COSC460

Honours Project Report

Department of Computer Science

University of Canterbury

The Application of Prolog to Real-Time Process Control

Supervisor: Dr. W Kreutzer

Paul Manson

October 1985

Contents

1. Introduction 1

2. The Problem 2

(A). Real-Time Process Control 2

(B). The Effluent Plant 2

(C). Alarm Diagnosis 4

(D). Enter Prolog 5

(E). Performance Constraints Imposed by the Plant 6

3. Theory (What I thought about) 8

(A). Prolog, a brief introduction 8

(B). Modelling in Prolog 12

(C). Configuration 14

(D). Expert System 17

(E). Concurrent, Real-Time Prolog 18

4. Practice (What I actually did) 29

(A). Modelling the Effluent 29

(B). A Prolog Interpreter 33

(C). Effluent Plant Configuration using Prolog 35

(D). Implementing CRT Prolog in Modula-2 38

5. Conclusions 49

6. References 50

7. Appendices

1. Introduction.

This project involved examining the application of the Prolog programming language to a real-time process control environment. I have specifically applied Prolog to certain aspects of a control system for an automated effluent treatment plant that has been constructed for the Canterbury Frozen Meat Company Ltd (CFM).

My aims were to determine whether or not Prolog has a contribution to make in the process control environment, and what difficulties are likely to arise in this area of application.

2. The Problem.

(A). Real-Time Process Control.

A typical real-time process control system must be capable of overseeing the concurrent operation of several physical processes. These may be either multiprogrammed (sharing a single processor), multiprocessed (one processor per physical process), or a combination of the two. My project is concerned only with the former case, as distributed process control systems are still very scarce, especially in New Zealand. Until relatively recently, real-time process control has been the domain of assembly language and Fortran (with isolated use of RTL/2, Pearl, etc.) but the advent of small, efficient high-level systems programming languages (such as C and Modula-2), coupled with an emphasis on reliable software rather than sheer speed of execution, has caused higher level languages and tools to be applied to this field.....but Prolog ?.

(B). The Effluent Plant.

During the 1984/1985 summer vacation I was employed as a systems programmer (slave labour) by Technology Solutions Ltd. of Christchurch. Their major project at the time was the writing of control software for the CFM effluent plant at Belfast, Chch.

The Effluent treatment plant is designed to process the freezing works' effluent from the company's Belfast and Canterbury works. The screened effluent is first acidified by addition of concentrated sulphuric acid, and then made alkaline by adding lime in two mixing stages. This process assists the coagulation of the dissolved solids in the effluent. The solids are then removed by passing the effluent through a dissolved air flotation (DAF) tank. The solids (up to 140 wet tons per day) are returned to the

Works' rendering plant for conversion to tallow and stock meal. The processed effluent is discharged into the Waimakariri River after settling in a holding pond.

The plant is designed for flows of between 100 and 1100 cubic meters per hour, but will normally operate at between 200 and 900 m^3/hour. It was designed from the outset to be fully computer-automated, and thus contains redundant equipment to facilitate operation despite the possible failure of one (or several) of the plant's components.

The plant's operation is overseen by a shift engineer via a large colour graphical display called a Tesselator (from ASEA, Sweden). The state of the plant is displayed on the Tesselator as a sequence of "Mimic Diagrams", each of which represents a part of the plant (using icons for pumps, valves, etc) and can be "animated" by flashing of lights and colour. The shift engineer manipulates components of the plant using the Tesselator keyboard. In general, the shift engineer need only make "physical contact" with the plant in "crisis" situations, and for maintenance purposes.

At a low level, the plant is controlled by a PLC (Programmable Logic Controller) from ASEA. The PLC monitors the state of the plant via several hundred digital and analogue measuring devices, and uses the information thus gathered to manipulate the various pumps, valves and meters in the plant.

At a higher level, a Digital Equipment MicroPDP-11 is responsible for data collection and archiving, driving the graphical display, configuration of the system, and other duties. The PLC supplies the PDP with a "censored" account of the events occurring in the plant, and the PDP uses this information to keep Tesselator Mimic diagrams updated, and for logging purposes. The PDP runs the RSX operating system, and the plant control software was written by TS entirely in Modula-2.

(C). Alarm diagnosis.

The effluent plant, like all computer systems, is subject to the laws of entropy and Murphy, and hence things go wrong. The PLC interprets these plant situations as Alarm conditions, meaning that something out-of-the ordinary has occurred. There are 400-500 possible alarms varying in importance and complexity.

Many alarms have straightforward solutions, and are handled by the PLC itself. For example, when the PLC attempts to start a pump with a faulty motor, the alarm "pump inoperable" may be raised. This alarm can usually by handled by using another pump (due to redundancy of equipment). The PLC will perform this substitution, and inform the PDP of the alarm and the steps taken to remedy the situation.

More complex alarms are immediately passed on to the PDP, and many of these are able to be handled at this point. Around 15-20% of the possible alarms, however, are sufficiently complex that they are not currently handled automatically. Such alarms are often interrelated, or are the result of failures further up the system, and it is difficult to diagnose either their cause or solution using conventional (static) programming techniques and languages. These alarms are identified by the PDP, and immediately displayed on the Tesselator screen. It is then the responsibility of the shift engineer to determine the cause of the alarm, and remedy the situation.

This state of affairs requires a shift engineer to be on duty whenever the plant is in operation, which is generally around the clock. As shift engineers are typically "imported" from Europe or the U.S.A, and command a salary of over $40,000 CFM are anxious to avoid having to employ three or four of them. They desire that the effluent plant control system be able to diagnose and respond to a sufficient proportion of the possible alarms that only one duty shift engineer is necessary, and that during the day. The

plant could then be left unsupervised during the night, and in the event that some condition did arise at night which the alarm diagnosis system could not remedy, the system could be shut down until the engineer arrive the next morning. For this to be acceptable, the probability of an such a situation must be sufficiently small that this does not happen "often". As the PDP is already over-utilised, it is envisaged that a microprocessor be connected to the system and be responsible solely for complex alarm diagnosis tasks.

(D). Enter Prolog.

In February 1985, Technology Solutions (TS) proposed that research into the application of non-conventional ("artificial intelligence") languages to the alarm diagnosis problem be carried out, and the most promising languages appeared to be Lisp and Prolog.

CFM suggested at this point that an expert system capability would be a desirable feature, providing an accessible record of the plant's event, alarm and maintainance history. This facility could aid "intelligent" diagnosis of alarms as well as being an invaluable means of retaining experience with the plant (despite turnover of personnel).

There is a significant amount of research underway, largely in the U.S.A, related to expert systems and also process control using Lisp. Because of the current interest in Prolog for artificial intelligence applications and because of the Japanese "Fifth Generation Project" which is aimed at implementing a Prolog-like language efficiently at the machine level, TS chose to look at the application of Prolog to the above problems.

This decision made, TS felt that Prolog could also be advantageously applied to plant configuration tasks. Each component of a system can usually exist in one of several states, for example a pump may by "in service", "standby" or "out of service". A configuration is a specification of

the state of each component in the system, and this configuration must comply with certain configuration rules to be acceptable. The effluent plant configuration is currently implemented in Modula-2 on the PDP, but the TS programmer responsible for writing the configuration software struggled with the problem for several months, and eventually wrote a very simple interpreter for rules similar to those in Prolog. The difficulties of writing such a rule-based program in Modula-2 (and keeping it general enough to be re-usable) prompted interest in applying Prolog to this area.

(E). Performance Constraints Imposed by the plant.

The use of a language such as Prolog pre-supposes that no rigid time constraints must be met, as current machine architectures are not well suited to Prolog execution. Although some quite efficient Prolog interpreters and compilers exist, their performance cannot compare with that of a compiled procedural language such as Modula-2.

Fortunately the effluent plant can be categorised as a "slow" system, in the sense that most physical operations require considerable real time. For example, even though the pumps used have a throughput of one million litres of effluent per hour, it still requires 30 minutes to empty a 500,000 litre pond. Thus whether or not Prolog is accepted as a "good" language to use in this particular application will depend not primarily upon its efficiency of execution, but rather on its convenience and elegance from the programmer's viewpoint.

Returning to the general field of process control, obviously Prolog's execution characteristics will not be adequate in some applications that are time-critical (eg. NASA would not be pleased if a space shuttle crashed due to the Prolog flight controller performing garbage-collection at the wrong moment !!). However if the claims of the Japanese concerning the

performance of their fifth generation machines are to be believed, there may soon exist "Prolog Engines" (in the tradition of the Lisp machines), and these could make Prolog a much more viable proposition.

[For application specifically to the CFM effluent plant, this examination of Prolog assumes an underlying system written in Modula-2, although any similar language supporting concurrency and low-level machine access would suffice].

3. Theory (What I thought about).

(A). Prolog, a brief introduction.

Prolog is a programming language based on the horn-clause dialect of first-order predicate logic. It was developed at the University of Marseilles by Alain Colmerauer in 1975, and was initially employed in natural language recognition and symbolic integration. Since then Prolog has been used in several symbol processing applications, with varying degrees of success. It is usually implemented as an interpretive, interactive language, but several compilers are now also available, offering quite efficient execution of some aspects of a Prolog program.

(1). Syntax of Prolog.

There is no accepted standard for the syntax of Prolog, but the following outlines the general features of the language, using the syntax of the most common implementations.

A Prolog database (or "program") is a set of predicates (or boolean functions), where each predicate comprises a number of clauses. Each predicate can be classified by its arity, which is the number of parameters it takes. A clause begins with a head and ends with a body, the two being separated by the symbol ":-". A head comprises the predicate name, followed by its parameter list. A body consists of a number (>= 0) of goals (or function calls) which impose conditions for the head to be true. If a clause has no body, it is called a fact, otherwise it is called a rule.

All simple or compound data objects in Prolog are called terms. A term is either a variable (distinguished by an initial capital letter), an atom (some symbolic or numeric constant), a compound term (function call), or a list. A term of the form [H|T] represents a list with head H and tail T. (In

Lisp terminology, (car [H|T]) = H and (cdr [H|T]) = T). The empty list is denoted by [ ], and a list of exactly two elements by [A,B].

For example (a classic Prolog example !), the following is a predicate with two clauses and an arity of three. The first clause is a fact (has an empty body), and the second a rule whose body consists of a single goal, namely a self-recursive call.

append([ ], X, X).

append([A|B], C, [A|D]) :- append(B,C,D).

(2). Semantics of Prolog.

There are two quite different interpretations of the semantics of a Prolog program.

a). Declarative (or "denotational") semantics, which Prolog inherits from logic, defines (recursively) the set of terms which are asserted to be true according to a program. A term is true if it is the head of some clause instance, and each of the goals of that clause instance is true, where each instance of a clause is obtained by substituting for each of its variables some term, for all occurrences of that variable. A term that is a fact is always true. The declarative interpretation of the append predicate above would be

"A list X is the result of appending [ ] (nil) to X, and

A list with head A and tail D is the result of appending a list C to a list with head A and tail B, where D is the result of appending C to B".

b). Procedural (or "operational") semantics, describes the sequence of states generated by execution of a "logic" program (in the more familiar sense of programming language semantics). The procedural interpretation of the append predicate would be

"To append [ ] (nil) to a list X, do nothing (result is X), and

To append a list C to a list with head A and tail B, append C to B".

The basic computational mechanism of Prolog is a pattern matching process called unification. This involves assigning values to variables in such a way as to cause two patterns to match. To execute a goal, the system searches for the first clause in the database whose head matches the goal, or can be unified with the goal. If at any time the system fails to find a match for a goal, it backtracks, rejecting the most recently activated clause, reconsidering the original goal which activated the rejected clause, and tries to find a subsequent clause that also matches the goal. The effect of this execution scheme is that of a left-to-right depth-first search through all solutions to a predicate or program.

Prolog provides one other mechanism for specifying control information, namely the "cut" symbol (denoted "!"). It is inserted into a clause body just like a goal, but should be ignored as far as the declarative semantics are concerned. The cut succeeds the first time it is executed, but if backtracking ever reaches it, the effect is to fail the goal which caused the clause containing the cut to be activated. In other words, the cut operation commits the system to all choices made since execution of the goal activating the clause began.

To illustrate the effect of the cut, examine:

honoursStudent(X) :- tired(X), poor(X).

honoursStudent(X) :- tired(X), poor(X), !.

the first clause would find all of the honours students in the database, but the second would find only the first honours student in the database, and then fail upon backtracking.

The conjunction of goals in a Prolog clause may be viewed as a conjunction of boolean functions by logical ANDs. Most Prolog systems also allow optional goals in a clause (using the ";" symbol). For example,

Clause :- Goal1 ; Goal2.

specifies that Clause succeeds if either Goal1 succeeds **OR** Goal2 succeeds.

(3). Applications of Prolog

When provided with a (static) database of facts and rules, Prolog can be asked to ascertain whether a certain condition is true for this "mini-world", and to return all data objects for which the condition holds. Thus Prolog may be described as a mechanical theorem prover, making inferences from the axioms (facts and rules) in its database. This view of Prolog is based on the resolution principle discovered by J. Alan Robinson in the 1960's, which takes account only of the declarative semantics of the language. Prolog has recently been much-vaunted as a language for "programming in logic", with all the associated benefits of such a strong underlying mathematical system.

Prolog has, however, been applied to many tasks that are far-removed from theorem proving, and to facilitate application to such areas, most Prolog systems provide a library of built-in "predicates". In contradiction to the independence laws of logic, some of these "predicates" (to use the word in the most colloquial sense) perform I/O operations, alter the database (adding or removing clauses), remove backtrack points, etc.

In particular, the application of Prolog to any situation where the database is expected to change over time (such as expert systems, or a dynamic model of an effluent plant) is in contradiction with the predicate logic basis underlying Prolog, which assumes the mathematical system (or database) to be **closed** (with no outside interference, and no alterations). In logic, each fact or rule states a truth independent of other facts and rules, yet in Prolog our set of "axioms" may be different at varying points in a "proof".

Despite these inconsistencies, Prolog is still a usable language with a foundation in classical logic. For the purposes of this report it is treated as a language providing a (relational) database and a very flexible query facility, rather than a language for logic programming.

(B). Modelling In Prolog.

In investigating Prolog's suitability for process control applications, the first criterion is that a process control plant be representable in a form that Prolog can easily manipulate.

(1). Static Model.

The most intuitive method is to represent each entity ("component" or "icon") in the system by a Prolog fact. This makes good use of Prolog's declarative features, and allows a very convenient reading of the plant's status at any point. Every component of the effluent plant (pump, valve, tank, etc) has an identifying name (mnemonic of its purpose) which is unique over the plant. As Prolog allows completely arbitrary data structures, it is quite natural to "declare" components of a plant as (compound) facts, for instance, a pump can be identified by its name, and it is convenient to store other information such as its type, wattage, capacity and status.

pump(p63, fixed, on, 2.2, 9.0).

represents a pump called p63 that has a fixed speed, is switched on, draws 2.2 Watts, and can pump 9.0 cubic meters per second.

Relationships between components of the plant can also be represented by facts, for example

connect(p63, tank2, 1000).

specifies that pump p63 is connected to tank2 (and the connecting pipe has a capacity of 1000 cubic meters per hour). In the context of an effluent plant, it is convenient to think of this as a directed connection, with flow from the pump to the tank, but Prolog leaves this entirely to the user's interpretation.

(2). Animating the Model.

Most Prolog systems provide "predicates" which (as side effects) add clauses to (**assert**) and delete clauses from (**retract**) the database. These predicates may be used to "animate" a static model, by providing the database with the ability to change over time.

For example, to represent the transition of a valve in a plant from closed to open, it is necessary to retract a fact which states that the particular valve is closed, and then assert a new fact stating that the valve is open, viz.

```
openValve(V) :- retract( valve(V,closed) ),
                assert( valve(V,open) ).
```

which is a little clumsy, but nonetheless quite acceptable. (The verbosity of this operation is due in main to the unnatural inclusion in Prolog of dynamic databases).

[An alternative representation of the Plant is via lists and instantiated variables, and although this allows more flexibility in dynamic databases, it is considerably more complex and much less intuitive, as well as being more volatile in the face of execution errors].

(C). Configuration.

(1). Plant Configuration.

One task for which Prolog appears well suited is that of providing a general mechanism for generating and verifying plant configurations. For the effluent plant, CFM stated a large set of requirements (or "rules") concerning its configuration which must be satisfied for it to be operational.

For example, for the plant to run, three of the four reaction tanks must be usable, and designated "Acid Tank", "Lime Tank 1" and "Lime Tank 2". The fourth tank may be (in order of preference) "standby", "out of service" or "bypassed". Chemical considerations dictate strict adherance to rules concerning which states certain Tanks may take, and for practical reasons, CFM also have (not so strict) preferences stating the most desirable configuration(s).

The Plant is configured dynamically on the Tesselator by the shift engineer, and configuration may be achieved in one of several ways.

(a). Simply ask for a configuration to be generated,

(b). Specify that a certain tank be taken "out of service" (for example, for cleaning purposes), and the system should generate the most desirable valid configuration around this.

(c). Specify all tank-state pairs (and the system merely verifies the validity of this configuration).

Because of its pattern-matching (unification) facilities, Prolog can either generate a solution, or validate one very easily. A very concise solution to the effluent plant configuration problem is given in the "Practice" section, and the effort required was minimal compared to that of programming the equivalent in Modula-2.

(2). Model Design.

The ability to dynamically configure an existing plant from a computer terminal is a special case of the more general ability to create a control system for a Plant automatically. The process of actually entering the details of a Plant into a control system is at present generally accomplished by entering large streams of numbers and names that in some way describe the Plant. This is both tedious and error-prone, as this representation of the Plant is far from natural to humans. A better method would be to graphically enter the plant as a series of diagrams, supplying further detail only where necessary.

The computer "image" of the effluent plant was "created" by a combination of the two methods, most of the technical data being entered numerically, and later combined with Mimic diagrams designed on the Tesselator by a draftsman. Currently, in drawing the Mimic diagrams for the effluent plant, the draftsman selects objects (as icons, using a "trac-ball" which is a form of upside down mouse) and places them onto the screen. He can then connect objects by dragging pipes between them. The Tesselator stores up to 50 Mimic diagrams, and they can be animated (flashing, colours, etc) from the PDP-11. To actually create the plant graphically, the system could prompt the draftsman for necessary information as each object is declared (placed on the screen).

The possibility of having a draftsman actually create the entire plant image on the screen has prompted considerable research into this field on the part of large process control firms (ASEA, etc). Because of its declarative nature, Prolog provides a very convenient framework for such an application. For example, if the draftsman places a pump on the Tesselator screen, the Tesselator recognises it as a pump (all icons are pre-defined by the draftsman), and informs the (Prolog) system of the pump's appearance. Because the Prolog model has a skeleton for a pump

declaration,

pump(Name, Type, On/Off, Wattage, Capacity).

it knows what information is required by the system about a pump, and can ask the Tesselator to prompt the user for the required facts. Methods such as 'query by example' could be useful to make this process as natural as possible for the draftsman. When it has the required information, Prolog can simply declare the pump, possibly with some additional information from the Tesselator, (indicating, for example, which picture the pump lies in).

The concept of automatic, graphical plant modelling and configuration is not new, but in the past has been a very expensive undertaking. Less expensive hardware and more powerful software have made this field of far wider practical application, and it is attracting considerable interest overseas. The problems of elegantly incorporating complex rules (for instance those governing flow rates and directions, dilutions, configuration rules, etc) into such a plant are of particular interest in the context of Prolog.

(D). Expert System.

Because of the current interest[1] in knowledge based systems and "expert" systems, particularly those involving Prolog, the possibilities of incorporating an expert system in a Prolog-based process-control plant must be considered. The interest in this field is evident from the number of commercially available Expert systems for process control that have appeared during this year. In February 1985 there were none (as far as I could determine), but three have been advertised in "Automation and Control" in the past two months.

Because of it's rule based nature, Prolog has been a very popular language for describing model expert systems (for educational purposes) and for prototyping real expert systems. Despite this, most practical expert systems are currently written in languages such as (compiled) Lisp, which are executed far more efficiently on existing machine architectures. I think that it is fair to assume that this will eventually change, and that practical expert systems in Prolog will begin to appear, if not in large quantities. After all, the first Expert systems were not written in Lisp, but in Fortran, and only the advent of Lisp Machines has made expert systems in Lisp acceptably efficient. I see no reason why this cannot also be true of Prolog, especially in light of the claims of the Japanese 5GL project.

For the purposes of this report, I choose to accept that it is quite possible to implement an expert system for process control in Prolog, and be more concerned with what such a system could be used for.

[1. The September 1985 CACM and August 1985 BYTE feature several articles on "Knowledge Based systems"].

(E). Concurrent, Real-time Prolog.

(1). Alarm Diagnosis in Prolog.

Given a Prolog model of a plant that is static, but able to be animated, we may examine the suitability of Prolog for writing alarm diagnosis queries. As an example, the following is an outline of queries to diagnose the cause of a "High Tank Level" alarm:

```
highLevelSwitchAlarm( LVSw ) :-
        levelSwitch( LVSw, high, on ),
        levelSwitchInTank( LVSw, Tank ),
        ( checkLowerLevelSwitch( Tank ) ;
          checkLevelTransmitter( Tank ) ;
          tryLoweringTheLevel(   Tank ) ).

checkLowerLevelSwitch( Tank ) :-
        levelSwitch( LowSwitch, low, off ),
        levelSwitchInTank( LowSwitch, Tank ),
        write('Contradiction : Low level switch is off ').

checkLevelTransmitter( Tank ) :-
        levelTransmitter( LVT, Level ),
        levelTransmitterInTank( LVT, Tank ),
        tank( Tank, Capacity ),
        (( Level < Capacity,
           write('Contradiction : Transmitter says not full')) ;
         ( write('Confirmation : Transmitter agrees !!'))).

tryLoweringTheLevelOf( Tank ) :-
        closeAllValvesInto( Tank ),
        turnOnPumpsOutOf( Tank ),
        delay( 1200 ),
        examineLevelOf( Tank ).

closeAllValvesInto( Tank ) :-
        repeat,
        (( valve(V,_), connect(V,Tank,_), closeAValve(V), fail) ;
         ( true )).

closeAValve(V) :-
        retract( valve(V,open)),   assert( valve(V,closed)).
```

which ignore such considerations as tanks fed directly by sources, tanks without pumps, etc.

Points of import are:

1. The simple way in which the context surrounding one level switch (it's associated tank, other switches, etc) is accessible from the model base. One strength of Prolog is the way in which arbitrary data structures may be related.

2. The difficulty of iterating over all valves connected to a tank. Although Prolog's backtracking is a higher level control structure than iteration, it sometimes forces inelegant code to implement a simple concept such as the universal qualifier (for all).

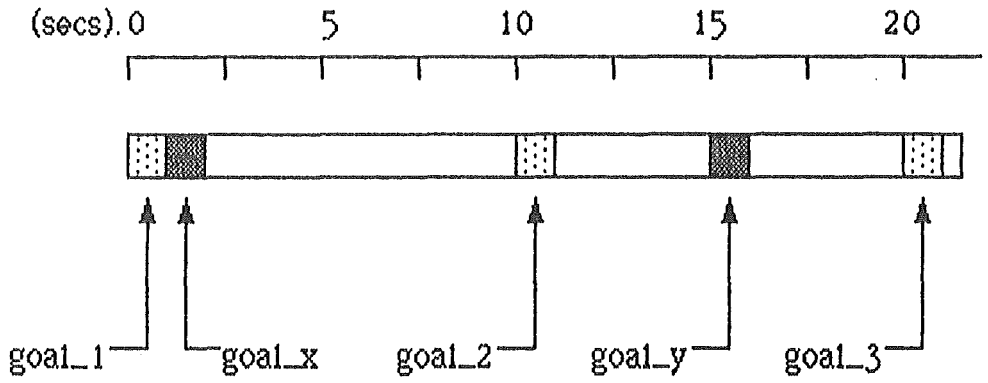
3. The intrusion of time in the clause **tryLoweringTheLevelOf**, in the form of a predicate called **delay**. This predicate is assumed to allow the clause to pause for 1200 (seconds) between turning on the pumps and examining the level of the tank. Because of the "slow" nature of this particular plant, it is often necessary, in determining the exact cause of an alarm, to perform some actions, wait for a period of (real) time, and then examine the results of that action.

As a query such as the above is likely to require less than one second of processor time actually doing things, but 1200 seconds waiting for the results of those actions to become apparent, it seems reasonable to allow other queries to execute in the interim, for example the following two queries

query1 :- goal_1, delay(10), goal_2, delay(10), goal_3.

query2 :- goal_x, delay(15), goal_y.

could be executed approximately as follows:



where the gaps denote times when the processor may be idle, or attending to other tasks. If queries are allowed to be executed in an interleaved fashion, considerations such as data consistency become important, as does the amount of control that the Prolog programmer has over the way queries are executed.

It would seem that the best way to divide the responsibilities introduced by the concept of interleaved queries is:

1. Make the Prolog programmer responsible for not relying upon some fact which may have become invalidated by the intervening execution of some other Prolog query.

For example, a query such as

ifItsThere_DeleteIt(X) :- X, delay(10), retract(X).

is potentially asking for trouble, and should be avoided. In practice, this is not a very restricting requirement, and primitives can be easily written (at the Prolog level) to allow a query such as this to be executed without interference if necessary.

2. Make the implementor responsible for ensuring that database updates (asserts and retracts) be executed as atomic operations to avoid the concurrent update problem. This is the situation which would arise if two queries attempt to update a fact at the same time, and their interleaved execution produces a result that is different from that of any serialised execution of the two queries.

(2). Related Research.

Some Research has been performed in the area of concurrent execution of Prolog queries, namely

1. Using Prolog for discrete event simulation, and
2. Parallel execution of Prolog queries.

and it is worthwhile to briefly examine these approaches.

Firstly, a ("very") high level language called T-Prolog (where the "T" stands for "time") was developed for discrete event simulation by Iván Futó and János Szeredi, in Hungary. This system provides many of the (pseudo-concurrent) coroutine facilities of Simula within the framework of Prolog. The system has primitives for creating new processes and manipulating resources, an internal clock, and a scheduler. Processes may communicate through common logical variables, the (common) shared database, or by sending and receiving messages. The scheduler is similar to that of Simula, where (simulation) time is incremented to that of the next scheduled event on the event queue. The system permits backtracking in simulation time in the case of a deadlock or failure situation. T-Prolog is only experimental, although it has been used for some years in Hungary, and several Universities have acquired the system.

Secondly, the Japanese "Five year Artificial Intelligence Project" is using a language "similar" to Prolog, called Parallel Prolog (or, in some circles, Concurrent Prolog), variations of which have been developed in various parts of the world (eg. by Ehud Shapiro in Israel, John Cleary in Canada, etc.). Parallel Prolog is essentially standard Prolog that has been extended and restricted to allow different goals of a query to be executed on different processors (genuine concurrency). A query of the form

query :- someGoal // anotherGoal.

specifies that the two goals should, if possible, be executed on separate processors, but if only one processor exists, they should be executed with

separate stacks to emulate the effect of parallel execution. The actual distribution of tasks is the responsibility of the underlying operating system, and Parallel Prolog does not rely upon it. This mechanism necessarily restricts the backtracking facilities available to the user, and this is the largest difference from the programmer's point of view. A version of Parallel Prolog (Parlog) has recently become available for the Department's VAX.

(3). Requirements for a Concurrent Real-Time System.

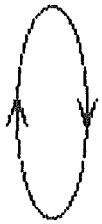
Neither of these approaches is quite what is desired in the real-time control environment. Parallel Prolog is really only of value on a multi-processor architecture, and T-Prolog takes no account of **real** time. T-Prolog has, however, many features that appear quite applicable to real-time process control, and if one ignores the possibility of backtracking in real time, (for reasons of sanity) T-Prolog seems to be a good basis from which to design a Prolog that enables (pseudo) concurrent queries in real-time.

The facilities required of such a system are the ability to;

1. halt a query for some period of real time,
2. execute queries in an interleaved fashion, and
3. communicate between concurrently executing queries.

[The communication can be either via the shared database (implicit), or using some higher-level protocol such as message passing (explicit).]

As most Prolog interpreters perform a



- read (next query).
- interpret.
- execute.

cycle, if a query that is being executed asks to be **delay**ed, or to **wait** for some condition to be satisfied, the interpreter can carry on to read, interpret and execute the next query in line (until a point is reached when no more concurrent queries can be started due to lack of workspace). At this point the interpreter must wait for a query to finish, or if no queries are either running or waiting on time, declare the situation a deadlock.

The above pseudo-concurrent process may be interpreted as an example of coroutines, which is convenient, as Modula-2 (the language upon which TS would most likely implement concurrent Prolog) features coroutines.

(4). Prolog Memory Management.

In order to implement interleaved execution of Prolog queries, it is necessary to have an understanding of Prolog's memory management scheme. Prolog implementations vary in their choice of strategy, but the following describes in general terms the approach taken by the most common Prolog systems.

Clauses in the database are represented in skeletal form. Bodies which are a conjunction of subgoals are represented as a chain of skeletal terms linked together as Brother nodes. All the clauses for functions with the same name are stored in a chain, and a pointer to the head of this chain is kept in the atom node for the name.

Overflow chained hash tables are used to store names of terms, atoms and variables, and hashing is used at execution time to enable fast unification of a clause head with a goal to be satisfied.

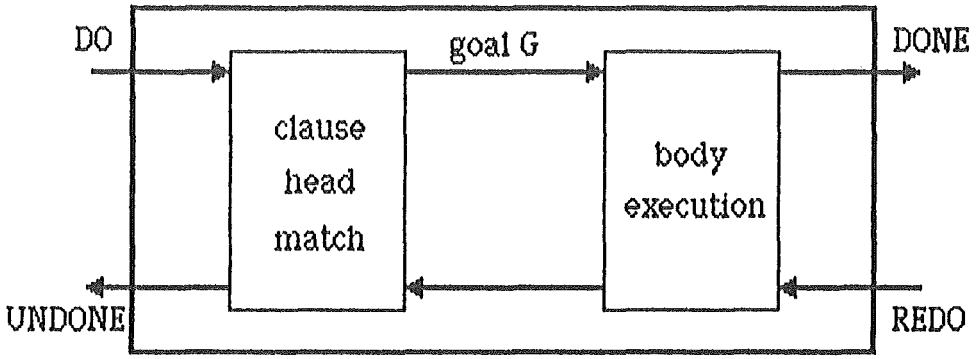
The execution of a Prolog query conceptually builds a "proof tree" (the leaf nodes of which represent the solutions to the query), but because of the depth-first strategy, this tree may be represented as a stack. In practice, Prolog goals are executed by an abstract machine, the memory of which is organised as two stacks:

(a). The local stack, holding clause activation records and local storage for clauses, and

(b). The global stack, holding terms created during execution.

The activation records (or "binding environments") enable reentrant execution of Prolog clauses. Each activation record on the stack also represents a backtrack point, to which execution returns if the next node up the stack fails.

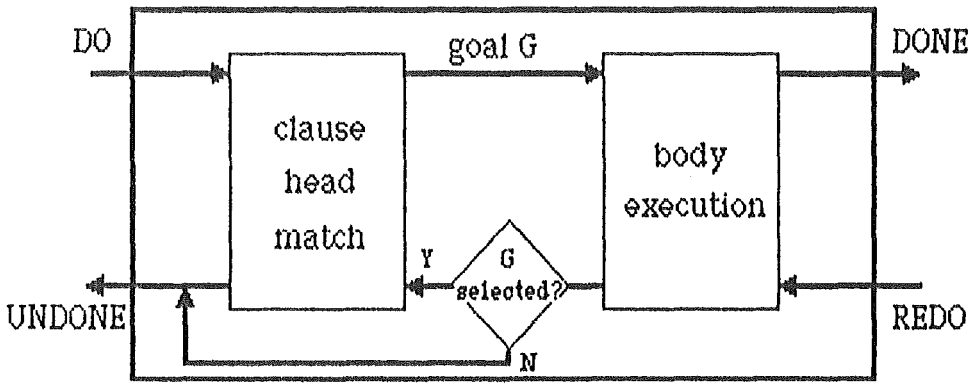
Most Prolog interpreters do not generate backtrack points for built-in procedures (such as write, assert, etc), and thus if execution backtracks to such a procedure, the effect is to fail that goal (and backtrack to the previous goal). This **selective backtracking** contradicts the declarative semantics of Prolog, but ensures that the procedural semantics have the "desired" effect. In general, every node of the (virtual) proof tree can be illustrated as a diagram



where the DO port is entered when the goal is first activated, whereas the DONE port is exited on complete execution of the goal. On backtracking, control re-enters the goal execution box via the REDO port, and exits at the UNDONE port on unsuccessful execution. The goal execution box is itself made up of two similar boxes: the clause-head matching box and the clause-body execution box.

The central concept of selective backtracking is to select, at each failed goal, a single goal to backtrack to, not necessarily the previous goal as in standard backtracking. Entry into the REDO port of G is allowed only if G

has been selected as the backtrack goal for the last failed goal; otherwise control flows directly to the UNDONE port of G, viz.



The most common Prolog interpreters "implement" selective backtracking by treating built-in predicates as special cases, and not actually placing an activation record for them on the stack. Thus any attempt to backtrack past them appears as if they have failed upon backtracking.

(5). Proposal for Concurrent Real-Time Prolog.

To implement Prolog queries that are capable of being executed in an interleaved manner, we must provide each query with its own local and global stacks, as well as access to the (shared) database. To cause a query to wait for n seconds, a procedure **delay(n)** is required. Waiting on specified conditions is also possible, and can be implemented in one of two ways;

(a). At the Prolog level, goals can now be declared which wait for complex conditions to become satisfied before continuing, for example

```
waitCondition(C) :-
        clause(C,Body),
        repeat,
        ( Body ; ( delay(0), fail )), !.
```

can be used to wait for any clause **C** to become true. This predicate

assumes that **delay(0)** causes the current query to relinquish the CPU and allow other queries to execute. This method ("active" wait) obviously places a heavy load on the system, but it also enables a query to respond to implicitly occurring events in the database.

(b). At the implementation level, primitives can be supplied to implement semaphores or signals, and queries can wait on conditions that are signalled from other queries. This represents a higher (and more efficient) level of synchronisation protocol, but with the requirement that events must be explicitly signalled. A signal is defined as a (possibly empty) queue of waiting processes. There are two operations on signals:

send(s) - **if** queue(s) is not empty **then**
remove first process from queue(s) and resume it.
end;

wait(s) - suspend calling process and place it at end of queue(s).

Each signal should correspond to a certain condition or state of the database (or queries manipulating it). The sending of a signal reactivates at most one process (otherwise one of the awakened processes might quickly invalidate the condition, causing the other processes to proceed on false premises). Signals are quite easily implemented in Modula-2.
[Signals are used in a manner similar to semaphores (Dijkstra), but can be considered as more primitive entities, since semaphores can be expressed in terms of signals and ordinary variables].

It is felt that both approaches should be allowed as this provides flexibility of operation to the Prolog programmer.

In summary, the features required are:

delay(N) - where N is a number of seconds,

wait(S) - where S is a Signal,

send(S) - where S is a Signal,

and with these primitives, we can examine the implementation of concurrent real-time queries in Prolog. It may also be necessary to provide a query with the ability to start another query (as a concurrent process), but this is a special case of the read-interpret-execute cycle, and should pose no extra problems.

These primitives should not generate backtrackpoints, as the intent of a clause such as

clause :- goal1, delay(10), goal2.

is probably that if goal2 fails, goal1 should be re-satisfied, (rather than merely delaying again before re-trying goal2). Selective backtracking for these predicates may be implemented as for other built-in predicates, and poses no extra expense. Should it be desirable that a backtrackpoint is implemented for such a predicate in a certain situation, the effect may be simulated via

clause :- goal1, repeat, delay(10), goal2.

or some similar construct.

For purposes of nomenclature, concurrent real-time Prolog is henceforth referred to as CRT Prolog.

(6). Use of CRT Prolog.

The Prolog programmer may use the facilities provided to implement **resources** enabling him to lock portions of the database at any point. The following clauses provide resource management at the Prolog level:

```
newResource(R) :- assert(resource(R)), assert(available(R)).
eraseResource(R) :- available(R), retract(resource(R)),
                    retract(available(R)).
seize(R) :- resource(R), available(R), retract(available(R)).
release(R) :- resource(R), not available(R),
              assert(available(R)).
```

and a query to seize a resource, do some processing, and then release it

could look like

```
query :- ...., waitCondition(available(thisResource)),
                seize(thisResource),
                doSomethingWith(thisResource),
                release(thisResource), ....
```

Obviously these predicates require the Prolog programmer to be disciplined in his access of possibly critical parts of the database, as they do not enforce adherance to resource usage, but implementing them at the Prolog level provides the maximum flexibility, as well as the simplest implementation. Adherance to the resource manipulation conventions could be enforced if resources were implemented at a lower level, or if the identity of the Prolog query requesting a resource could be determined.

4. Practice (What I actually DID).

(A). Modelling The Effluent Plant.

(1). Static Model.

At the beginning of 1985 the only Prolog implementation available at Canterbury was an interpreter written in C for the Prime 750. This interpreter is not particularly robust, and is rather limited in its supply of built-in functions. The arrival of the VAX and C-Prolog provided a much more useful environment and implementation, although this was still interpretive. The Lisp/Prolog system from Salford that was tested on the Prime during this year provided a compiler for Prolog clauses, although by the time this arrived the effluent plant model had been developed on the VAX (and performance was at no stage a primary concern so the efficiency of the Salford system was not needed).

Implementing a static model of the plant in C-Prolog proved a reasonably straightforward task, as the mapping from a draftsman's plan of the plant to a Prolog model is fairly mechanical, the only design decisions being which objects to abstract away, and which to keep. I chose to declare components such as level switches and stirrers separately from the tanks or ponds in which they reside, (despite the fact that they are always associated with some tank or pond), in the interests of generality. For example,

levelSwitch(lVSw17, high, off, peMakeupTank).

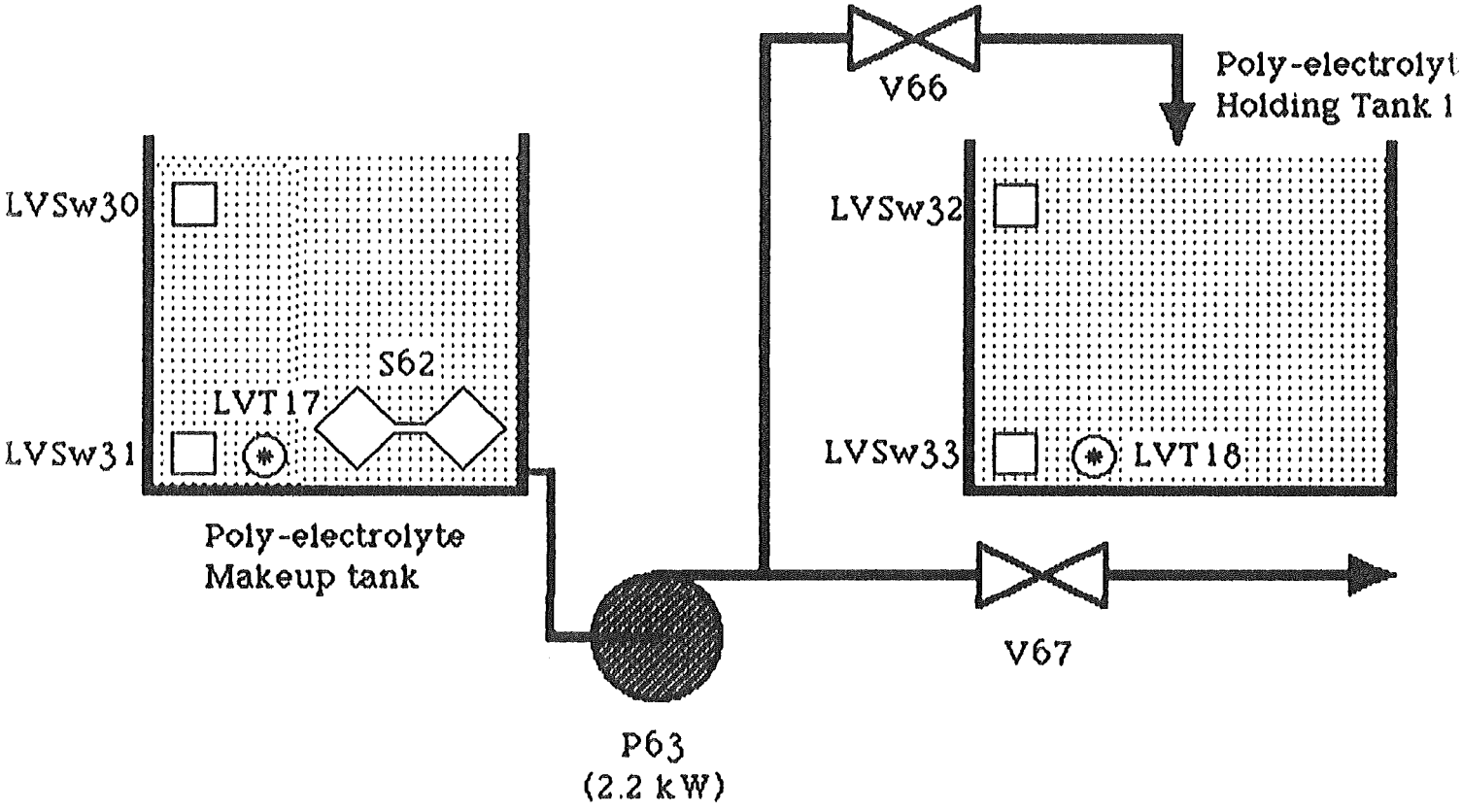
is more concise than

levelSwitch(lVSw17, high, off).

levelSwitchInTank(lVSw17, peMakeupTank).

but I feel that the extra flexibility of the latter is worth the verbosity.

As an example, the following section of the effluent plant



could be represented in Prolog by:

/* Declare all of the components */

```
tank( peMakeupTank, mixTank, 1.5).
tank( peHoldingTank13, duty, 7).
levelSwitch( lVSw30, high, off).
levelSwitch( lVSw31, low, on).
levelSwitch( lVSw32, high, on).
levelSwitch( lVSw33, low, on).
levelTransmitter( lVT17, 1.0).
levelTransmitter( lVT18, 6.98).
stirrer( s63, on).
pump( p63, fixed, on, 2.2, 9.0).
valve( v66, open).
valve( v67, closed).
```

```
/* Declare relationships between the components */

levelSwitchInTank( lVSw30, peMakeupTank).
levelSwitchInTank( lVSw31, peMakeupTank).
levelSwitchInTank( lVSw32, peHoldingTank13).
levelSwitchInTank( lVSw33, peHoldingTank13).

levelTransmitterInTank( lVT17, peMakeupTank).
levelTransmitterInTank( lVT18, peMakeupTank).
stirrerInTank( s62, peMakeupTank).

/* Declare physical connections between components */

connect( peMakeupTank, p63, 9.0).
connect( p63, v66, 9.0).
connect( p63, v66, 9.0).
```

The modelling process required several communications with CFM to ascertain the directions of flow in some parts of the plant, and to determine the exact specifications of some items of hardware. It was interesting to find that even CFM did not know exactly what some of their equipment was capable of !. In this vein, Prolog is quite happy to accept a fact such as

tank(sump4, bigSump, ???).

where ??? signifies that sump4's capacity is not known (and cannot be determined). Several of the facts comprising the Prolog Model of the plant "feature" such indecision, but these are rather obscure parts of the plant, and are not really relevant to the testing of the model.

The Prolog model of the entire plant appears as an Appendix.

(2). Animation of the Model.

In order to animate the model, a Prolog program is required to repeatedly read a new fact from a file, and update the corresponding existing fact in the database. The following clever (convoluted) Prolog enables this without knowing what the fact to be replaced actually was, due to the unique name of each component of the plant:

```
animate :-
        see( updateFile ),
        repeat,
          read(Term),
          removeOld( Term ),
          assert( Term ),
          random( N, maxDelay ),
          delay( N ),
        fail.

removeOld( T ) :-
        functor( T, Name, Arity ),
        T =.. ListOfT,
        findFirst( ListOfT, First ),
        listOfVars( LOV, Arity - 1),
        append( [Name,First], LOV, FullList ),
        OldT =.. FullList,
        retract( OldT ), !.

findFirst([Name,First|_], First).

listOfVars( [], 0 ).
listOfVars( L, N ) :-
        listOfVars( L1, N - 1 ),
        append( [Var], L1, L ), !.
```

And each iteration, the predicate **animate** takes the new fact read from the file, extracts the functor (predicate name) and the first argument (which will be the name of the object in the plant), generates a list of enough variables to consume the remaining argument spaces, and then retracts the corresponding fact. It then asserts the new fact.

This produces the effect of a very simple live plant, and complex inter-relationships may be build into the actual facts in the **updateFile**. For serious simulation of an operating plant, the time between alterations would need to be dependent on the actual alarms arising, rather than (pseudo) random. Such complex simulation is beyond the scope of this report.

(B). A Prolog Interpreter.

In June, a Prolog interpreter written in Pascal was kindly donated to TS by the Meat Research Institute (courtesy, Mr. Mark Loeffen, who is probably the only person in N.Z. able to proudly say that he has a micro-VAX II under his desk). We were not told where the MRI acquired it, and deemed it best not to ask, but the fact that all the identifiers are Dutch seems to hint at its origin.

(1). Translation of the interpreter. The interpreter comprised 2200 lines of Pascal source, and we received four pages of documentation (in English !) describing how to alter parameters of the system. The task of converting the Pascal source to Modula-2 was relatively straightforward, the only real difficulties arising in the form of a **goto**. It required several hours to produce the same effect in Modula-2, although most of this time was spent ensuring that no errors had been made.

Having produced what was possibly the world's first implementation of Prolog in Modula-2, the rejoicing was short lived. This Prolog interpreter proved to be very rudimentary, with only half a dozen built-in functions, no support for file I/O, completely static database size, and no ability to recover the storage freed by retracting clauses from the database (actually, there was no facility for asserting or retracting clauses from within a Prolog program !!). This was deemed unsatisfactory, and another Prolog interpreter written in Pascal was ordered from the University of York. This appeared to be a much better, complete, "standard" implementation of Prolog, (also written in Pascal). Despite several (telephone) assurances that it was on the way, nothing had materialised by July, and it appeared best to work with the Dutch Prolog, and assume that whatever was learned with it could be applied to the York interpreter at least as easily.

Syntax Modifications.

The Dutch Prolog accepted facts as

+ fact(a,b);

and queries as

- fact(*X, *Y);

(yes indeed, variables are actually *X). Compound queries are represented thus;

- query(*X) - clause(*X) - / - another(*X, *Y);

where the **/** is the cut symbol and **;** is the termination symbol. No form of OR is provided, and must be achieved by alternative clauses.

Lists were implemented in the most primitive manner, namely via the dot (.) operator, for example the list [a, b, c, d] can only be represented by the following **.(a, .(b, .(c, .(d, nil))))** which is not particularly lucid.

The above proved a nuisance, and the interpreter was altered to accept facts as **+fact(a, b);** queries as **? fact(a, *X);** and compound queries of the form

? query(*X) - clause(*X), !, another(*X, *Y);

which is nearer "standard" Prolog syntax. No attempt was made to alter the syntax of lists, as this is a rather more complex undertaking, and this interpreter was only intended as a test-bed for the concepts of concurrency and real-time operation in Prolog.

(C). Effluent Plant Configuration Using Prolog.

The Effluent Plant contains 4 large Reaction Tanks in which the effluent is combined with first acid, and then lime. Chemical considerations dictate that certain tanks may contain only specified chemicals, a table of possible states each tank may take being

tank 1	acidTank	———
tank2	limeTank 1	acidTank
tank3	limeTank2	limeTank 1
tank4	limeTank2	———

(each tank may also be "standby", "out of service" or "bypassed").

From this table, a total of 12 valid configurations may be generated, and in this case, it would not be difficult to simply enumerate the cases, and verify configurations from these. However in a larger plant, many more options may be present, and we require a general tool for generating and/or verifying such configurations. A plant may have a valid configuration, yet not be in a runnable state, for instance if each tank was taken out of service for maintenance or cleaning.

Because a Prolog function can generate solutions as easily as verifying them, it is very easy to write a Prolog program to solve the above problem:

```
configurations( T1, T2, T3, T4 ) :-
        valid( tank1, T1 ),
        valid( tank2, T2 ),
        valid( tank3, T3 ),
        valid( tank4, T4 ).
```

```
runnableState( T1, T2, T3, T4 ) :-
        configurations( T1, T2, T3, T4 ),
        member( acidTank, [ T1, T2, T3, T4 ] ),
        member( limeTank1, [ T1, T2, T3, T4 ] ),
        member( limeTank2, [ T1, T2, T3, T4 ] ),
        ( member( standby, [ T1, T2, T3, T4 ] ) ;
          member( outOfService, [ T1, T2, T3, T4 ] ) ;
          member( bypassed, [ T1, T2, T3, T4 ] ) ).

member( X, [X|_] ).
member( X, [_|Y] ) :- member( X, Y ).

valid( tank1, acidTank ).
valid( tank2, limeTank1 ).
valid( tank2, acidTank ).
valid( tank3, limeTank2 ).
valid( tank3, limeTank1 ).
valid( tank4, limeTank2 ).
valid( _, standby ).
valid( _, outOfService ).
valid( _, bypassed ).
```

where the **configurations** predicate generates all valid configurations in order of preference (due to the order of the **valid** facts), and **runnableState** checks that such a configuration represents a state in which the plant is operational. Preferences are accounted for in the order in which **valid(_,_)** facts are declared to the system.

Some examples of use are;

1. Generate any configuration

runnableState(S1, S2, S3, S4).

results in

S1 = acidTank

S2 = limeTank1

S3 = limeTank2

S4 = standby .

which is CFM's preferred configuration.

2. Set tank 3 as out of service (for cleaning).

runnableState(S1, S2, outOfService, S4).

results in

S1 = acidTank

S2 = limeTank1

S4 = limeTank2.

3. Set up all tank states explicitly

runnableState(outOfService, limeTank2, limeTank1, acidTank).

results in Prolog saying

no

as this is not a valid configuration (deliberate error, honest !!).

Repeated calls to the above functions will generate all valid configurations, and all runnable states (in some order of preference).

To program even the above simple case in a procedural language like Modula-2, in a general way, would be quite difficult, and as the number of choices increases, the complexity of the program would soon get out of hand. In Prolog, because of its pattern matching nature, this task is very straightforward, and the solution is much more transparent.

(D). Implementing CRT Prolog in Modula-2.

(1). Introduction

In making Prolog real-time and concurrent, it is not necessary to alter the actual (Prolog) interpreter a great deal, but rather to provide the required facilities at the level of the implementation language, in this case Modula-2. What is required at this level is a real-time process scheduler. Modula-2 supports coroutining and allows low level machine access. From these must be built mechanisms for concurrency control (using signals), and time dependency.

(2). Modula-2 Facilities.

In the Modula-2 report, Wirth describes a process scheduler based on coroutines, signals and a ring of process descriptors. This scheme takes no account of the real-time, and all process interaction is via signals. Although this scheme actually contains an error (the PDP-11 on which Wirth supposedly ran his scheduler cannot transfer execution from a process to itself, which his does !), its concept is sound, and provides a good basis for building a similar scheduler that **is** real-time. D. A. Sewry has provided several pointers concerning Modula-2 and real-time schedulers, and I have followed his approach, (although I also found an error in this .. can no-one get it right ?).

The (slightly) abstracted interface provided by the scheduler (which CRT Prolog utilises) is;

```
DEFINITION MODULE Processes;
TYPE
   SIGNAL;

PROCEDURE StartProcess( P : PROCEDURE;    WorkSpace : CARDINAL;
                                Priority : INTEGER );
(* Start a process P with Workspace and Priority *)

PROCEDURE StopProcess;
(* Terminate the current process *)

PROCEDURE SEND( S : SIGNAL );
(* Reactivate a process WAITing on S *)

PROCEDURE WAIT( S : SIGNAL );
(* WAIT for some process to SEND S *)

PROCEDURE DELAY( NumberOfSeconds : CARDINAL );
(* DELAY the current process for a NumberOfSeconds *)

END Processes .
```

In implementing the above scheduler, the following Modula-2 facilities were utilised:

1. The type PROCESS which represents a parameterless procedure and can be executed as a coroutine.
2. The procedure NEWPROCESS which creates a workspace for a PROCESS.
3. The procedure TRANSFER which transfers execution from the current PROCESS (saving its environment) to another PROCESS.
4. The procedure IOTRANSFER which is similar to TRANSFER, except that it also specifies an interrupt vector. When the corresponding interrupt occurs after the TRANSFER, an implicit transfer is made back to the original PROCESS. This procedure is useful for interrupt

driven I/O devices.

5. Software priorities for use with interrupt routines.
6. The procedure LISTEN which temporarily lowers the priority of a PROCESS or program to enable interrupts of a lower priority to be detected and processed.

(3). Physical Resources.

I had at my disposal (courtesy of TS) a SAGE IV microcomputer (Interrupt driven Motorola MC68000 at 8MHz without wait states, with 512k bytes of RAM). The SAGE can be run either under CP/M-68k or under the UCSD p-system. The p-system was chosen as the development operating system because it provided a better programming environment than CP/M, as well as a very quick one-pass Modula-2 compiler, even though the compiler is not native-code (naturally, it produces p-codes). The Prolog Interpreter was converted from Pascal to Modula-2 and tested on the SAGE, and was subsequently (as an exercise in portability) ported to an IBM PC, an ICL PC, and a MicroPDP-11/72.

Modula-2 on the SAGE simulates interrupts in software rather than relying on the underlying hardware interrupt mechanism. It provides access to the SAGE's interrupt system in the form of procedures to ENABLE and DISABLE interrupts, ATTACH or DETACH interrupt vectors from the underlying I/O system, and set the priority of the currently executing task (for critical regions of code).

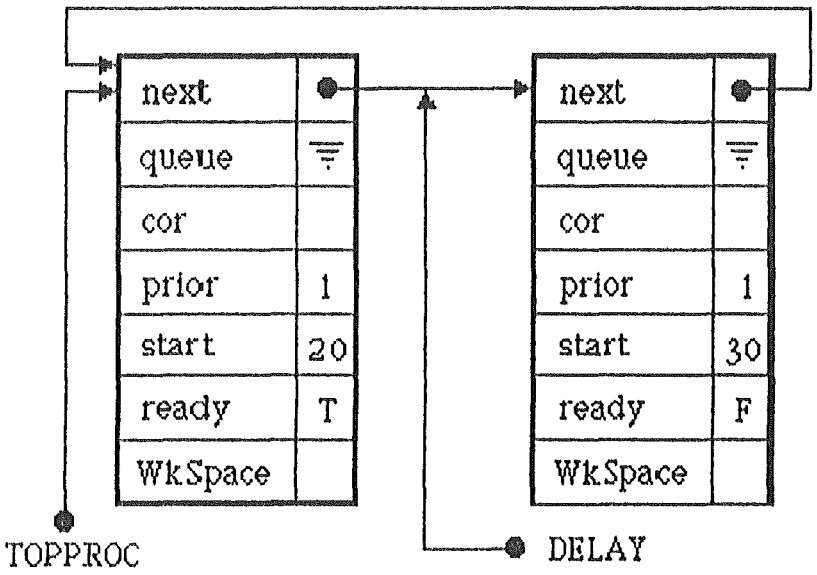
The SAGE itself supplies several types of I/O "events" which may be ATTACHed to a semaphore. Four of these are "scheduler events" which may be set up to signal their semaphore at periodic intervals. These can be coupled to an interrupt vector to implement timeslicing. Also provided are procedures to obtain the system TIME (since boot), as well as the ability to SLEEP for a length of time.

(4). The Modula-2 Process Scheduler.

Wirth's process scheduler implemented a ring of process descriptors, each of which is associated with a coroutine; its workspace, priority, and status. A new coroutine is "created" by a call to the procedure StartProcess, which creates a new process descriptor, slots it into the ring, and associates a new workspace with this descriptor. Transfer between coroutines is only effected via explicit calls to WAIT and SEND. When a coroutine executes a WAIT, its process descriptor is placed on the queue for the signal involved, and execution is transferred to the next process in the ring that is able to execute (not waiting). Thus a coroutine may only be in the queue for one signal at any time.

Wirth's scheduler did not specify what happens when a PROCESS terminates, and most implementations following his scheme terminate the entire program when a coroutine (PROCESS) execution ends. In our case, we want Prolog queries to be able to terminate without affecting the remainder of the system, and while at least one coroutine is alive, the program should not terminate. To facilitate this, a procedure StopProcess is introduced, and every coroutine is required to execute a call to StopProcess as their last statement (Will and Testament). StopProcess removes the process descriptor of the current process from the ring, and deallocates the storage used by it (workspace and descriptor).

For real-time behaviour, it is required that a process transfer take place not only when SEND and WAIT are called, but also when a coroutine executes a DELAY, and possibly when a process with a higher priority than the starting process is started. DELAYed processes can conveniently be chained to a signal called DELAYED, in order of the time at which they return to life (closest first).



The SAGE clock may be accessed via a call to the system TIME procedure. One of the four scheduler events may be used to generate an interrupt periodically for timeslice purposes. Setting the timeslice to 100ms and the clock to priority 3 seemed reasonable.

(a). Timeslicing

At this point a major decision needed to be made concerning the form that the timeslice routine may take. There are essentially two approaches:

1. Preemptive Scheduling. At each timeslice, the currently executing process is swapped out, and the next in the ring will be restarted when the timeslice process ends. A priority scheme may be used to allocate each process a certain number of slices (say the same number as its priority), and if the process is timed out this number of times in a row, it is then swapped out and another process is resumed. This scheme is specially applicable in an operating system environment.

2. Non-preemptive Scheduling. At each timeslice, merely examine the list of processes waiting on time (DELAYed processes) to determine which have served their time, setting these as "ready-to-run". Process switches only take place when a process explicitly WAITs, SENDs, or DELAYs, and the process with the highest priority that is ready-to-run is started at this

point. In the case of Prolog queries which spend a small time executing and a much larger amount of time waiting, there is rarely a need to slice a query out. A considerable side-effect is inherited from this mechanism, namely the atomicity of all Prolog queries (including assert and retract operations). Thus the Prolog programmmer can write his real-time queries knowing that noone will interrupt him, and noone will execute interveningly unless he WAITs, SENDs, or DELAYs.

The second strategy was eventually chosen because of the atomicity benefits. Initially both were implemented, but the difficulties introduced by preemptive scheduling and the convenience of non-preemptive scheduling from the Prolog programmer's point of view meant that the latter appeared to be the best in this situation. [A compromise of these two schemes could be to specify at newprocess time whether or not a process is to be interruptable at all, or make this interruptability a function of the process' priority].

(b). Outline of routines involved.

As the clock was started at priority 3, the process ring should exist at some priority $p > 3$ in order that procedures which manipulate the ring are uninterruptable. Modula-2 allows the programmer to specify a module's priority, and thus the process ring could be protected by enclosure in a module with priority (say) 4.

When the process scheduler is imported by any other program, it enters a process descriptor into the ring for that (main) program. This is to ensure that the main program is treated as a process in an analagous way to other processes it may spawn.

The following are the basic algorithms used by the procedures that manipulate the process ring:

```
StartProcess.
Allocate workspace and process descriptor.
Insert into the ring ( in order of priority ).
Disable interrupts, start at priority, and enable again.
Start highest read-to-run process.

StopProcess.
IF this was the last process on the ring THEN halt.
Remove process descriptor from the ring.
IF there are no ready-to-run processes THEN
        IF there are no processes waiting on the clock THEN
                || Deadlock ||
        ELSE
                Wait for first sleeping process to awake
        END;
END;
Deallocate workspace and process descriptor
Transfer to next ready-to-run process.

SEND( S ).
IF there are processes WAITing on S THEN
        Mark the first as ready-to-run.
        Transfer control to it.
END.

WAIT( S ).
Insert process descriptor at end of queue for S.
IF there are no ready-to-run processes THEN
        IF there are no processes waiting on the clock THEN
                || Deadlock ||
        ELSE
                wait for first sleeping process to awake
        END
END
Transfer execution to next ready-to-run process.
```

DELAY(N).
Calculate the time (ticks) at which to awake.
Insert process descriptor into DELAYED queue at correct point.
IF no ready-to-run process THEN
    wait for first sleeping process to awake.
    (* There must be one .. we just suspended it *)
END.
Transfer execution to next ready-to-run process.

Deadlock situations may be easily detected by the process scheduler, as they occur when there are no processes ready-to-run or waiting on time. In the context of a CRT Prolog interpreter, a deadlock would occur only when all executing queries are WAITing on events other than time, and there is insufficient memory to read, interpret and execute a new query.

(5). Testing the Process Scheduler.

A classic concurrent processes example is that of a Producer process generating a stream of data, which is transmitted to a Consumer process. The above process scheduler may be used to implement this example in Modula-2:

```
MODULE ProducersAndConsumers;
IMPORT BUFFER, Datum, Create, Full, Empty, Put, Get,
       Produce, Consume;
FROM Processes IMPORT StartProcess, StopProcess,
                      SEND, WAIT, Init, SIGNAL;
VAR
   B : BUFFER;
   nonFull, nonEmpty : SIGNAL;

   PROCEDURE PRODUCER;
   VAR
      x : Datum;
   BEGIN
      LOOP
         Produce(x);
         IF Full(B) THEN
            WAIT( nonFull );
         END;
         Put( B, x );
         SEND( nonEmpty );
      END;
   END PRODUCER;

   PROCEDURE CONSUMER;
   VAR
      x : Datum;
   BEGIN
      LOOP
         IF Empty(B) THEN
            WAIT( nonEmpty );
         END;
         Get( B, x );
         SEND( nonFull );
         Consume(x);
      END;
   END CONSUMER;
```

```
BEGIN
   Init( nonFull );  Init( nonEmpty );  Create(B);
   StartProcess( PRODUCER, 200, 1);
   StartProcess( CONSUMER, 200, 1);
   StopProcess;
END ProducersAndConsumers .
```

and although this example does not utilise the process scheduler's real-time capabilities, it does illustrate the SIGNAL facilities and their use. The details such as the BUFFER and its associated primitives are abstracted away to make the concurrent operation clearer.

A more appropriate (to this report) example is the following, which may be viewed as a skeleton CRT Prolog interpreter.

```
MODULE Prolog;

PROCEDURE Execute;
BEGIN
   DELAY( RANDOM );
END Execute;

BEGIN
   LOOP
      ReadAQuery;
      LOOP
      IF AvailableSpace THEN EXIT END;
         DELAY(0);
      END;
      StartProcess( Execute );
   END;
END Prolog .
```

This example illustrates a Prolog interpreter as a loop which reads a query, waits until there is space to execute a query, and then start it as a (pseudo-concurrent) process , before returning to read another query.

The main loop should be at a lower priority than the actual "execute" processes, so that these execute whenever they can, and new queries are only read when everything is delaying.

(6). Applying the Process Scheduler to the Prolog Interpreter

Unfortunately the Prolog interpreter that was converted to Modula-2 differed from the norm in that it implemented its database and execution stack as a single global array, and separating the two to enable concurrent queries appeared to be a pointlessly complex operation.

[As it supplied no assert or retract primitives, this interpreter needn't be concerned about the database changing during the execution of a query, and it was thus able to start its stack from the current top of the database].

Nonetheless, the interpreter was altered so that a **delay** predicate could be called to cause a query to pause for a period of time. However the user must ensure that he does not attempt to start a new query until the first has terminated as the two share the same "stack" space(!).

The Prolog interpreter from the University of York features two execution stacks and a separate dynamically-sized database, and should prove much more condusive to modification for concurrent real-time behaviour. As this arrived only recently, it has not been examined in detail, but from the (extensive) documentation accompanying the system, it appears to be a powerful and flexible interpreter, and has already been translated to Modula-2 by TS.

5. Conclusions.

Several aspects pertaining to the application of Prolog to real-time process control have been examined. Perhaps intuitively, Prolog would seem to be ill-equipped for real-time control, as it lacks low-level communications and concurrency facilities, but I feel it has been shown that with minor modifications, Prolog provides an elegant and convenient programming environment in some control situations.

In particular, Prolog seems well suited to plant configuration and modelling, and naturally to expert systems, where its rule-based nature allows elegant expression of solutions. In the area of alarm diagnosis, Prolog is an acceptable language, and for other applications such as actual device control, Prolog is simply unusable.

Recently papers have appeared concerning the alteration of Prolog to correct formal errors in its backtracking mechanism, but I feel that for many applications, consideration should be given to the inclusion of conventional control structures in Prolog. The Salford Lisp/Prolog system in which Lisp functions may be called from Prolog programs (and vice versa) appears to provide quite a useful combination of the two languages, and such amalgamations may prove to be the tool required.

As to the future, the Japanese 5GL project will doubtless produce some form of Prolog (or similar) machine, enabling truly concurrent execution of "Prolog" queries, which could be applied to the real-time process control environment as distributed hardware becomes more widely available.

For the meantime, TS are examining the practical application of Prolog to the alarm diagnosis, configuration and plant design problems as concerns the effluent plant over the summer of 1985/86, with the possibility of actually using Prolog in an Automated Machine Room to be designed for CFM in 1986.

6. References.

[1]. Clocksin, W. F. & Mellish, C. S; *Programming In Prolog.* Springer-Verlag, 1981.

[2]. Wirth, N; *Programming In Modula-2, Second Edition.* Springer-Verlag, 1983.

[3]. Young, S.J; *Real Time Languages, design and development.* Ellis Horwood, 1982.

[4]. Brinch Hansen, P; *The Architecture of Concurrent Programs.* Prentice-Hall, 1977.

[5]. Coelho H., Cotta J. C. & Periera L. M; *How to solve it with Prolog* Ministério da Habitação e Obras Públicas, Lisbon, 1980.

[6]. Sewry, D. A; "Modula-2 Process Facilities", *ACM SIGPLAN Notices, Vol. 19, November 1984.*

[7]. McDermott, D; "The Prolog Phenomenon", *ACM SIGART No.72, July 1980.*

[8]. Warren, D. H. D. & Periera, L. M; "Prolog - The Language and its Implementation compared with Lisp", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages. SIGART Newsletter No.64, August 1977.*

[9]. Andrews G. R. & Schneider, F. B; "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys, Vol.15 No.1, March 1983.*

[10]. Wirth, N; "Towards a Discipline of Real-Time Programming", *CACM, Vol. 20, No. 8, August 1977.*

[11]. Hoppe, J; "An operating system nucleus in Modula-2", *Software Practice & Experience, Vol.10, 1980.*

[12]. Futó, I. & Szeredi, J; "A very high level Discrete Simulation System, T-Prolog", *Computational Linguistics and Computer Languages, Vol. 15, 1982.*

[13]. Several papers from *Logic Programming,*

(a). Bruynooghe, M; "The Memory Management of Prolog Implementations".

(b). Periera, L. M & Porto, A; "Selective Backtracking".

(c). van Emden, M. H. & de Lucena Filho, G. J; "Predicate Logic as a language for parallel programming".

(d). Kowalski, R; "Logic as a Computer Language".

[14]. Canterbury Frozen Meat Company, Ltd; "Belfast Effluent Processing Plant - technical report", June 1984.

[15]. Clark, K.L. & McGabe, F.G; "Prolog: A Language for Implementing Expert Systems", *Imperial College Technical Report 80/21, November 1980.*

[16]. Knowledge Based Systems - *Communications of the ACM, Vol. 28, No.9, September 1985.*

[17]. University of York; *Portable Prolog System, Installation and Technical Manual.*

Appendices

A. Diagram of the CFM Effluent Plant

B. Sample Tesselator Mimic Diagrams

C. Listing of Process Scheduler

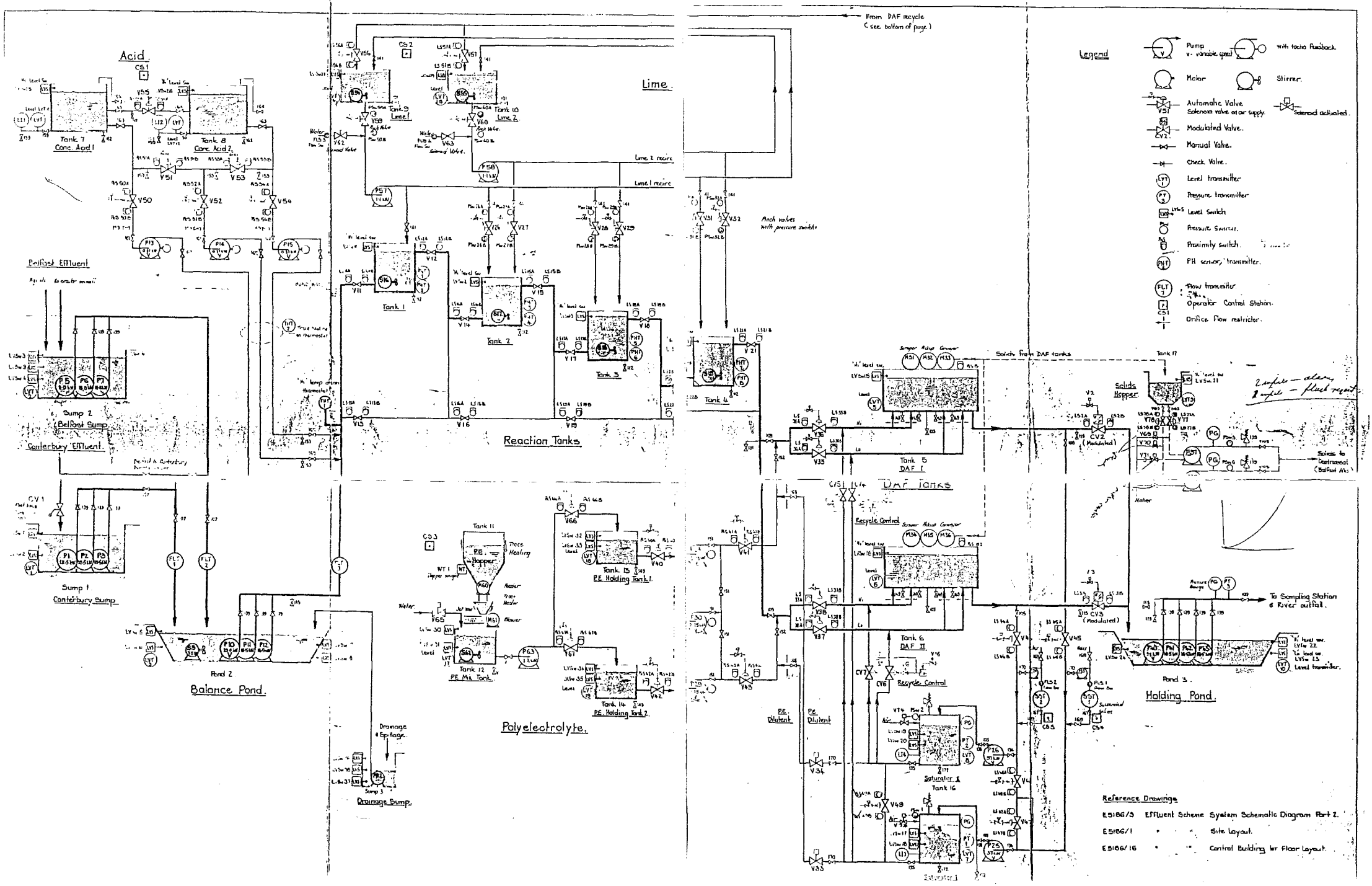
D. Listing of C-Prolog Model of the Plant*

E. Listing of Prolog Interpreter*

[* In accompanying listing folder.]

Appendix A

Draftsman's Diagram of the CFM Effluent Treatment Plant



Acid
Lime
Tank 7
Conc. Acid 1
Tank 8
Conc. Acid 2
Tank 9
Lime 1
Tank 10
Lime 2
Lime 2 recirc
Lime 1 recirc
Belfast Effluent
Sump 2
Belfast Sump
Canterbury Effluent
Tank 1
Tank 2
Tank 3
Tank 4
Reaction Tanks
From DAF recycle
(see bottom of page)
Pinch valves with pressure switch
Solids from DAF tanks
Tank 17
Solids Hopper
Tank 5
DAF I
DAF Tanks
Recycle Control
Tank 6
DAF II
Recycle Control
Sump 1
Canterbury Sump
Pond 2
Balance Pond
Tank 11
P.E. Hopper
Tank 12
P.E. Mix Tank
Tank 13
P.E. Holding Tank 1
Tank 14
P.E. Holding Tank 2
Polyelectrolyte
Drainage & Spillage
Sump 3
Drainage Sump
P.E. Dilutent
Saturator II
Tank 16
Saturator I
Pond 3
Holding Pond
To Sampling Station & River outfall
Legend
Pump
v - variable speed
with tacho feedback
Motor
Stirrer
Automatic Valve
Solenoid valve on air supply
Solenoid activated
Modulated Valve
Manual Valve
Check Valve
Level transmitter
Pressure transmitter
Level Switch
Pressure Switch
Proximity switch
PH sensor, transmitter
Flow transmitter
Operator Control Station
Orifice Flow restrictor
Reference Drawings
E5186/3 Effluent Scheme System Schematic Diagram Part 2.
E5186/1 " " Site Layout.
E5186/16 " " Control Building 1st Floor Layout.

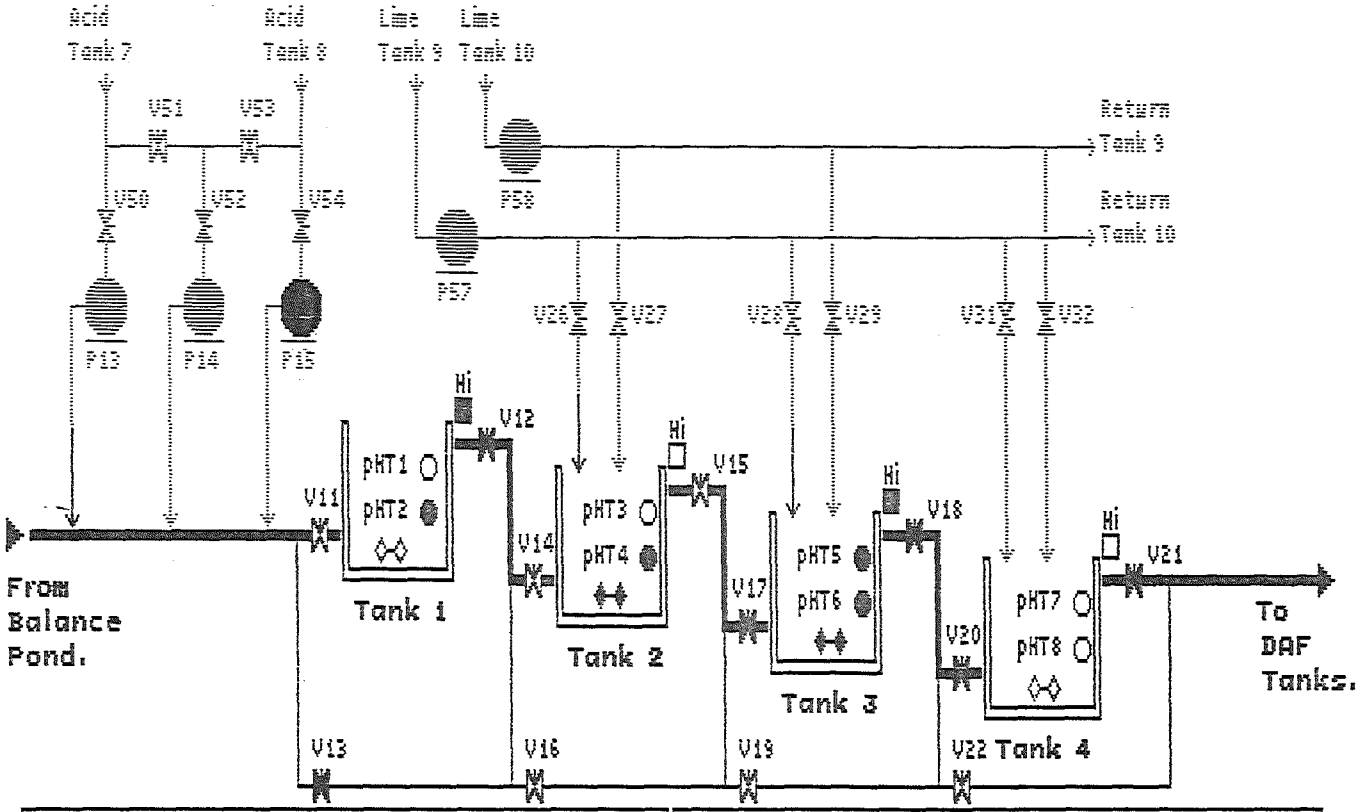
Appendix B

Sample Tesselator Mimic Diagrams for the CFM Effluent Treatment Plant

TESSELATOR HARD COPY

13-Apr-85 11:47:07

REACTION TANKS MIMIC



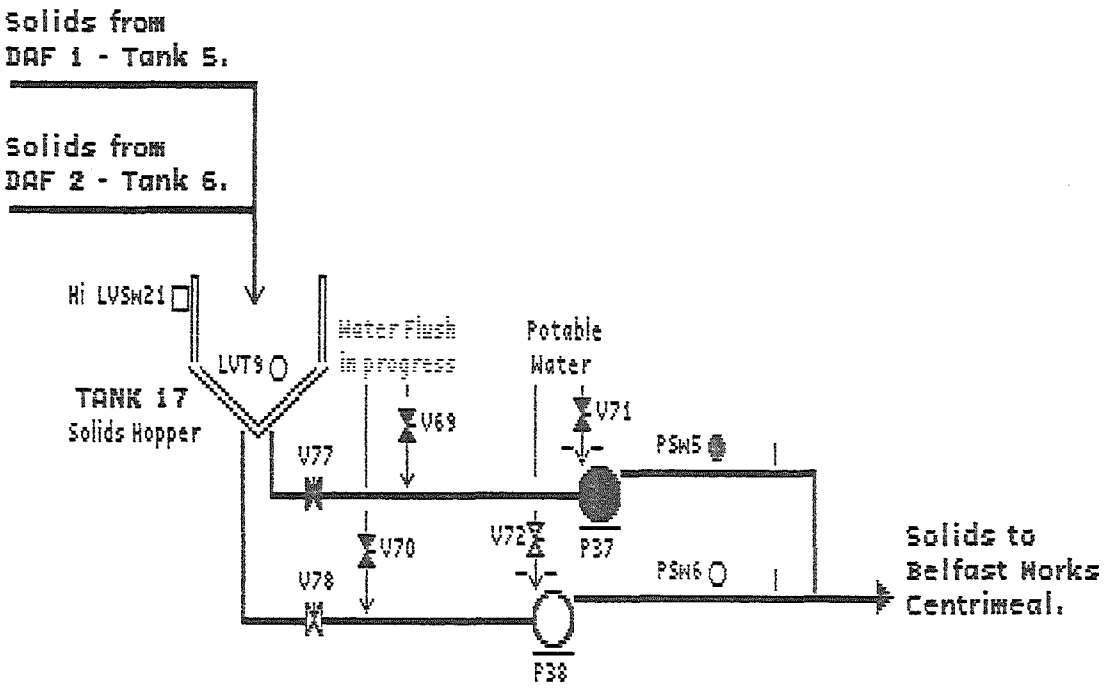
DESCRIPTION	DATA	DESCRIPTION	DATA
PLANT FEED	102 m^3/hr. (Max = 1200 m^3/hr.)	ACID FEED	658 ml/s (Normal =)
pH TANK 1	In Serv. pH 634 Stdby. pH 634	LIME 1 FEED	661 kg/s (Normal =)
pH TANK 2	pH 636 pH 636	LIME 2 FEED	663 kg/s (Normal =)
pH TANK 3	pH 643 pH 645	ACID / VOL EFFLUENT	FAL ltr/m^3 (Normal =)
pH TANK 4	pH 654 pH 650	LIME 1 / VOL EFFLUENT	FAL ltr/m^3 (Normal =)
		LIME 2 / VOL EFFLUENT	FAL ltr/m^3 (Normal =)

PROCEED PIC:3

TESSELATOR HARD COPY

13-Apr-85 12:02:43

SOLIDS HANDLING MIMIC



DESCRIPTION	DATA	DESCRIPTION	DATA
TANK 17 SOLIDS WEIGHT	316 T. (Max = T.)	CENTRIMEAL STORAGE	AVAILABLE FULL

STORAGE URGENTLY REQUIRED

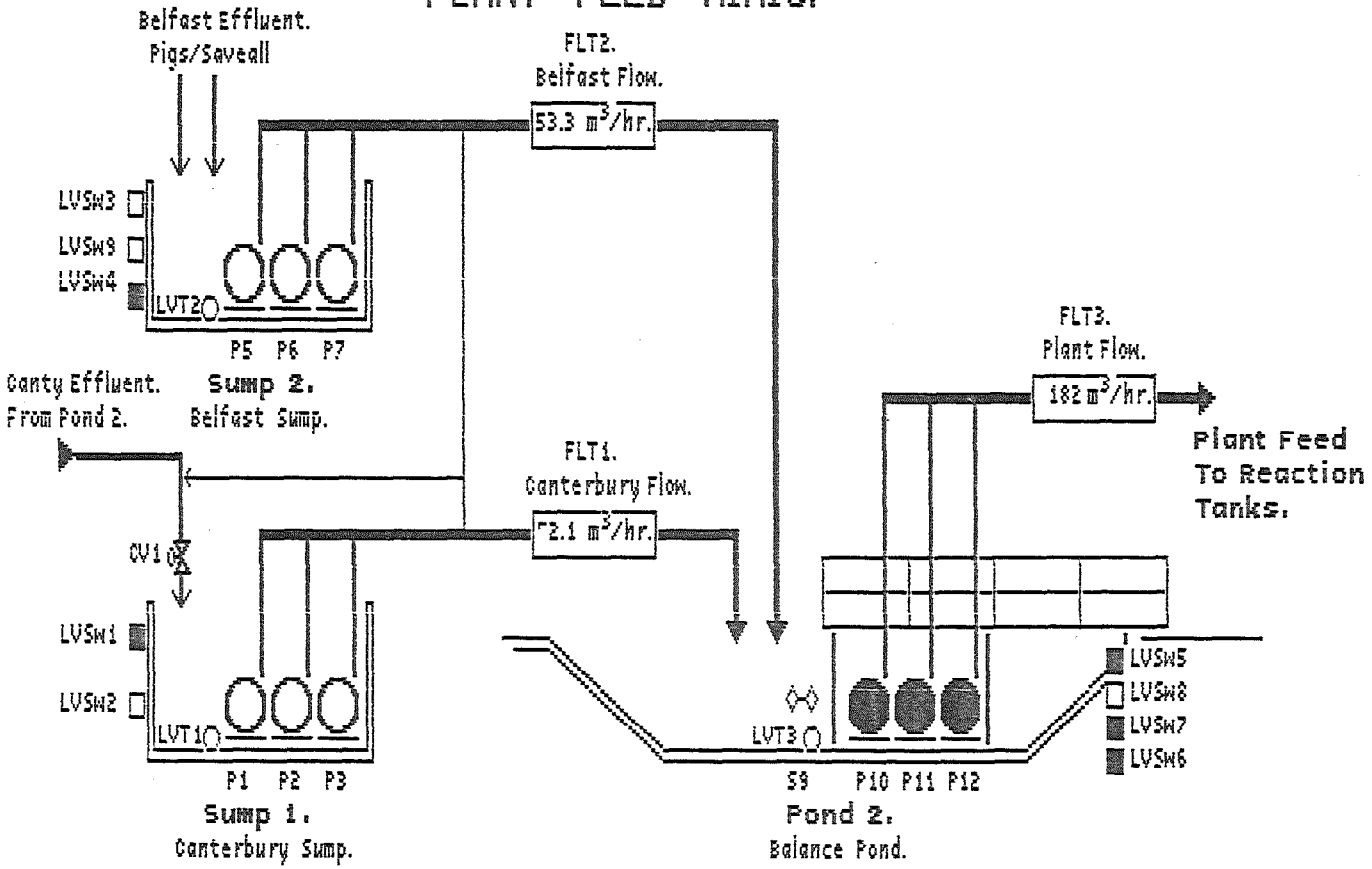
G.N.T.

PROCEED P10:28

TESSELATOR HARD COPY

13-Apr-85 11:52:09

PLANT FEED MIMIC.

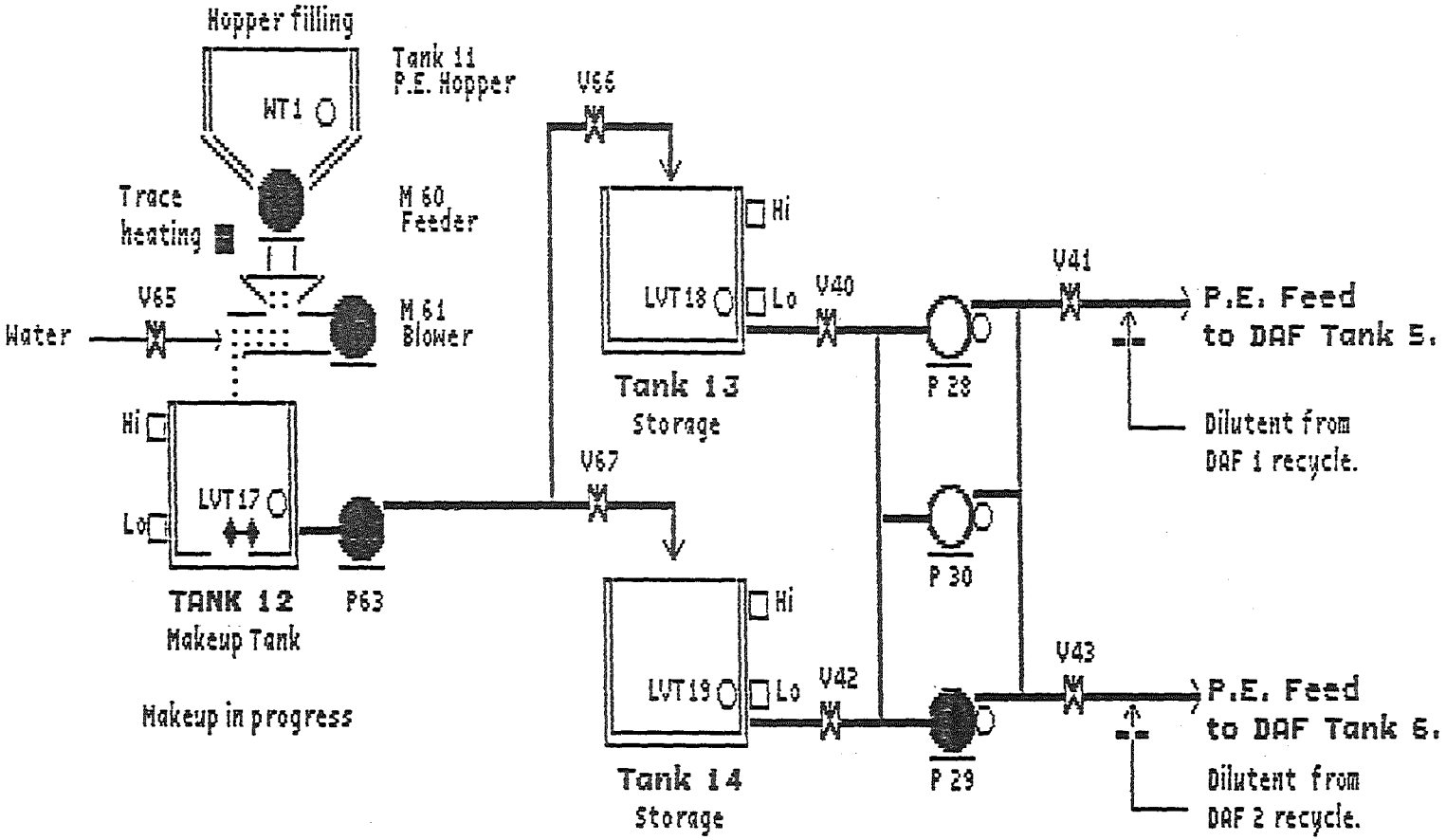


DESCRIPTION	DATA	DESCRIPTION	DATA
Sump 1 level Canty.	50.9 m (Full = 4.0 m)	Canty flow FLT1.	52.1 m^3/hr.
Sump 2 level Belfast	49.7 m (Full = 4.0 m)	Belfast flow FLT2.	53.3 m^3/hr.
Pond 2 contents Balance	48.6 m^3 (Full = 1500 m^3)	Plant Feed flow FLT3.	182 m^3/hr.

PROCEED P10:24

14:20:55

POLYELECTROLYTE MAKE-UP & ADDITION MIMIC



DESCRIPTION	DATA
P.E. HOPPER WEIGHT	75 kg. (Full = 100 kg)
TANK 12 CONTENTS LVT 17	1200 ltr. (Full = 1500 ltr)
TANK 13 CONTENTS LVT 18	5500 m. (Full = 7000 ltr)
TAMK 14 CONTENTS LVT 19	5700 m. (Full = 7000 ltr)
P.E. CONCENTRATION	0.09 % (Normal = 0.1 %)

POLYELECTROLYTE MAKEUP SEQUENCE			
STEP	SEQUENCE		DATA
1	Fill with water.		
2	P.E. addition.	Calculated time.	mins
		Elapsed Time.	mins
3	Stirring in progress.	Elapsed time	mins
4	Transfer to storage		

Appendix C

Listing of the Modula-2 Process Scheduler

```
DEFINITION MODULE Processes;
(* $SEG := 43; *)
(*****************************************************************
 *                                                               *
 *        MODULE Processes                                       *
 *                                                               *
 *        Interrupt driven real-time Process Scheduler.          *
 *        Process Termination added.                             *
 *        Priority assigned to processes. Always activate        *
 *        process with highest priority.                         *
 *                                                               *
 *        For SAGE IV Modula-2 ( Volition Systems ).             *
 *                                                               *
 *        Adapted from suggestions by D. A. Sewry and N. Wirth   *
 *                                                               *
 *****************************************************************)

EXPORT QUALIFIED StartProcess, StopProcess, SEND, WAIT, DELAY,
                 Awaited,      Init,        SIGNAL;

TYPE
  SIGNAL;

PROCEDURE StartProcess(P : PROC;  N : CARDINAL;  PRIORITY : INTEGER);
(* Start a process P with workspace size N and priority PRIORITY *)

PROCEDURE StopProcess;
(* Terminate the current process *)

PROCEDURE SEND(VAR S : SIGNAL);
(* Reactivate a process WAITing on S *)

PROCEDURE WAIT(VAR S : SIGNAL);
(* WAIT for some process to send S *)

PROCEDURE DELAY(numberOfSeconds : CARDINAL);
(* DELAY for some length of time *)

PROCEDURE Awaited(S : SIGNAL) : BOOLEAN;
(* Any processes WAITing on S *)

PROCEDURE Init(VAR S : SIGNAL);
(* Initialise S *)

END Processes .
```

```
IMPLEMENTATION MODULE Processes[1];

FROM SYSTEM     IMPORT ADDRESS, ADR, WORD, SIZE, TSIZE, PROCESS,
                       NEWPROCESS,   TRANSFER,   IOTRANSFER;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
FROM Program    IMPORT SetEnvelope, FirstCall;
FROM UnitIO     IMPORT UnitWrite;
FROM SYSTEM68   IMPORT ClearVector, SetPriority, Attach, Detach, Disable
FROM Standards IMPORT Time;
FROM Wides      IMPORT WIDE, MakeWide, IntToWide, Geq, Lss, Add, Mul, L

CONST
  ClockIVector  = 9;     (* clock interrupt vector *)
  EventINumber  = 36;    (* clock event number     *)
  ClockDevice   = 131;   (* clock device number    *)
  ClockPriority = 3;     (* clock priority number  *)
  StartClock    = 1;     (* clock start indicator  *)
  StopClock     = 0;     (* clock stop indicator   *)

TYPE
  SIGNAL = POINTER TO ProcessDescriptor;
  ProcessDescriptor = RECORD
    next   : SIGNAL;
    queue  : SIGNAL;
    cor    : PROCESS;
    prior  : INTEGER;
    start  : WIDE;
    ready  : BOOLEAN;
    WSP    : ADDRESS;
    Wsize  : CARDINAL;
  END;

VAR
  CLK,                                              (* clock process
  UserProcess : PROCESS;                            (* current user proces
  CLKWSP      : ARRAY [ 0 .. 499 ] OF WORD;         (* clock workspace
  T           : ARRAY [ 0 .. 1   ] OF CARDINAL;     (* timeslice duration

  MODULE SYNCHRONISE[4];
  (* Non-interruptable module to protect process ring *)

  IMPORT ADDRESS, TSIZE, PROCESS, NEWPROCESS, TRANSFER, ALLOCATE, DEALL
         SIGNAL, ProcessDescriptor, UserProcess, SetPriority, Disable,
  IMPORT WIDE, MakeWide, IntToWide, Geq, Lss, Leq, Add, Mul, Time;

  EXPORT StartProcess, StopProcess, SEND, WAIT, Awaited, Init,
         TimeSlice,    CurrentProcess,    DELAY;
  VAR
    CurrentProcess,         (* pointer to current process
    DELAYED,                (* list of DELAYed processes
    TOPPROC     : SIGNAL;   (* pointer to process with highest priority

    PROCEDURE FirstDelayedProcess () : SIGNAL;
    (*****************************************
     * Returns the first Process in the DELAYED *
     * queue .. uses "hot" loop to wait until   *
     * this process is ready to execute ..      *
     *****************************************)
    VAR
      FinalTime : WIDE;
      Hi, Lo    : CARDINAL;
      FirstProc : SIGNAL;
    BEGIN
      FinalTime := DELAYED^.start;
```

```
    REPEAT
      (* Busy loop until first waiting process is ready *)
      Time( Hi, Lo );
    UNTIL Geq( MakeWide(Hi,Lo), FinalTime );
    (* Remove first from DELAYed queue *)
    FirstProc := DELAYED;
    DELAYED   := FirstProc^.queue;
    FirstProc^.ready := TRUE;
    FirstProc^.queue := NIL;
    RETURN FirstProc;
END FirstDelayedProcess;

PROCEDURE StartProcess( P : PROC;   N : CARDINAL;   PRIORITY : INTE
VAR
    S0, S1, PREVIOUS : SIGNAL;
    PRIOR            : CARDINAL;
BEGIN
    S0 := CurrentProcess;     S1 := TOPPROC;
    (** Allocate workspace and process descriptor for new process **)
    ALLOCATE(CurrentProcess, TSIZE(ProcessDescriptor));
    ALLOCATE(CurrentProcess^.WSP,N);
    (** save the workspace size for later ( deallocation )**)
    CurrentProcess^.Wsize := N;
    IF S1^.prior < PRIORITY THEN
      (* new process has highest priority *)
      REPEAT
        S1 := S1^.next;
      UNTIL S1^.next = TOPPROC;
      PREVIOUS := S1;
      (* amend top priority pointer *)
      TOPPROC  := CurrentProcess;
    ELSE
      (* find correct place to insert descriptor *)
      PREVIOUS := S1;
      S1 := S1^.next;
      WHILE ((S1^.prior >= PRIORITY) AND (S1 # TOPPROC)) DO
        PREVIOUS := S1;
        S1 := S1^.next;
      END;
    END;
    WITH CurrentProcess^ DO
      (* link descriptor into ring *)
      next := PREVIOUS^.next;
      PREVIOUS^.next := CurrentProcess;
      ready := TRUE;
      prior := PRIORITY;
      queue := NIL;
    END;
    (** Start at PRIORITY -- disable interrupts **)
    Disable;
    PRIOR := SetPriority(PRIORITY);
    NEWPROCESS(P, CurrentProcess^.WSP, N, CurrentProcess^.cor);
    PRIOR := SetPriority(PRIOR);
    Enable;
    (** enable interrupts again **)
    S1 := TOPPROC;
    WHILE NOT S1^.ready DO
      (* search for ready process with highest priority *)
      S1 := S1^.next;
    END;
    CurrentProcess := S1;
    TRANSFER( S0^.cor, CurrentProcess^.cor );
END StartProcess;
```

```
PROCEDURE StopProcess;
VAR
  S : SIGNAL;
  P : PROCESS;
BEGIN
  IF CurrentProcess = CurrentProcess^.next THEN
    (** All processes terminated .. end of program **)
    HALT;
  END;
  S := CurrentProcess;
  REPEAT
    (** find next ready-to-run process **)
    CurrentProcess := CurrentProcess^.next;
  UNTIL CurrentProcess^.next = S;
  (** remove descriptor from ring  **)
  CurrentProcess^.next := S^.next;
  IF S = TOPPROC THEN
    (** process with top priority terminated **)
    TOPPROC := TOPPROC^.next;
  END;
  CurrentProcess := TOPPROC;
  (* find next ready-to-run process *)
  WHILE ((NOT CurrentProcess^.ready) AND (CurrentProcess^.next # TO
    CurrentProcess := CurrentProcess^.next;
  END;
  IF NOT CurrentProcess^.ready THEN
    IF DELAYED = NIL THEN
      HALT; (*** DeadLock ***)
    ELSE
      CurrentProcess := FirstDelayedProcess();
    END;
  END;
  P := S^.cor; (* Save process for transfer *)
  DEALLOCATE(S^.WSP, S^.Wsize);
  DEALLOCATE(S, TSIZE(ProcessDescriptor));
  TRANSFER( P, CurrentProcess^.cor );
END StopProcess;

PROCEDURE SEND( VAR S : SIGNAL );
VAR
  S0 : SIGNAL;
BEGIN
  IF S # NIL THEN
    (* a process is WAITing *)
    S0 := CurrentProcess;
    CurrentProcess := S;
    WITH CurrentProcess^ DO
      (* mark signalled process as active *)
      S := queue;
      ready := TRUE;
      queue := NIL;
    END;
    TRANSFER( S0^.cor, CurrentProcess^.cor );
  END;
END SEND;

PROCEDURE WAIT(VAR S : SIGNAL);
VAR
  S0, S1 : SIGNAL;
BEGIN
  IF S = NIL THEN
    (* first such process to WAIT *)
    S := CurrentProcess;
  ELSE
    (* add to existing queue *)
    S0 := S;
    S1 := S0^.queue;
    WHILE S1 # NIL DO
```

```
      (* search for tail *)
      S0 := S1;
      S1 := S0^.queue;
    END;
    S0^.queue := CurrentProcess;
  END;
  CurrentProcess^.ready := FALSE; (* deactivate process *)
  S0 := CurrentProcess;
  S1 := TOPPROC;
  (* find next ready-to-run process *)
  WHILE ((NOT S1^.ready) AND (S1^.next # TOPPROC)) DO
    S1 := S1^.next;
  END;
  IF NOT S1^.ready THEN
    IF DELAYED = NIL THEN
      HALT; (* DeadLock *)
    ELSE
      S1 := FirstDelayedProcess();
    END;
  END;
  CurrentProcess := S1;
  TRANSFER( S0^.cor, CurrentProcess^.cor );
END WAIT;

PROCEDURE DELAY (numberOfSeconds : CARDINAL);
CONST
  Sixty = WIDE( 10 .. 21 );
VAR
  S0, S1     : SIGNAL   ;
  Hi, Lo     : CARDINAL;
  FinalTime : WIDE     ;
BEGIN
  Time(Hi,Lo);
  CurrentProcess^.start    := Add( MakeWide(Hi,Lo),
                              Mul( MakeWide(0,numberOfSeconds), Sixt
  IF DELAYED = NIL THEN
    (* first such process to WAIT *)
    CurrentProcess^.queue := NIL;
    DELAYED := CurrentProcess;
  ELSIF Lss(CurrentProcess^.start,DELAYED^.start) THEN
    (* insert before first in DELAYED queue *)
    CurrentProcess^.queue := DELAYED;
    DELAYED      := CurrentProcess;
  ELSE
    (* find correct position *)
    S0 := DELAYED;
    (* insert in increasing order of start times *)
    WHILE (S0^.queue # NIL) AND Geq(CurrentProcess^.start,S0^.queue
      S0 := S0^.queue;
    END;
    CurrentProcess^.queue := S0^.queue;
    S0^.queue := CurrentProcess;
  END;
  CurrentProcess^.ready := FALSE; (* deactivate process *)
  S0                := CurrentProcess;
  CurrentProcess := TOPPROC;
  (* find next ready-to-run process *)
  WHILE ((NOT CurrentProcess^.ready) AND (CurrentProcess^.next # TO
    CurrentProcess := CurrentProcess^.next;
  END;
  IF NOT CurrentProcess^.ready THEN
    (* There MUST be one of these .. we just started it !! *)
    CurrentProcess := FirstDelayedProcess();
  END;
  TRANSFER( S0^.cor, CurrentProcess^.cor );
END DELAY;
```

```
  PROCEDURE Awaited( S : SIGNAL ): BOOLEAN;
  BEGIN
    RETURN S # NIL;
  END Awaited;

  PROCEDURE Init( VAR S : SIGNAL );
  BEGIN
    S := NIL;
  END Init;

  PROCEDURE TimeSlice;
  (*************************************************************************
   * Traverse DELAYED chain and if any DELAYed process has served its
   * set it ready-to-run .. this procedure does NOT actually restart
   * a process. This must be done explicitly by DELAY, SEND, or WAIT
   *************************************************************************)
  VAR
    Hi, Lo   : CARDINAL;
    TimeNow : WIDE     ;
    Process : SIGNAL   ;
  BEGIN
    IF DELAYED = NIL THEN
      RETURN;
    END;
    Time(Hi,Lo);
    TimeNow := MakeWide(Hi,Lo);
    Process := DELAYED;
    WHILE (Process # NIL) AND Leq(Process^.start,TimeNow) DO
      (* delete from list *)
      Process^.ready := TRUE;
      Process        := Process^.queue;
    END;
    DELAYED := Process;
  END TimeSlice;

BEGIN (* SYNCHRONISE *)
  ALLOCATE(CurrentProcess, TSIZE(ProcessDescriptor));
  WITH CurrentProcess^ DO
    (* Enter main program into ring *)
    next := CurrentProcess;
    ready:= TRUE;
    prior:= 1;
    queue:= NIL;
  END;
  DELAYED    := NIL;
  TOPPROC    := CurrentProcess ;
END SYNCHRONISE;

PROCEDURE CLOCK;
(* Driver for Clock Process *)
BEGIN
  Attach(Clock1Vector);
  LOOP
    IOTRANSFER(CLK, UserProcess, Clock1Vector);
    SYNCHRONISE.TimeSlice;
  END;
END CLOCK;
```

```
  PROCEDURE CLKinit;
  (* Start SAGE IV Clock *)
  VAR
    P:CARDINAL;
  BEGIN
    P := SetPriority(ClockPriority);
    NEWPROCESS(CLOCK, ADR(CLKWSP), SIZE(CLKWSP), CLK);
    P := SetPriority(P);
    T[0] := 0;
    T[1] := 6400;    (* 100 millisec timeslice *)
    TRANSFER(UserProcess, CLK);
    CurrentProcess^.cor := UserProcess;
    UnitWrite(ClockDevice, ADR(T), 0, 0, Event1Number, {StartClock} );
  END CLKinit;

  PROCEDURE CLKterm;
  (* Stop the SAGE IV Clock *)
  BEGIN
    UnitWrite(ClockDevice, NIL, 0, 0, Event1Number, {StopClock} );
    Detach(Clock1Vector);
    ClearVector(NIL, NIL, Clock1Vector);
  END CLKterm;

BEGIN (* Processes *)
  SetEnvelope( CLKinit, CLKterm, FirstCall );
END Processes .
```

```
MODULE ProducersAndConsumers;

FROM ProdCons  IMPORT BUFFER, Datum, Create, Full, Empty, Put, Get, Prod
                      Consume;
FROM Processes IMPORT StartProcess, StopProcess, SEND, WAIT, Init, SIGNA

VAR
  B : BUFFER;
  nonFull, nonEmpty : SIGNAL;

  PROCEDURE PRODUCER;
  VAR
    x : Datum;
  BEGIN
    LOOP
      Produce(x);
      IF Full(B) THEN
        WAIT( nonFull );
      END;
      Put(B,x);
      SEND( nonEmpty );
    END;
  END PRODUCER;

  PROCEDURE CONSUMER;
  VAR
    x : Datum;
  BEGIN
    LOOP
      IF Empty(B) THEN
        WAIT( nonEmpty );
      END;
      Get(B,x);
      SEND( nonFull );
      Consume(x);
    END;
  END CONSUMER;

BEGIN
  Init( nonFull );
  Init( nonEmpty );
  Create( B );
  StartProcess( PRODUCER, 200, 1 );
  StartProcess( CONSUMER, 200, 1 );
  StopProcess;
END ProducersAndConsumers .
```