        gain = state_of_plant.gain;
        task_flags.new_read_flag = ON;
    }
    yield();
}

```

```

void write_to_plant(void)

```

```

/*

```

```

This task writes the output signal from the controller (stored
in a global variable) to the plant actuator, using the B/A
converter card;
*/

```

```

*/

```

```

{

```

```

    double compensated_output;

```

```

    for(;;)
    {

```

```

        if ( task_flags.new_write_flag == ON )
        {

```

```

            compensated_output = linearize_valve(plant_variables
                                                    controller_output);

```

```

            writetocard(compensated_output);

```

```

            task_flags.new_write_flag = OFF;
        }

```

```

        yield();
    }
}

```

```
void short_all(void)
```

```
/*
```

This task allows to suspend execution of the other tasks and to abort the multitasking environment altogether. It is also used to simulate a controller failure.

```
*/
```

```
{
```

```
  short_time;
```

```
  for(;;)
```

```
  {
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

```
    /* ... */
```

B.2 Sup-fcns.c

```
/*
```

filename: SUP-FCNS.C

description: More primitive functions used by actions and tasks.

written by: A. Korpala - 3/91

*/

void linearize_flowrate(void)

/*

Change the actual flowrate to the range 0 - 10 and linearize.
The voltage from the flowrate sensor (pressure transducer) reflects
a square of the actual flowrate and the approximate range is 0 - 3,
also the zero reading is about 0.1 (offset of 0.1).

*/

{

double offset_flowrate;

offset_flowrate = plant_variables.flowrate - 0.1;

if (offset_flowrate <= 0)

offset_flowrate = 0.001;

plant_variables.flowrate = sqrt(offset_flowrate) * 10/3;

double linearize_valve(controller_output)

double controller_output;

/*

Roughly linearize the response of the pneumatic valve for signals
between 0 and 10 Volts. The valve is extremely nonlinear (stiction,
velocity limited motion etc.) and this is a rough model, based on
observed deviations.

*/

{

double modified_output;

if (controller_output < 0.25)

modified_output = 0.0;

else if (controller_output < 1.0)

modified_output = controller_output + 1.0;

else if (controller_output < 6.0)

modified_output = controller_output + 1.5;

else if (controller_output < 7.0)

modified_output = controller_output + 1.0;

else if (controller_output < 8.0)

modified_output = controller_output + 0.5;

else

modified_output = controller_output;

```

    return(modified_output);
}

```

```

time_up(action_name)
char *action_name;

```

```

/*
    Use timing functions provided by BIOS (with 55msec clock res.) to
    determine if the next sampling time is approaching and if yes
    return 1, otherwise return 0.
*/

```

```

{
    float current_time;
    float time_passed;
    float time_left;
    float execution_time;

    if ( plant_variables.last_sample_time != 0.0 )
    {
        current_time = biostime(0,0L) * CLOCK_RESOLUTION;
        time_passed = current_time - plant_variables.last_sample_time;
        time_left = SAMPLING_PERIOD - time_passed;
        if ( strcmp(action_name,"Pump") == TRUE )
            execution_time = TIME_MARGIN + 1;
        else
            execution_time = TIME_MARGIN;
        if ( time_left < execution_time )
            return(1);
    }
    return(0);
}

```

```

synchronize_sampling(void)
/*

```

```

    Use timing functions provided by BIOS (with 55msec clock res.) to
    dynamically adjust the execution time of supervising actions, thus
    ensuring that sampling occurs at constant time intervals as required
    by the digital controllers. If it is not time yet, return 0.
*/

```

```

{
    float current_time;
    float time_passed;
    float time_left;
    unsigned duration;

    if ( plant_variables.last_sample_time != 0.0 )
    {
        current_time = biostime(0,0L) * CLOCK_RESOLUTION;
        time_passed = current_time - plant_variables.last_sample_time;

```

```

time_left = SAMPLING_PERIOD - time_passed;
if ( time_left > ( TIME_MARGIN + 1 ) )
    return(0);
else
{
    duration = time_left * 1000;
    delay(duration);
}
}
return(1);
}

```

```

void display_alarm_message(void)
/*
    Display alarm message on the screen.
*/
{
    colour_viewport(X1,Y1,X5,Y2-2,SPECIAL_COLOUR);
    settextstyle(SPECIAL_FONT,HORIZ_DIR,SPECIAL_SIZE);
    setcolor(TEXT_COLOUR);
    outtextxy(220,4,"ALARM CONDITION");
}

```

```

void display_steady_state(void)
/*
    Display a message on the screen informing that a steady state
    condition has been reached by the plant.
*/
{
    colour_viewport(X1,Y1,X5,Y2-2,TEXT_BKG);
    settextstyle(SPECIAL_FONT,HORIZ_DIR,SPECIAL_SIZE);
    setcolor(TEXT_COLOUR);
    outtextxy(120,4,"STEADY STATE CONDITION REACHED");
}

```

```

void display_shutdown_message(void)
/*
    Display a message on the screen that the plant has been shut down.
*/
{
    colour_viewport(X1,Y1,X5,Y2-2,SPECIAL_COLOUR);
    settextstyle(SPECIAL_FONT,HORIZ_DIR,SPECIAL_SIZE);
    setcolor(TEXT_COLOUR);
    outtextxy(180,4,"EMERGENCY SHUTDOWN");
}

```

```

void display_states(present_adr, goal_adr)
char *present_adr;
char *goal_adr;
/*
    Show the present and goal states in full.
*/
{
    register i;

    colour_viewport(X1,Y1,X5,Y2-2,TEXT_BKG);
    settxtstyle(TEXT_FONT,HORIZ_DIR,TEXT_SIZE);
    setcolor(TEXT_COLOUR);
    outtextxy(230,18,"Special message area");

    colour_viewport(X1,Y2,X2-2,Y3-2,TEXT_BKG);
    settxtstyle(TEXT_FONT,HORIZ_DIR,TEXT_SIZE);
    setcolor(TEXT_COLOUR);
    print_state(present_adr,"STARTING STATE: ");

    colour_viewport(X2,Y2,X3-2,Y3-2,TEXT_BKG);
    settxtstyle(TEXT_FONT,HORIZ_DIR,TEXT_SIZE);
    setcolor(TEXT_COLOUR);
    print_state(goal_adr,"GOAL STATE: ");
}

```

```

void arrow(x_pos, y_pos)
int x_pos;
int y_pos;
/*
    Draw an arrow at the given position.
*/
{
    line(x_pos, y_pos+TEXT_CHAR_HEIGHT/2-1, x_pos+15,
        y_pos+TEXT_CHAR_HEIGHT/2-1);
    line(x_pos+11, y_pos+1-1, x_pos+15, y_pos+TEXT_CHAR_HEIGHT/2-1);
    line(x_pos+11, y_pos+TEXT_CHAR_HEIGHT-1-1, x_pos+15,
        y_pos+TEXT_CHAR_HEIGHT/2-1);
    line(x_pos+11, y_pos+1-1, x_pos+11, y_pos+TEXT_CHAR_HEIGHT-1-1);
}

```

```

void show_next_action(void)
/*
    Move the location of the arrow to point to the next action on the
    screen.
*/
{
    setviewport(X4,Y2,X5,Y3-2,1);
}

```

```

    if ( next_action > 1 )
    {
        setcolor(TEXT_BKG);
        arrow(3, (next_action) * TEXT_CHAR_HEIGHT);
    }
    if ( next_action < stack_top )
    {
        setcolor(ARROW_COLOUR);
        arrow(3, (1+next_action) * TEXT_CHAR_HEIGHT);
    }
}

void display_plan(void)
/*
    Display the solution plan generated by the planner and the expected
    final state, on the same screen as the present and goal states.
*/
{
    char string[30];
    register i;

    colour_viewport(X3,Y2,X4-2,Y3-2,TEXT_BKG);
    setttextstyle(TEXT_FONT,HORIZ_DIR,TEXT_SIZE);
    setcolor(TEXT_COLOUR);
    print_state(plan_stack[stack_top-1].intermediate_state,
        "EXPECTED STATE:");

    colour_viewport(X4,Y2,X5,Y3-2,TEXT_BKG);
    setttextstyle(TEXT_FONT,HORIZ_DIR,TEXT_SIZE);
    setcolor(TEXT_COLOUR);
    outtextxy(22,3,"SOLUTION P. N:");

    for( i = 1 ; i < stack_top ; i++ )
    {
        sprintf(string,"%d%s %s",i,":",plan_stack[i].operator_name);
        outtextxy(22, (i+1) * TEXT_CHAR_HEIGHT, string);
    }

    setcolor(ARROW_COLOUR);
    arrow(3, 2 * TEXT_CHAR_HEIGHT);
    setttextstyle(SMALL_FONT,HORIZ_DIR,1);
    setcolor(TEXT_COLOUR);
    setviewport(X1,Y1,X5,Y4,1);
    outtextxy(GRAPH1_X + sample_counter, GRAPH1_Y - 5, "P");
}

void plot_value(x_origin, y_origin, y_span, y_range, y_value1, y_value2,
    colour)
int x_origin;

```

```

int y_origin;
int y_span;
int y_range;
double y_value1;
double y_value2;
int colour;
/*
    Plot the next point of a graph located at x and y origin, with the y
    axis y_span pixels long, corresponding to the actual range y_range.
    To ensure continuous plots use the line() function, also use the given
    colour. Advance by 1 pixel along the x axis.
*/
{
    int x_pos;
    int y_pos1;
    int y_pos2;
    int y1;
    int y2;

    x_pos = x_origin + sample_counter;
    y1 = y_value1 * y_span/y_range + 0.5;
    y_pos1 = y_origin - y1;
    y2 = y_value2 * y_span/y_range + 0.5;
    y_pos2 = y_origin - y2;
    setcolor(colour);
    line(x_pos-1, y_pos1, x_pos, y_pos2);
}

void plot_variables(level, setpoint, output, flowrate, gain)
double level;
double setpoint;
double output;
double flowrate;
int gain;
/*
    Plot the setpoint, level, flowrate, controller output and controller
    gain on the three sets of axes prepared by setup_control_screen().
*/
{
    char string[30];
    int minutes;
    int seconds;
    int combined_time;
    int x_pos;
    int y_pos1;
    int y_pos2;

    combined_time = sample_counter * SAMPLING_PERIOD;
    minutes = combined_time / 60;
    seconds = combined_time - (minutes * 60);

```

```

setviewport(16,Y3+1,200,Y3+10,1);
clearviewport();
settextstyle(TEXT_FONT,HORIZ_DIR,TEXT_SIZE);
setcolor(TEXT_COLOUR);
sprintf(string,"%d%d",Time(mm:ss) = ",minutes,".",seconds);
outtextxy(0,2,string);
setviewport(220,Y3+1,420,Y3+10,1);
clearviewport();
sprintf(string,"%d",Sample counter = ",sample_counter);
outtextxy(0,2,string);

setviewport(11,Y1,X5,Y4,1);

plot_value(GRAPH1_X, GRAPH1_Y, 60, 100, setpoint,
            (double) plant_variables.setpoint, PLOT2_COLOUR);
plot_value(GRAPH1_X, GRAPH1_Y, 60, 10, level, plant_variables.level,
            PLOT1_COLOUR);
plot_value(GRAPH2_X, GRAPH2_Y, 60, 10, flowrate,
            plant_variables.flowrate, PLOT2_COLOUR);
plot_value(GRAPH2_X, GRAPH2_Y, 60, 10, output,
            plant_variables.controller_output, PLOT1_COLOUR);

if ( state_of_plant.active != STANDARD_CONTROL )
{
    x_pos = GRAPH3_X + sample_counter;
    if ( gain == LOW )
        y_pos1 = GRAPH3_Y;
    else if ( gain == HIGH )
        y_pos1 = GRAPH3_Y - 18;
    if ( state_of_plant.gain == LOW )
        y_pos2 = GRAPH3_Y;
    else if ( state_of_plant.gain == HIGH )
        y_pos2 = GRAPH3_Y - 18;
    setcolor(PLOT3_COLOUR);
    line(x_pos-1, y_pos1, x_pos, y_pos2);
}
}

```

B.3 Sup-ifac.c

filename: SUP-IFAC.C

description: Turbo C (2.0) specific, screen input and output routines
for the user interface of the supervisor.

written by: A. Korpala - 3/91

```

#include <graphics.h>
#include <conio.h>

```

```

void colour_window(left, top, right, bottom, bkg_colour, txt_colour)
int left, top, right, bottom;
int bkg_colour, txt_colour;
/*
    Define a window on the screen and fill it with the given colour.
*/
{
    textbackground(bkg_colour);
    textcolor(txt_colour);
    window(left, top, right, bottom);
    clrscr();
}

```

```

void colour_viewport(left, top, right, bottom, bkg_colour)
int left;
int top;
int right;
int bottom;
int bkg_colour;
/*
    Define a viewport with the given coordinates, clear it, fill with
    the given colour and draw a border around it.
*/
{
    setviewport(left, top, right, bottom, 1);
    clearviewport();
    setcolor(GRAPH_COLOR);
    rectangle(0, 0, right-left, bottom-top);
    setfillstyle(SOLID_FILL, bkg_colour);
    floodfill(1, 1, GRAPH_COLOUR);
}

```

```

void print_state(state_adr, state_name)
char *state_adr;
char *state_name;
/*
    Prints the given state relative to the current viewport in
    graphics mode.
*/
{
    char *state_term[TERMS];

```

```

register i;

i = 0;
state_term[0] = state_adr;
outtxtry(TEXT_CHAR_WIDTH, 3, state_name);
settextstyle(LABEL_FONT, HORIZ_DIR, SCALE_SIZE);

while ( *state_term[i] != END )
{
    outtxtry(TEXT_CHAR_WIDTH, (i+2) * TEXT_CHAR_HEIGHT, state_term[i]);
    i++;
    state_term[i] = state_adr + (i * LETTERS);
}

}

void read_state(state_adr)
char *state_adr;
/*
    Allow entry of a state, i.e., group of terms, given the address
    of the beginning of a 2d array.
*/
{
    register i;
    char *state_term[TERMS];

    i = 0;
    textbackground(BLUE);
    textcolor(YELLOW);
    window(1,1,80,25);
    gotoxy(2,1);
    cprintf("%s", "Enter the terms one by one.",
        " terminating the state with the '0' term.");
    textbackground(WHITE);
    textcolor(BLUE);
    window(10,5,40,20);
    do
    {
        if (i < USER_TERMS)
        {
            gotoxy(2,i+3);
            state_term[i] = state_adr + (i * LETTERS);
            scanf( "%s", state_term[i] );
        }
        else
            *state_term[i] = END;
        i++;
    } while ( *state_term[i-1] != END);
    colour_window(1,1,80,25,BLACK,WHITE);
}

```

```

void get_goal(goal_state_adr)
char *goal_state_adr;
/*
    Prompt the user to enter a goal state.
*/
{
    colour_window(1,1,80,25,BLUE,YELLOW);
    colour_window(10,5,40,20,WHITE,BLUE);
    cprintf(" GOAL STATE:");
    read_state(goal_state_adr);
}

```

```

void draw_graph_axes(x_origin, y_origin, x_span, y_span)
int x_origin;
int y_origin;
int x_span;
int y_span;
/*
    Draw x and y axes for plotting purposes, starting at the origin
    point and extending for x_span and y_span, respectively.
*/
{
    setcolor(GRAPH_COLOUR);
    line(x_origin, y_origin, x_origin+x_span, y_origin);
    line(x_origin, y_origin, x_origin, y_origin-y_span);
}

```

```

void scale_graph_axes( x_origin, y_origin, x_span, y_span, x_min_division,
                      y_min_division, x_numbered_division,
                      y_numbered_division, x_number, y_number )

int x_origin;
int y_origin;
int x_span;
int y_span;
int x_min_division;
int y_min_division;
int x_numbered_division;
int y_numbered_division;
int x_number;
int y_number;
/*
    Provide scaling on a set of axes drawn with draw_graph_axes(),
    by marking off each axis every min_division, and numbering each
    axis every numbered_division (these numbers are calculated from
    the given basic numbers).
*/
{

```

```

register i, j;
int value;
char value_string[5];
unsigned length;

settextstyle(LABEL_FONT, HORIZ_DIR, SCALE_SIZE);
setcolor(GRAPH_COLOUR);

for ( i = x_origin; i <= x_origin + x_span; i = i + x_min_division)
    putpixel(i, y_origin+1, GRAPH_COLOUR);

for ( i = x_origin, j = 0; i <= x_origin + x_span;
      i = i + x_numbered_division, j++ )
{
    value = j * x_number;
    itoa(value, value_string, 10);
    length = strlen(value_string);
    outtextxy(i-(SCALE_CHAR_WIDTH/2+length), y_origin+1, value_string);
}

for ( i = y_origin; i >= y_origin - y_span; i = i - y_min_division)
    putpixel(x_origin-1, i, GRAPH_COLOUR);

for ( i = y_origin, j = 0; i >= y_origin - y_span;
      i = i - y_numbered_division, j++ )
{
    value = j * y_number;
    itoa(value, value_string, 10);
    length = strlen(value_string);
    outtextxy(x_origin-(SCALE_CHAR_WIDTH+length)-3,
              i-SCALE_CHAR_HEIGHT/2, value_string);
}

void label_graph_axes(x_origin, y_origin, x_span, y_span, x_label,
                     y_label1, y_label2)
int x_origin;
int y_origin;
int x_span;
int y_span;
char *x_label;
char *y_label1;
char *y_label2;
/*
    Label a set of axes drawn with draw_graph_axes(), with a single
    label for the x axis and 1 or 2 labels for the y axis.
*/
{
    int length1;
    int length2;
    int length;

```

```
settextstyle(LABEL_FONT,HORIZ_DIR,LABEL_SIZE);
```

```
length1 = strlen(y_label1);
length2 = strlen(y_label2);
length = length1;
if ( length2 > length )
    length = length2;
```

```
setcolor(PLOT1_COLOUR);
outtextxy( x_origin - 2 - (LABEL_CHAR_WIDTH * (length) + 25),
           y_origin - y_span + LABEL_CHAR_HEIGHT/2, y_label1);
if ( length2 > 0 )
{
    setcolor(PLOT2_COLOUR);
    outtextxy( x_origin - 2 - (LABEL_CHAR_WIDTH * (length) + 25),
               y_origin - y_span + LABEL_CHAR_HEIGHT * 1.5 + 2, y_label2);
}
length = strlen(x_label);
setcolor(GRAPH_COLOUR);
outtextxy( x_origin + x_span - (LABEL_CHAR_WIDTH * length),
           y_origin + 2, x_label);
}
```

```
void digital_axis(x_origin, y_origin, y_span, y_label1, y_label2)
int x_origin;
int y_origin;
int y_span;
char *y_label1;
char *y_label2;
```

```
/*
   Draw a single short vertical axis with only two markings labelled
   with two descriptive values, for plotting a digit: (2 valued)
   variable;
*/
```

```
{
    int length1;
    int length2;

    settextstyle(LABEL_FONT,HORIZ_DIR,SCALE_SIZE);
    setcolor(GRAPH_COLOUR);

    line(x_origin, y_origin, x_origin, y_origin - y_span);
    putpixel(x_origin-1, y_origin, GRAPH_COLOUR);
    putpixel(x_origin-1, y_origin - y_span, GRAPH_COLOUR);

    length1 = strlen(y_label1);
    length2 = strlen(y_label2);

    outtextxy( x_origin - 2 - (SCALE_CHAR_WIDTH * length1),
```

```

check_add_effects(operator_adr, present_adr, goal_adr)
struct operator *operator_adr;
char *present_adr;
char *goal_adr;
/*
 * Check if any of the add_effects of the operator considered is
 * absent from the present state, but present in the goal state, if
 * yes return 1, if no return 0.
 */
{
    char *present_term[TERMS];
    char *goal_term[TERMS];
    int match1_flag;
    int match2_flag;
    register i, j;

    i = 0;
    match1_flag = 0;
    match2_flag = 0;
    while ( *operator_adr->add_effect[i] != END )
    {
        j = 0;
        goal_term[j] = goal_adr;
        while ( *goal_term[j] != END && match1_flag == OFF )
        {
            if ( compare_terms( operator_adr->add_effect[i], goal_term[j] )
                == TRUE )
                match1_flag = ON;
            j++;
            goal_term[j] = goal_adr + (j * LETTERS);
        }
        if ( match1_flag == ON )
        {
            j = 0;
            present_term[j] = present_adr;
            while ( *present_term[j] != END && match2_flag == OFF )
            {
                if ( compare_terms( operator_adr->add_effect[i], present_term[j] )
                    == TRUE )
                    match2_flag = ON;
                j++;
                present_term[j] = present_adr + (j * LETTERS);
            }
            if ( match2_flag == OFF ) return(1);
        }
        i++;
    }
    return(0);
}

```

```

        y_origin - y_span, y_label1);
outtextxy( x_origin - 2 - (SCALE_CHAR_WIDTH * length2),
        y_origin - SCALE_CHAR_HEIGHT, y_label2);
}

```

```

void setup_control_screen(void)

```

```

/*
    Initialize and set up the screen, to make it ready for real time
    plotting.
*/
{
    char string[30];
    char string1[10];
    char string2[10];

    colour_viewport(X1, Y3, X5, Y4, BLACK);
    setviewport(X1, Y1, X5, Y4, 1);
    draw_graph_axes(GRAPH1_X, GRAPH1_Y, 540, 60);
    scale_graph_axes(GRAPH1_X, GRAPH1_Y, 540, 60, 12, 6, 120, 30, 10, 50);
    label_graph_axes(GRAPH1_X, GRAPH1_Y, 540, 80, "time", "level", "setpt");
    draw_graph_axes(GRAPH2_X, GRAPH2_Y, 540, 60);
    scale_graph_axes(GRAPH2_X, GRAPH2_Y, 540, 60, 12, 6, 120, 30, 10, 50);
    label_graph_axes(GRAPH2_X, GRAPH2_Y, 540, 60, "time", "output", "flow");
    sprintf(string1, "%s %d", "HIGH", HIGH_GAIN);
    sprintf(string2, "%s %d", "LOW", LOW_GAIN);
    digit_1((GRAPH3_X, GRAPH3_Y, 18, string1, string2);

    setviewport(433, Y3+1, 633, Y3+10, 1);
    sprintf(string, "%s %d", "Sampling period (s) = ", SAMPLING_PERIOD);
    settextstyle(TEXT_FONT, HORIZ_DIR, TEXT_SIZE);
    outtextxy(0, 2, string);
}

```

```
check_operator(operator_adr, present_adr, goal_adr)
```

```
struct operator *operator_adr;
```

```
char *present_adr;
```

```
char *goal_adr;
```

```
/*
```

```
Check if the preconditions allow the operator to be applied and
if the effects make it more promising; return 0 if preconditions
violated, 3 if preconditions satisfied but effects not helping,
2 if either one of add or del effects helping, 1 if both.
```

```
/
```

```
{
```

```
int value = 0;
```

```
if ( check_neg_conditions(operator_adr, present_adr) == TRUE )
```

```
{
```

```
if ( check_pos_conditions(operator_adr, present_adr) == TRUE )
```

```
{
```

```
value = 3;
```

```
if (check_del_effects(operator_adr, present_adr, goal_adr) == TRUE)
```

```
value--;
```

```
if (check_add_effects(operator_adr, present_adr, goal_adr) == TRUE)
```

```
value--;
```

```
}
```

```
}
```

```
return(value);
```

```
}
```

```
void copy_operator(operator1_adr, operator2_adr)
```

```
struct operator *operator1_adr;
```

```
struct operator *operator2_adr;
```

```
/*
```

```
Copy the contents of the structure at address operator2_adr into
the structure at address operator1_adr.
```

```
*/
```

```
{
```

```
strcpy(operator1_adr->name, operator2_adr->name);
```

```
copy_state(operator1_adr->pos_condition, operator2_adr->pos_condition);
```

```
copy_state(operator1_adr->neg_condition, operator2_adr->neg_condition);
```

```
copy_state(operator1_adr->add_effect, operator2_adr->add_effect);
```

```
copy_state(operator1_adr->del_effect, operator2_adr->del_effect);
```

```
}
```

```
void push_on_stack(operator_name, state_adr)
```

```
char *operator_name;
```

```
char *state_adr;
```

```
/*
```

```
Push the given item onto top of the search stack and fill in the
place above.
```

```
*/
```

Appendix C

Planning task

The source code for the AI-based part of the system is listed in this appendix. It includes the planning task and the lower level functions used by it.

C.1 Sup-plan.c

```
/*  
-----  
filename:    SUP-PLAN.C  
description: The planning task.  
written by:  A. Korpala - 3/91  
-----  
*/
```

```
#include "plan-def.c"  
#include "pl-co-df.c"  
  
#include "plan-fcn.c"  
#include "pl-co-fc.c"  
#include "pl-su-fc.c"
```

```
void plan(void)  
{  
    char terminal_state[2][LETTERS] =  
        { "plant_down", "0" };  
    char present_state[TERMS][LETTERS];  
    char goal_state[TERMS][LETTERS];
```

```

{
    if ( stack_top < STACKSIZE )
    {
        strcpy(plan_stack[stack_top].operator_name, operator_name);
        copy_state(plan_stack[stack_top].intermediate_state, state_adr);
        *(plan_stack[stack_top].inhibited_operators[0]) = END;
        stack_top++;
    }
    else
        printf("\aStack full.\n");
}

```

```

void pop_off_stack(operator_name, state_adr)
char *operator_name;
char *state_adr;
/*
    Pop off the item from top of the stack.
*/
{
    if ( stack_top > 1 )
    {
        stack_top = stack_top - 2;
        strcpy(operator_name, plan_stack[stack_top].operator_name);
        copy_state(state_adr, plan_stack[stack_top].intermediate_state);
        stack_top++;
    }
    else
        printf("\aStack empty.\n");
}

```

```

void put_on_record(operator_name, operator_rating)
char *operator_name;
//int operator_rating;
/*
    Put the given item on the record.
*/
{
    if ( record_top < RECORDSIZE )
    {
        strcpy(record[record_top].operator_name, operator_name);
        record[record_top].operator_rating = operator_rating;
        record_top++;
    }
}

```

```

int driver;
int mode;

get_goal(goal_state);

driver = BGA;
mode = EGANI;
initgraph(&driver, &mode, "");

setup_control_screen();

for (;;)
{
    if ( task_flags.plan_flag == ON && time_up("Plan") == FALSE )
    {
        stack_top = 0;
        current_bottom = 0;
        make_abstract_state(present_state);
        display_states(present_state, goal_state);
        initialize_operators(present_state, goal_state);
        search(present_state, goal_state, terminal_state);
        display_plan();
        next_action = 1;
        task_flags.plan_flag = OFF;
    }
    yield();
}
}

```

C.2 Plan-fcn.c

```

/*
-----
filename:    PLAN-FCN.C

description: Problem independent functions used by the planning task.

written by:  A. Korpala - 11/80
-----
*/

end_of_search(present_state, goal_state, terminal_state)
char *present_state;
char *goal_state;
char *terminal_state;
/*
Returns 1 if the search should be terminated, 0 if not.

```

```
check_inhibited_operators(operator_name)
```

```
char *operator_name;
```

```
/*
```

```
Check if the given operator has been tried and failed before.
return 1 if yes, 0 if not.
```

```
*/
```

```
{
```

```
register i;
```

```
i = 0;
```

```
while( *(plan_stack[stack_top-1].inhibited_operators[i]) != END &&
        *(plan_stack[stack_top-1].inhibited_operators[i]) != '\0' &&
        i < INH_OPERATORS )
```

```
{
```

```
if ( compare_terms(operator_name, plan_stack[stack_top-1].
        inhibited_operators[i] ) == TRUE )
```

```
return(1);
```

```
i++;
```

```
}
```

```
return(0);
```

```
}
```

```
void inhibit_operator(void)
```

```
/*
```

```
On backtracking, memorize the operator which has just failed.
```

```
*/
```

```
{
```

```
register i;
```

```
for(i=0; *(plan_stack[stack_top-2].inhibited_operators[i]) != END &&
        *(plan_stack[stack_top-2].inhibited_operators[i]) != '\0' &&
        i < INH_OPERATORS; i++);
```

```
strcpy(plan_stack[stack_top-2].inhibited_operators[i],
        plan_stack[stack_top-1].operator_name);
```

```
i++;
```

```
*(plan_stack[stack_top-2].inhibited_operators[i]) = END;
```

```
}
```

```
void uninhibit_operators(void)
```

```
/*
```

```
Clear all inhibited operators from the starting node.
```

```
*/
```

```
{
```

```
*(plan_stack[stack_top-1].inhibited_operators[0]) = END;
```

```
}
```

```

*/
{
    int match_type;

    match_type = compare_states(present_state, goal_state);
    if (( match_type == 1 || match_type == 2 || (match_type == 3 &&
        second_mode_flag == ON) ) || (compare_states(present_state,
        terminal_state) == 2 ))
        return(1);
    else return(0);
}

```

```

compare_states(super_state_adr, sub_state_adr)

```

```

char *sub_state_adr;

```

```

char *super_state_adr;

```

```

/*

```

Compares the two states and returns 1 if identical, 2 if super_state contains sub_state, 3 if if super_state contains the essential subset of the sub_state and 0 otherwise.

```

*/

```

```

{
    char *sub_state_term[TERMS], *super_state_term[TERMS];
    int partial_match_flag, unessential_flag, match_flag;
    register i, j;

```

```

    i = 0;

```

```

    partial_match_flag = OFF;

```

```

    sub_state_term[0] = sub_state_adr;

```

```

    while ( *sub_state_term[i] != END )

```

```

    {

```

```

        j = 0;

```

```

        match_flag = OFF;

```

```

        unessential_flag = OFF;

```

```

        super_state_term[0] = super_state_adr;

```

```

        while ( *super_state_term[j] != END && match_flag == OFF )

```

```

        {

```

```

            if ( compare_terms( sub_state_term[i], super_state_term[j] ) == 1)

```

```

                match_flag = ON;

```

```

            if ( compare_terms( sub_state_term[i], super_state_term[j] ) == 2)

```

```

                unessential_flag = ON;

```

```

            j++;

```

```

            super_state_term[j] = super_state_adr + (j * LETTERS);

```

```

        }

```

```

        if ( match_flag == OFF && unessential_flag == OFF)

```

```

            return(0);

```

```

        if ( match_flag == OFF && unessential_flag == ON)

```

```

            partial_match_flag = ON;

```

```

        i++;

```

```

        sub_state_term[i] = sub_state_adr + (i * LETTERS);

```

```

    }

```

```

void search(present_state, goal_state, terminal_state)
char *present_state;
char *goal_state;
char *terminal_state;
/*
    Perform a difference/data driven, depth-first search by
    itself recursively and storing the current search-path on the
    plan stack.
*/
{
    int index, rating;
    int search_type;
    char name[LETTERS];
    int failed_flag;
    register i;

    if ( stack_top < STACKSIZE )
    {
        if ( end_of_search(present_state, goal_state, terminal_state)
            == FALSE )
        {
            if (stack_top == 0)
                push_on_stack("Initial_state", present_state);
            failed_flag = OFF;
            rating = 0;
            search_type = 0;
            index = 0;
            i = 0;

            while( *(operator_buffer[i].name) != END && rating != 1 )
            {
                if (check_inhibited_operators(operator_buffer[i].name) == FALSE)
                {
                    rating = check_operator(operator_buffer[i], present_state,
                                            goal_state);

                    if ( rating == 1 )
                    {
                        search_type = 1;
                        index = i;
                    }
                    else if ( rating == 2 && search_type != 2 )
                    {
                        search_type = 2;
                        index = i;
                    }
                    else if ( rating == 3 && search_type == 0 )
                    {
                        search_type = 3;
                        index = i;
                    }
                }
                i++;
            }
        }
    }
}

```

```

j = 0;
super_state_term[0] = super_state_adr;
while ( *super_state_term[j] != END )
{
    j++;
    super_state_term[j] = super_state_adr + (j * LETTERS);
}
if ( partial_match_flag == ON ) return(3);
else if ( i == j ) return(1);
else return(2);
}

```

```

void copy_state(state1_adr, state2_adr)
char *state1_adr, *state2_adr;
/*
    Copy state at address state2_adr to state at address state1_adr.
*/
{
    char *state1_term[TERMS], *state2_term[TERMS];
    register i;

    i = 0;
    do
    {
        state1_term[i] = state1_adr + (i * LETTERS);
        state2_term[i] = state2_adr + (i * LETTERS);
        strcpy( state1_term[i], state2_term[i] );
        i++;
    } while( *state2_term[i-1] != END );
}

```

```

void compress_state(state_adr)
char *state_adr;
/*
    Eliminate all empty terms (i.e., strings starting with '\0') from
    the given state.
*/
{
    char *term[TERMS];
    register i, j;

    i = j = 0;
    term[0] = state_adr;

    while( *term[i] != END )
    {
        if ( *term[i] != '\0' )
        {

```

```

    }
    if ( search_type != 0 )
    {
        update_present_state(&operator_buffer[index], present_state);
        push_on_stack(operator_buffer[index].name, present_state);
    }
    else if ( stack_top > current_bottom )
    {
        inhibit_operator();
        pop_off_stack(name, present_state);
    }
    else if ( stack_top == current_bottom && second_mode_flag == OFF )
    {
        second_mode_flag = ON;
        uninhibit_operators();
    }
    else
    {
        failed_flag = ON;
        printf("\n\n Search unsuccessful - no solution found.");
        getch();
    }
    if ( failed_flag == OFF )
        search(present_state, goal_state, terminal_state);
}
current_bottom = stack_top;
}

```

C.3 Pl-co-fc.c

```

/*
-----
filename:      PL-CU-FC.C

description:   Initialization routines for operators specific to the
               control problem (case study) and other problem specific
               functions.

written by:    A. Korpala - 11/90
-----
*/

```

```

void expand_state(state_adr)
char *state_adr;

```

```

/*

```

```

        if ( i != j )
            strcpy( term[j], term[i]);
        j++;
    }
    i++;
    term[i] = state_adr + (i * LETTERS);
    term[j] = state_adr + (j * LETTERS);
}
*term[j] = END;
}

```

```

void del_from_state(operator_adr, present_adr)
struct operator *operator_adr;
char *present_adr;
/*
Delete del_effects of the given operator from the given present
state.
*/
{
    char *present_term[TERMS];
    register i, j;
    int match_flag;

    i = 0;
    present_term[0] = present_adr;

    while( *present_term[i] != END )
    {
        j = 0;
        match_flag = OFF;
        while ( *(operator_adr->del_effect)[j] != END && match_flag==OFF )
        {
            if ( compare_terms(present_term[i], operator_adr->del_effect[j])
                == TRUE )
            {
                *present_term[i] = '\0';
                match_flag = ON;
            }
            j++;
        }
        i++;
        present_term[i] = present_adr + (i * LETTERS);
    }
    compress_state(present_adr);
}

```

Add certain terms which the user does not have to enter, but which are needed by the planner.

```

*/
{
    char *state_term[TERMS];
    char term[LETTERS];
    char value[LETTERS];
    int match_flag;
    register i;

    i = 0;
    match_flag = OFF;
    state_term[0] = state_addr;
    while( *state_term[i] != END && match_flag == OFF )
    {
        match_flag = check_term_start(state_term[i], "setpoint");
        i++;
        state_term[i] = state_addr + (i * LETTERS);
    }
    if ( match_flag == OFF )
    {
        i = 0;
        state_term[0] = state_addr;
        while( *state_term[i] != END )
        {
            if ( check_term_start(state_term[i], "level") == TRUE )
            {
                copy_term_end(value, state_term[i]);
                match_flag = ON;
            }
            i++;
            state_term[i] = state_addr + (i * LETTERS);
        }
        if ( match_flag == ON )
        {
            strcpy(term, "setpoint_");
            strcat(term, value);
            strcpy(state_term[i], term);
            i++;
            state_term[i] = state_addr + (i * LETTERS);
            *state_term[i] = END;
        }
    }
}

```

void reset_operators(void)

/*

Copy operators from the operator_definition structure to the operator_buffer structure.

*/

{

```

void add_to_state(operator_adr, present_adr)
struct operator *operator_adr;
char *present_adr;
/*
    Add add_effects of the given operator to the given present state.
*/
{
    char *present_term[TERMS];
    register i, j;
    int match_flag;

    i = 0;
    present_term[0] = present_adr;

    while( *present_term[i] != END && i < TERMS )
    {
        j = 0;
        match_flag = OFF;
        while ( *(operator_adr->add_effect)[j] != END && match_flag == OFF )
        {
            if ( compare_terms(present_term[i], operator_adr->add_effect[j])
                == TRUE )
            {
                *present_term[i] = '\0';
                match_flag = ON;
            }
            j++;
        }
        i++;
        present_term[i] = present_adr + (i * LETTERS);
    }

    compress_state(present_adr);
    present_term[0] = present_adr;
    i = 0;
    while( *present_term[i] != END )
    {
        i++;
        present_term[i] = present_adr + (i * LETTERS);
    }
    j = 0;
    while( *(operator_adr->add_effect)[j] != END && i+j < TERMS-1 )
    {
        str_cat(present_term[i+j], operator_adr->add_effect[j]);
        j++;
        present_term[i+j] = present_adr + (i+j) * LETTERS;
    }
    *present_term[i+j] = END;
}

```

```

register i;

i = 0;
while( *(operator_definition[i].name) != END && i<OPERATORS )
{
    copy_operator(&operator_buffer[i],&operator_definition[i]);
    i++;
}
*(operator_buffer[i].name) = END;
}

```

```

void initialize_operators(present_adr, goal_adr)
char *present_adr;
char *goal_adr;
/*
    Substitute values for those preconditions and effects which take
    arguments.
*/

```

```

{
    register i;

    reset_operators();
    expand_state(present_adr);
    expand_state(goal_adr);
    i = 0;
    while( *(operator_buffer[i].name) != END && i<OPERATORS )
    {
        initialize_pos_conditions(&operator_buffer[i],present_adr,goal_adr);
        initialize_neg_conditions(&operator_buffer[i],present_adr,goal_adr);
        initialize_add_effects(&operator_buffer[i],present_adr,goal_adr);
        initialize_del_effects(&operator_buffer[i],present_adr,goal_adr);
        initialize_operator_name(&operator_buffer[i],goal_adr);
        i++;
    }
}

```

```

void substitute_terms(operator_term_adr, state_adr)
char *state_adr;
char *operator_term_adr;
/*
    Given the address of an uninitialized operator term find the
    corresponding term in the given state and substitute it for
    the first term. If no corresponding term in the state is found,
    initialize to symbol '*', i.e., a wildcard. Calls check_term_start.
*/
{
    char *state_term[TERMS];
    char string_buffer[15];
    register j;

```

```

void update_present_state(operator_adr, present_adr)
struct operator *operator_adr;
char *present_adr;
/*
    Apply the operator to the present state.
*/
{
    del_from_state(operator_adr, present_adr);
    add_to_state(operator_adr, present_adr);
}

```

```

compare_terms(string1, string2)
char *string1;
char *string2;
/*
    Compare 3 strings, return 1 if equal, 2 if different but neither
    marked as essential and 0 if different and essential. The '*'
    character is treated like in DOS.
*/
{
    int nb_flag;

    nb_flag = OFF;
    if (*string1 == '#0')
    {
        nb_flag = ON;
        string1++;
    }
    if (*string2 == '#0')
    {
        nb_flag = ON;
        string2++;
    }
    for( ; *string1 == *string2; string1++, string2++ )
    {
        if ( *string1 == '\0' )
            return(1);
    }
    if ( *string1 == '*' || *string2 == '*' ) return(1);
    if ( nb_flag == OFF ) return(2);
    else return(0);
}

```

```

void copy_term_start(string1, string2)
char *string1;
char *string2;
/*
    Copy the characters from the beginning of string2 up to the

```

```

    match_flag;

j = 0;
match_flag = OFF;
state_term[0] = state_adr;
while ( *state_term[j] != END && j < TERMS && match_flag == OFF )
{
    if ( check_term_start(operator_term_adr, state_term[j]) == TRUE )
    {
        if ( *state_term[j] == NB ) state_term[j]++;
        strcpy(operator_term_adr, state_term[j]);
        match_flag = ON;
    }
    j++;
    state_term[j] = state_adr + (j * LETTERS);
}
if (match_flag == OFF)
{
    copy_term_start(string_buffer, operator_term_adr);
    strcat(string_buffer, "=");
    strcpy(operator_term_adr, string_buffer);
}
}

```

```

initialize_operator_name(operator_adr, state_adr)
struct operator *operator_adr;
char *state_adr;
/*
    Substitute a value for argument in those operator names which take
    arguments.
*/
{
    char *state_term[TERMS], *value[LETTERS];
    register i;
    int match_flag;

    if ( check_term_end(operator_adr->name, "goal") == TRUE )
    {
        if ( check_term_start(operator_adr->name, "Set_setpoint") == TRUE )
        {
            i = 0;
            match_flag = OFF;
            state_term[0] = state_adr;
            while( *state_term[i] != END && i < TERMS && match_flag == OFF )
            {
                if ( check_term_start(state_term[i], "setpoint") == TRUE )
                {
                    copy_term_end(value, state_term[i]);
                    match_flag = ON;
                }
                i++;
            }
        }
    }
}

```

underscore into string1.

```
*/
{
    for (; *string2 != '_' || *string2 != '\0' ; string2++, string1++)
    {
        *string1 = *string2;
    }
    *string1 = '_';
    *string1++;
    *string1 = '\0';
}
}
```

void copy_term_end(string1, string2)

char *string1;

char *string2;

/*

Copy the characters from the underscore up to the end of string2 into string1.

*/

```
{
    for (; *string2 != '_' || *string2 != '\0' ; string2++ );

    if( *string2 == '_' )
    {
        string2++;
        for (; *string2 != '\0' ; string2++, string1++)
            *string1 = *string2;
    }
    *string1 = '\0';
}
}
```

check_term_end(string1, string2)

char *string1;

char *string2;

/*

Check if string2 ends string1 (i.e., at least NTH last characters of string1 are equal to string2), if yes return 1, if no return 0.

*/

```
{
    char *str1, *str2;
    register i;

    str1 = string1;
    do
    {
        str2 = string2;
        for( ; *str1 != *str2; str1++ )
        {
            if ( *str1 == '\0' || *str2 == '\0' )

```

```

        state_term[i] = state_adr + (i * LETTERS);
    }
    if ( match_flag == ON )
    {
        strcpy(operator_adr->name, "Set_setpoint_at_");
        strcat(operator_adr->name, value);
    }
}
}

initialize_neg_conditions(operator_adr, present_adr, goal_adr)
struct operator *operator_adr;
char *present_adr;
char *goal_adr;
/*
    Find all uninitialized neg_conditions of the given operator and
    initialize them by calling substitute_terms. Calls check_term_end
    and substitute_terms.
*/
{
    register i;

    i = 0;
    while( *(operator_adr->neg_condition)[i] != END && i < TERMS )
    {
        if ( check_term_end(operator_adr->neg_condition[i], "_pres") == TRUE )
            substitute_terms(operator_adr->neg_condition[i], present_adr);
        else
        {
            if (check_term_end(operator_adr->neg_condition[i], "_goal") == TRUE)
                substitute_terms(operator_adr->neg_condition[i], goal_adr);
        }
        i++;
    }
}

initialize_pos_conditions(operator_adr, present_adr, goal_adr)
struct operator *operator_adr;
char *present_adr;
char *goal_adr;
/*
    Find all uninitialized pos_conditions of the given operator and
    initialize them by calling substitute_terms. Calls check_term_end
    and substitute_terms.
*/
{
    register i;

```

```

        return(0);
    }
    for( i = 0;; *stri == *str2; stri++, str2++, i++ )
    {
        if ( *stri == '\0' && i >= MIN )
            return(1);
        if ( *stri == '\0' && i < MIN )
            return(0);
    }
    } while( *stri != '\0' );
    return(0);
}

```

check_term_start(string1, string2)

char *string1;

char *string2;

/*

Check if string2 starts string1 (i.e., at least MIN first
characters of string1 are equal to string2). If yes return 1,
if no return 0.

*/

{

register i;

if (*string1 == '\0') string1++;

if (*string2 == '\0') string2++;

for(i = 0; *string1 == *string2; string1++, string2++, i++)

{

if (i >= MIN)

return(1);

}

return(0);

}

check_neg_conditions(operator_adr, state_adr)

struct operator *operator_adr;

char *state_adr;

/*

Check if any of the neg_conditions of the operator considered
matches any of the terms of the present state. If yes return 0 to
abandon the operator, if no return 1 to carry on with the operator.

*/

{

char *state_term[TERMS];

register i,j;

i = 0;

while (*(operator_adr->neg_conditions)[i] != '\0')

```

i = 0;
while( *(operator_adr->pos_condition)[i] != END && i<TERMS )
{
    if ( check_term_end(operator_adr->pos_condition[i], "_pres") == TRUE )
        substitute_terms(operator_adr->pos_condition[i], present_adr);
    else
    {
        if ( check_term_end(operator_adr->pos_condition[i], "_goal") == TRUE )
            substitute_terms(operator_adr->pos_condition[i], goal_adr);
    }
    i++;
}

```

```

initialize_del_effects(operator_adr, present_adr, goal_adr)
struct operator *operator_adr;
char *present_adr;
char *goal_adr;
/*

```

Find all uninitialized del_effects of the given operator and initialize them by calling substitute_terms. Calls check_term_end and substitute_terms.

```

*/
{
    register i;
    i = 0;
    while( *(operator_adr->del_effect)[i] != END && i<TERMS )
    {
        if ( check_term_end(operator_adr->del_effect[i], "_pres") == TRUE )
            substitute_terms(operator_adr->del_effect[i], present_adr);
        else
        {
            if ( check_term_end(operator_adr->del_effect[i], "_goal") == TRUE )
                substitute_terms(operator_adr->del_effect[i], goal_adr);
        }
        i++;
    }
}

```

```

initialize_add_effects(operator_adr, present_adr, goal_adr)
struct operator *operator_adr;
char *present_adr;
char *goal_adr;
/*

```

Find all uninitialized add_effects of the given operator and initialize them by calling substitute_terms. Calls check_term_end and substitute_terms.

```

*/

```



```
register i;
```

```
i = 0;
```

```
while( *(operator_adr->add_effect)[i] != END && i < TERMS )
```

```
{  
    if ( check_term_end(operator_adr->add_effect[i], "_pres") == TRUE )
```

```
        substitute_terms(operator_adr->add_effect[i], present_adr);
```

```
    else
```

```
        if ( check_term_end(operator_adr->add_effect[i], "_goal") == TRUE )
```

```
            substitute_terms(operator_adr->add_effect[i], goal_adr);
```

```
    }
```

```
    i++;
```

C.4. Pl-su-fc.c

filename: PL-SU-FC.C

description: Functions used by the planner to cooperate with the supervisor.

written by: A. Korpala - 3/91

```
void make_abstract_state(present_adr)
```

```
char *present_adr;
```

```
/*
```

```
    Create an abstract state, i.e., in the form which can be processed  
    by the planner from the data contained in the state of plant  
    structure.
```

```
*/
```

```
{  
    char *present_term[TERMS];
```

```
    char value[3];
```

```
    int setpoint;
```

```
    register i;
```

```
    i = 0;
```

```
    if ( state_of_plant.large_disturbance == TRUE )
```

```
    {
```

```

check_del_effects(operator_adr, present_adr, goal_adr)
struct operator *operator_adr;
char *present_adr;
char *goal_adr;

```

Check if any of the del_effects of the operator considered is present in the present state, but absent from the goal state. if yes return 1, if no return 0.

```

*/
{
    char *present_term[TERMS];
    char *goal_term[TERMS];
    int match1_flag;
    int match2_flag;
    register i, j;

    i = 0;
    match1_flag = 0;
    match2_flag = 0;
    while ( *(operator_adr->del_effect)[i] != END )
    {
        j = 0;
        present_term[0] = present_adr;
        while ( *present_term[j] != END && match1_flag == OFF )
        {
            if ( compare_terms( operator_adr->del_effect[i], present_term[j] )
                == TRUE )
            {
                match1_flag = ON;
                j++;
                present_term[j] = present_adr + (j * LETTERS);
            }
            if ( match1_flag == ON )
            {
                j = 0;
                goal_term[0] = goal_adr;
                while ( *goal_term[j] != END && match2_flag == OFF )
                {
                    if ( compare_terms( operator_adr->del_effect[i], goal_term[j] )
                        == TRUE )
                    {
                        match2_flag = ON;
                        j++;
                        goal_term[j] = goal_adr + (j * LETTERS);
                    }
                    if ( match2_flag == OFF ) return(1);
                }
                j++;
            }
        }
        i++;
    }
    return(0);
}

```

```
present_term[i] = present_adr + (i * LETTERS);  
strcpy(present_term[i], "large_disturbance");  
i++;
```

```
if ( state_of_plant.overload == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "overload");  
    i++;
```

```
if ( state_of_plant.valve_failure == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "valve_failure");  
    i++;
```

```
if ( state_of_plant.advanced_failure == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "advanced_failure");  
    i++;
```

```
if ( state_of_plant.backup_failure == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "backup_failure");  
    i++;
```

```
if ( state_of_plant.alarm_condition == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "alarm_condition");  
    i++;
```

```
if ( state_of_plant.sensor_failure == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "sensor_failure");  
    i++;
```

```
if ( state_of_plant.fatal_failure == TRUE )
```

```
{  
    present_term[i] = present_adr + (i * LETTERS);  
    strcpy(present_term[i], "fatal_failure");  
    i++;
```

```

if ( state_of_plant.active == ADVANCED_CONTROL )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "active_advanced");
    i++;
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "supervisory_control");
    i++;
}

```

```

if ( state_of_plant.active == BACKUP_CONTROL )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "active_backup");
    i++;
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "supervisory_control");
    i++;
}

```

```

if ( state_of_plant.gain == HIGH )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "high_gain");
    i++;
}

```

```

if ( state_of_plant.integral == LOW )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "low_integral");
    i++;
}

```

```

if ( state_of_plant.derivative == HIGH )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "derivative_action");
    i++;
}

```

```

if ( task_flags.monitor_flag == ON )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "monitoring");
    i++;
}

```

```

if ( state_of_plant.active == 0 &&
    state_of_plant.large_disturbance == 0 &&
    state_of_plant.proximity == 0 &&
    state_of_plant.steady_state == 0 &&

```

```

state_of_plant.advance_failure = 0 )
{
    present_term[i] = present_adr + (i * LETTERS);
    strcpy(present_term[i], "level_0");
    i++;
}

```

```

present_term[i] = present_adr + (i * LETTERS);
setpoint = plant_variables.setpoint;
itoa(setpoint, value, 10);
strcpy(present_term[i], "setpoint_");
strcat(present_term[i], value);
i++;

```

```

present_term[i] = present_adr + (i * LETTERS);
*present_term[i] = END;
}

```

Appendix D

Additional code

This appendix includes the list of the concurrent tasks, the file containing the main function and the functions used for multitasking, and the software used for driving the A/D card.

D.1 Sup-list.c

```
/*  
-----  
filename:    SUP-LIST.C
```

```
description: This is where the elements of the circular list of tasks  
            and their order are defined. This list forms the basis of  
            the round-robin non-preemptive scheduling.  
-----  
*/
```

```
INIT("Plan", plan);  
  
INIT("Set_setpoint", set_setpoint);  
INIT("Increase_gain", increase_gain);  
INIT("Decrease_gain", decrease_gain);  
INIT("Increase_integral", increase_integral);  
INIT("Decrease_integral", decrease_integral);  
INIT("Turn_on_derivative", turn_on_derivative);  
INIT("Turn_off_derivative", turn_off_derivative);  
  
INIT("Wait_for_proximity", wait_for_proximity);  
INIT("Wait_for_steady_state", wait_for_steady_state);
```

```
INIT("Sound_alarm", sound_alarm);
INIT("Emergency_shutdown", emergency_shutdown);
```

```
INIT("Read_plant", read_plant);
INIT("Activate_advanced", activate_advanced);
INIT("Activate_backup", activate_backup);
INIT("Activate_standard_PI", activate_standard_PI);
INIT("Write_to_plant", write_to_plant);
INIT("Monitor", monitor);
```

```
INIT("Abort_all", abort_all);
```

D.2 Sup-kern.c

```
/*
```

```
file: SUP-KERN.C
```

description: This file contains the functions enabling non-preemptive scheduling of the supervisory tasks under DOS. It also contains the main function which activates the multitasking kernel, hence this is the compilable file.

This code is a modified version of the multitasking kernel written by J. Cohen.

```
*/
```

```
#include <alloc.h>
#include <mem.h>
#include <setjmp.h>
#include <dos.h>
#include <stdio.h>
#include <string.h>
#define UNWORD unsigned int
#define WORD int
#define BYTE char
#define LONG unsigned long int
#define BOOLEAN int

#define STD_STK (2*1024) /* Standard task stack size */
#define MAXTASKS 60 /* Maximum number of tasks allowed */

#define INIT(task_name, task_addr) initialize(task_name, task_addr, STD_STK)

struct rd_data
{
```

```

char name[35];
BYTE *stk_ptr;
WORD stk_sz;
void (*vp)();
jmp_buf state238;
struct nd_data *next, *prev;
};

```

```

typedef struct nd_data node_data;

```

```

node_data *cur_ptr;
int num_items;

```

```

void insert_item(node_data *item)
{
    if (num_items == MAXTASKS)
        printf("The List is Full.\n");
    if (current == NULL)
    {
        item->next = item;
        item->prev = item;
        current = item;
    }
    else
    {
        item->next = current->next;
        current->next = item;
        item->prev = current;
        item->next->prev = item;
    }
    num_items++;
    current=current->next;
}

```

```

void initialize(char *name, void (*taskname)(void), WORD stk_size)
{
    BYTE *stk;
    BYTE *sp;
    node_data *taskitem;

    taskitem = (node_data *) malloc(sizeof(node_data));
    if (!taskitem) printf("Cannot initialize new task.\n");
    stk = (BYTE *) malloc(stk_size);
    if (!stk) printf("Cannot allocate new stack.\n");

    taskitem->fp = taskname;
    taskitem->stk_ptr = stk;
    taskitem->stk_sz = stk_size;
    strcpy(taskitem->name, name);
}

```

```

    sp = stk_size + stack - 1;
    setjmp(taskitem->state286);
    taskitem->state286->j_ip = FP_OFF(taskname);
    taskitem->state286->j_cs = FP_SEG(taskname);
    taskitem->state286->j_bp = FP_OFF(sp);
    taskitem->state286->j_sp = FP_OFF(sp);
    taskitem->state286->j_es = FP_SEG(sp);

    insert_item(taskitem);
}

```

```

void yield(void)
{
    if (!setjmp(current->state286))
    {
        current = current->next;
        longjmp(current->state286, 1);
    }
}

```

```

void system_go(node_data *current)
{
    if (current)
        longjmp(current->state286, 1);
}

```

```

#include "sup-defs.c"
#include "sup-acts.c"
#include "sup-plan.c"

```

```

void main()
{
    current = NULL;
    clrscr();
    num_items = 0;
    #include "sup-list.c"
    current = current->next;
    system_go(current);
}

```

D.3 Sup-adac.c

```
/*
```

Filename : SUP-ADAC.C

Description: This is a slightly modified version of a program designed for interfacing with an Analog-to-Digital and Digital-to-Analog converter. The heading of the original code is retained below.

*/

/*

ADBOARD.C

This software is used to drive the DT2801-A DAC/ADC card.

Written by: Shana Zagnoon
Date : 30 April 1990
University of the Witwatersrand

*/

/*

/* Definition of commands and card registers */

```
#define DATAREGISTER 0x2EC /* Data register address */
#define COMMANDREGISTER 0x2ED /* Command register address */
#define STATUSREGISTER 0x2ED /* Status register address */
#define CCLEAR 0x1 /* Clear error */
#define WAITWAIT 0x2 /* Status register check */
#define CCLOCK 0x3 /* Set clock period */
#define COMMANDWAIT 0x4 /* Status register check */
#define READWAIT 0x5 /* Status register check */
#define CDOUT 0x8 /* Write D/A immediate */
#define CSDA 0x9 /* Set D/A parameters */
#define CSAD 0x10 /* Set A/D parameters */
#define CRAD 0x11 /* Read A/D */
#define CSTOP 0x1F /* Aborts present command */
#define DACSELECT 0 /* Selects DAC channel 0 */
```

/*

/* PROCEDURE CHECK - This procedure is used to ensure that a parameter chosen for the card falls between the limits that are acceptable by the card. If necessary, the value is adjusted to comply with the limits and is returned to the calling procedure. */

```
check(value, lower, upper, mess)
int value, lower, upper;
char mess;
```

{

```
    if (value < lower)
```

```
    {
```

```

printf("Invalid %s = %d - %s set to lower limit\n",
      mess, value, mess);
value = lower;
}
if (value > upper)
{
printf("Invalid %s = %d - %s set to upper limit\n",
      mess, value, mess);
value = upper;
}
return(value);
}

/* PROCEDURE LOWER - This procedure checks the LIMIT command
   and if it is found to be valid, the lower limit is set
   in the command register. */

void lower(char value)
{
  unsigned char lower;
  int i;

  while (i < 16)
  {
    if (value[i] == 'x')
    {
      /* This is the bit to set */
      lower |= 1 << i;
    }
    i++;
  }

  /* This procedure outputs a command
   to the command register. */

  void outputtoboard(value)
  unsigned char value;

  {
    pause(STATUSREGISTER, COMMAND, 0);
    outputtoboard(COMMANDREGISTER, value);
  }
}

/* PROCEDURE DATATOBARD - This procedure is used to output
   data to the data register. */

void datatoboard(value)
unsigned char value;

```

```

    pause(STATUSREGISTER, WRITEWAIT, WRITEWAIT);
    outportb(DATARREGISTER, value);
}

```

```

/* ----- */

/* PROCEDURE DATAFROMBOARD - This procedure is called to
read data from the data register */

```

```

datafromboard()

```

```

{
    int boarddata;

    pause(STATUSREGISTER, READWAIT, READWAIT);
    boarddata = inportb(DATARREGISTER);
    return(boarddata);
}

```

```

/* ----- */

/* PROCEDURE STOPANDCLEAR - This procedure is used to terminate
the present operation of the card and to clear all the registers */

```

```

void stopandclear()

```

```

{
    unsigned char temp;

    outport(COMMANDREGISTER, CSTOP);
    temp = inportb(DATARREGISTER);
    commandtoboard(CCLEAR);
}

```

```

/* ----- */

/* PROCEDURE HIGHLOW - This procedure divides a number into a
high and low byte. */

```

```

highlow(number, high, low)
long int number;
unsigned char *high, *low;

```

```

{
    *high = number/256;
    *low = number - (*high*256);
}

```

```

/* ----- */

/* PROCEDURE SETCLOCKPERIOD - This procedure is used to determine
the sampling period of the card in multiples of

```

2.5 microseconds. */

```
void setclockperiod(period)
long int period;
```

```
{
    unsigned char lowbyte, highbyte;
```

```
    highlow(period, &highbyte, &lowbyte);
```

```
    commandtoboard(CLOCK);
```

```
    datatoboard(lowbyte);
```

```
    datatoboard(highbyte);
```

```
    pause(STATUSREGISTER, COMMANDWAIT, 0);
```

```
    if ((inportb(STATUSREGISTER) & 0x80) != 0)
```

```
    {
```

```
        printf("\nError setting clock");
```

```
        exit(1);
```

```
    }
```

/* ----- */

/* PROCEDURE PROGRAMAD - This procedure is used to configure the A/D subsystem to perform multiple conversions. Setting the A/D parameters does not perform any A/D conversions but is a necessary prerequisite to the conversion operation. */

```
void programad(gain, startchannel, endchannel, count)
```

```
unsigned char gain, startchannel, endchannel;
```

```
int count;
```

```
{
```

```
    unsigned char lowcount, highcount;
```

```
    check(gain, 0, 1, "gain code");
```

```
    check(startchannel, 0, 15, "start channel");
```

```
    check(endchannel, 0, 15, "end channel");
```

```
    check(count, 3, 32767, "sample count");
```

```
    commandtoboard(CSAD);
```

```
    datatoboard(gain);
```

```
    datatoboard(startchannel);
```

```
    datatoboard(endchannel);
```

```
    highlow(count, &highcount, &lowcount);
```

```
    datatoboard(lowcount); /*Does nothing if DMA used*/
```

```
    datatoboard(highcount);
```

/* ----- */

/* PROCEDURE PROGRAMDA - This procedure is used to set up the D/A subsystem to perform multiple conversions. Setting the D/A parameters does not perform any conversions but is a necessary prerequisite to performing a digital to analog

conversion */

```
void programda(dacselect, count)
unsigned char  dacselect;
long int      count;
```

```
{
    unsigned char  lowcount, highcount;
```

```
    check(dacselect, 0, 2, "DAC select");
    commandtoboard(CSDA);
    datatoboard(dacselect);
    highlow(count, &highcount, &lowcount);
    datatoboard(lowcount);
    datatoboard(highcount);
}
```

/* ----- */

/*PROCEDURE WRITETOCARD - This procedure performs single
D/A conversion on the P/A channel selected. */

```
void writetocard(analogoutput)
double analogoutput;
```

```
{
    unsigned char  status, lowvalue, highvalue;
    long int      da_value;
```

```
    stopandclear();
    da_value = (long int)((analogoutput*10) * 4096 / 20);
    highlow(da_value, &highvalue, &lowvalue);
```

```
    commandtoboard(CDAOUT);
    check(DACSELECT, 0, 2, "DAC select");
    datatoboard(DACSELECT);
    datatoboard(lowvalue);
    datatoboard(highvalue);
    status = inportb(STATUSREGISTER);
    if ((status & 0x80) > 0)
    {
        printf("ERROR");
        exit(1);
    }
}
```

/* ----- */

/* PROCEDURE READCARD - This procedure initiates a scan of
clocked A/D conversions of the channels specified in a
SET A/D PARAMETERS command. The output is written into a
data structure which stores the information. */

```
readcard(gain, startchannel, endchannel);
unsigned char gain, startchannel, endchannel;
```

```
register i;
double factor;
long int reading[3];
int count, nchan, adl[9], adh[9];
```

```
stopandclear();
setclockperiod(200);
nchan = endchannel - startchannel + 1;
count = 3;
if(nchan > count) /* Ensures that at least three reads occur */
    count = nchan;
programed(gain, startchannel, endchannel, count);
commandtoboard(CMD);
for (i = 0; i <= count-1; ++i) {
    pause(STATUSREGISTER, READWAIT, 0);
    adl[i] = rtrb(DATAREGISTER);
    pause(STATUSREGISTER, READWAIT, 0);
    adh[i] = rtrb(DATAREGISTER);
    reading[i] = adl[i] + adh[i]*256;
    factor = (10.0/4095)/(gain+1);
}
plant_variables.level = 2 * (reading[0] * factor) - (10 / (gain+1));
plant_variables.flowrate = 2 * (reading[1] * factor) - (10 / (gain+1));
}
```

References

- [Agre and Chapman, 87] P. E. Agre and D. Chapman. Pengi: An Implementation of a Theory of Activity. *Proceedings of the 6th National Conference on Artificial Intelligence*, Seattle, WA, USA, July 1987, pp. 268-272.
- [Allen, 84] J. F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23, 1984, pp. 123-154.
- [Allen and Koomen, 88] J. F. Allen and J. A. Koomen. Planning Using a Temporal World Model. *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Karlsruhe, FRG, August 1983, pp. 741-747.
- [Alterman, 86] R. Alterman. An Adaptive Planner. *Proceedings of the 5th National Conference on Artificial Intelligence*, Philadelphia, Pa, USA, August 1986, pp. 65-69.
- [Alty, 87] J. L. Alty. The Limitations of Rule Based Expert Systems, in *Knowledge-Based Expert Systems in Industry* (ed. J. Kriz), Ellis Horwood, Chichester, West Sussex, UK, 1987.
- [Andert and Frasher, 88] E. Andert Jr. and W. Frasher. Compact Expert System for Planning and Control Applications, in *Artificial Intelligence in Engineering: Robotics and Processes* (ed. J. S. Gero), Elsevier, Amsterdam, The Netherlands, 1988.
- [Åström et al., 89] K. J. Åström, C. C. Hang and P. Persson. Towards Intelligent PID Control. *2nd IFAC Workshop on Artificial Intelligence in Real-Time Control*, PRC, Sept. 1989, pp. 38-43.
- [Barr and Feigenbaum, 82] A. Barr and E. A. Feigenbaum (eds.). *The Handbook of Artificial Intelligence*, vol. 1-2, W. Kaufmann, Los Altos, 1982.
- [Bobrow et al., 88] D. G. Bobrow, S. Mittal and M. J. Stefik. Expert Systems: Perils and Promise, in *Expert Systems in Engineering* (ed. D. T. Pham), IFS Publications, Kempston, Bedford, UK, 1988.

- [Carbonell, 89] J. G. Carbonell. Introduction: Paradigms for Machine Learning. *Artificial Intelligence*, vol. 40, 1989, pp. 1-9.
- [Chapman, 85] D. Chapman. Nonlinear Planning: A Rigorous Reconstruction. *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Los Angeles, California, USA, August 1985, pp. 1022-1024.
- [Chen and Bastani, 88] I-R. Chen and F. B. Bastani. Knowledge Representation, Planning and Learning in Fault Tolerant Process-Control Systems, in *Artificial Intelligence in Engineering: Robotics and Processes* (ed. J. S. Gero). Elsevier, Amsterdam, The Netherlands, 1988.
- [Cohen and Feigenbaum, 82] P. R. Cohen and E. A. Feigenbaum (eds.). *The Handbook of Artificial Intelligence*, vol. 3, W. Kaufmann, Los Altos, 1982.
- [Dean and Boddy, 88] T. Dean and M. Boddy. An Analysis of Time-Dependent Planning. *Proceedings of the 7th National Conference on Artificial Intelligence*, St. Paul, MN, USA, August 1988, pp. 49-54.
- [Doyle et al., 86] R. J. Doyle, D. J. Atkinson and R. S. Doshi. Generating Perception Requests and Expectations to Verify the Execution of Plans. *Proceedings of the 5th National Conference on Artificial Intelligence*, Philadelphia, Pa, USA, August 1986, pp. 81-88.
- [Edgar and Himmelblau, 88] T. F. Edgar and D. M. Himmelblau. *Optimization of Chemical Processes*. McGraw-Hill, USA, 1988.
- [Fikes and Nilsson, 71] R. E. Fikes and N. S. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, vol. 2, 1971, pp. 189-208.
- [Firby, 87] R. J. Firby. An Investigation into Reactive Planning in Complex Domains. *Proceedings of the 6th National Conference on Artificial Intelligence*, Seattle, WA, USA, July 1987, pp. 202-206.
- [Georgeff and Lansky, 87] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. *Proceedings of the 5th National Conference on Artificial Intelligence*, Seattle, WA, USA, July 1987, pp. 667-682.
- [Handelman et al., 88] D. A. Handelman, S. H. Lane and J. J. Gelfand. Integration of Knowledge-Based Systems and Neural Network Techniques for Robotic Control. *1st IFAC Workshop on Artificial Intelligence in Real-Time Control*, Swansea, UK, Sept. 1988.

- [Harbison and Steele, 84] S. P. Harbison and G. L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1984.
- [Harmon and King, 85] P. Harmon and D. King. *Expert Systems*. John Wiley & Sons, NY, USA, 1985.
- [Ishida, 88] Y. Ishida. Integrating Model-Based and Syndrome-Based Processing Plant Fault Diagnosis: A Qualitative Approach to Process Diagnosis. *Engineering Applications of Artificial Intelligence*, vol. 1, no. 1, March 1988, pp. 11-15.
- [Johnson, 67] E. F. Johnson. *Automatic Process Control*. McGraw-Hill, USA, 1967.
- [Kaelbling, 88] L. P. Kaelbling. Goals as Parallel Program Specifications. *Proceedings of the 7th National Conference on Artificial Intelligence*, St. Paul, MN, USA, August 1988, pp. 60-65.
- [Kernighan and Ritchie, 78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1978.
- [van Koppen, 88] J. van Koppen. A Survey of Expert System Development Tools, in *Expert Systems in Engineering* (ed. D. T. Pham). IFE Publications, Kempston, Bedford, UK, 1988.
- [Krijgsman et al., 88] A. J. Krijgsman, H. B. Verbruggen and P. M. Bruijn. Knowledge-Based Real-Time Control. *1st IFAC Workshop on Artificial Intelligence in Real-Time Control*, Swansea, UK, Sept. 1988, pp. 7-13.
- [Kuipers, 86] B. Kuipers. Qualitative Simulation. *Artificial Intelligence*, vol. 29, 1986, pp. 289-338.
- [Lamport, 86] L. Lamport. *TEX: A Document Preparation System*. Addison-Wesley, USA, 1986.
- [Lun and MacLeod, 90] V. Lun and I. M. MacLeod. *The Challenge of Intelligent Computing*. Internal Report, Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, RSA, June 1990.
- [Lun et al., 90] V. Lun, I. M. MacLeod and J. E. Cohen. *The Development of a Microcomputer-Based Non-Preemptive System Executive*. Internal Report, Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, RSA, 1990.
- [Pao, 88] Y-H. Pao. Engineering Artificial Intelligence. *Engineering Applications of Artificial Intelligence*, vol. 1, no. 1, March 1988, pp. 5-10.

- [Pham and Pham, 88] D. T. Pham and P. T. N. Pham. Expert Systems: A Review, in *Expert Systems in Engineering* (ed. D. T. Pham), IFS Publications, Kempston, Bedford, UK, 1988.
- [Sacerdoti, 74] E. D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, vol. 5, 1974, pp. 115-135.
- [Sacerdoti, 79] E. D. Sacerdoti. Problem Solving Tactics. *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, Tokyo, Japan, August 1979, pp. 1077-1085.
- [Savas, 65] E. S. Savas. *Computer Control of Industrial Processes*. McGraw-Hill, USA, 1965.
- [Shadbolt et al., 89] N. R. Shadbolt, R. K. Stobart and B. D. Robinson. A Framework for Knowledge Based Process Control. *2nd IFAC Workshop on Artificial Intelligence in Real-Time Control*, PRC, Sept. 1989, pp. 148-154.
- [Schildt, 88] H. Schildt. *Turbo C: The Complete Reference*. Borland-Osborne and McGraw-Hill, USA, 1988.
- [Simmonds, 88] W. H. Simmonds. Representation of Real Knowledge for Real Time Use. *1st IFAC Workshop on Artificial Intelligence in Real-Time Control*, Swansea, UK, Sept. 1988, pp. 63-68.
- [Smuts, 90] W. B. Smuts. *Supervisory Control for an Autonomous Vehicle*. PhD Thesis, University of the Witwatersrand, Johannesburg, RSA, March 1990.
- [Sripada et al., 88] N. R. Sripada, D. G. Fisher and A. J. Morris. Application for Process Regulation and Servo Control, in *Expert Systems in Engineering* (ed. D. T. Pham), IFS Publications, Kempston, Bedford, UK, 1988.
- [Sussman, 75] G. J. Sussman. *A Computer Model of Skill Acquisition*. Elsevier, New York, NY, USA, 1975.
- [Takahashi et al., 72] Y. Takahashi, M. J. Rabins and D. M. Auslander. *Control and Dynamic Systems*. Addison-Wesley, Massachusetts, USA, 1972.
- [Tate, 87] A. Tate. Applications of Knowledge-Based Planning Systems, in *Knowledge Based Expert Systems in Industry* (ed. J. Kriz), Ellis Horwood, Chichester, West Sussex, UK, 1987.
- [Taunton and Haspel, 88] J. C. Taunton and D. W. Haspel. The Application of Expert System Techniques in On-Line Process Control, in *Expert Systems in Engineering* (ed. D. T. Pham), IFS Publications, Kempston, Bedford, UK, 1988.

- [Tenenberg, 86] J. Tenenberg. Planning With Abstraction. *Proceedings of the 5th National Conference on Artificial Intelligence*, Philadelphia, Pa, USA, August 1986, pp. 76-80.
- [Tsai et al., 86] T. H. Tsai, J. W. Lane and C. S. Lin. *Modern Control Techniques for the Processing Industries*. Marcel Dekker, New York, NY, USA, 1986.
- [Verbruggen and Åström, 89] H. B. Verbruggen and K. J. Åström. Artificial Intelligence and Feedback Control. *2nd IFAC Workshop on Artificial Intelligence in Real-Time Control*, PRC, Sept. 1989, pp. 115-125.
- [Vere, 85] S. Vere. Temporal Scope of Assertions and Window Cutoff. *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Los Angeles, California, USA, August 1985, pp. 1055-1059.
- [Voss, 88] H. Voss. Architectural Issues for Expert Systems in Real-Time Control. *1st IFAC Workshop on Artificial Intelligence in Real-Time Control*, UK, Sept. 1988, pp. 1-6.
- [Wilkins, 83] D. E. Wilkins. Representation in a Domain Independent Planner. *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, Karlsruhe, FRG, August 1983, pp. 733-740.
- [Zagnoev, 91] S. Zagnoev. *The Development of a Modular Real-Time Process Control Language*. MSc Dissertation, University of the Witwatersrand, Johannesburg, RSA, 1991.

Author: Korpala Andrzej.

Name of thesis: The Development And Evaluation Of An Intelligent Supervisory System For Process Control.

PUBLISHER:

University of the Witwatersrand, Johannesburg

©2015

LEGALNOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.