UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

LAKEHEAD UNIVERSITY

The SIMULATION ENGINE : A Platform for Developing Industrial Process Knowledge Based Discrete Event Simulations

BY

Steven V. Falcigno

A THESIS SUBMITTED TO LAKEHEAD UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICAL SCIENCES

THUNDER BAY, ONTARIO

APRIL 15, 1997

0-612-33370-1

ACKNOWLEDGEMENTS

Many thanks to Dr Li, whose patience and guidance were of the utmost importance to the completion of my thesis.

A special thanks to my wife, Kim, who read my work and listened to me. Her understanding of technical writing was an incredible asset to the production of this thesis.

I would like to extend my gratitude to the operating staff at Avenor Inc. A special thanks to my father, Pat Falcigno, whose in depth understanding of the digester process gave me my first insight into the complexities of the industrial process.

Finally a special appreciation to my external examiner, Dr Meng, and my internal examiner, Dr Hasegawa for their input and recommendations.

ABSTRACT

This thesis is concerned with the design and implementation of an object oriented Simulation Engine capable of producing Knowledge Based simulations. The Simulation Engine provides a high-level Lisp-like script language for describing the process being simulated. As a detailed example, a Kaymr continuous digester is simulated. The Simulation Engine is made up of four distinct objects which have been implemented as individual programs in a Windows operating system.

This thesis describes the Simulation Engine in detail. The first chapter discusses the problem of complex knowledge based simulations in an industrial environment. A detailed example of an industrial process is provided. The second chapter provides an overview of the Simulation Engine in its design. The third chapter discusses the resources used to build the Simualtion Engine. The fourth chapter outlines the process of building the Simulation Engine. Chapter five demonstrates the Simulation Engine being used. The final chapter, chapter six, concludes with a discussion of advantages, disadvantages and possible enhancements for the Simulation Engine.

Table of Contents

List of Figures

CHAPTER 1 : INTRODUCTION

This paper discusses the production of a simulation development environment, which can be used to develop case-specific simulations. Originally I was planning on focusing my efforts to the creation of one simulation of a non-trivial industrial process. In order to accomplish this goal I studied the pulping process at a mill I was employed at as a Process MIS Specialist, during which discussions with the Digester Process Engineer (the chemical engineer in charge of improving the pulping process) led me to understand that simply applying the known mathematical formulae would not be sufficient to create a realistic digester simulation. Although proven methodologies exist for monitoring the production rate, quality control was limited to testing the finished product.

Further conversations with the operating staff at this site made me realize that the key to modeling any industrial process was to incorporate the knowledge of the operating staff. As well, it was apparent that such a simulation could be a useful tool for passing the experience of older operators on to younger ones without endangering either the people or the product. A training simulation would satisfy this goal.

By studying the pulping process I realized that the problem was that only an experienced operator really knew what was happening in the process. A computer programmer would never really be able to grasp the subtle nuances that an operator gathers over several years. To solve this problem, a highly flexible model would be required, one that easily allowed operator input.

Rather than focusing my efforts on a case-specific simulation only good for modeling a single industrial process, I decided to create a simulation development environment that could be used to build training models by providing a simple, inexpensive tool for collecting and simulating the existing operator experience.

To accomplish this goal I chose to work on a microcomputer running Windows 3.1. Borland C++ was chosen as the development language to allow an object-oriented programming solution to this problem. In order to permit the gathering of experience into the simulation generator I decided to use the *knowledge based* simulation paradigm.

This chapter discusses industrial process simulations in general, the continuous digester process, knowledge based simulations and finally presents a motivation and outline for this paper.

Section 1.1: Industrial Process Simulation

Simulation for industrial type systems involve complex chemical and mechanical processes that cannot be modelled properly in controlled environments. Although the science of the process may be understood and the chemical reactions and mechanical processes can be reduced to formulae, the human element adds a factor that is not easily measured. Despite this difficulty, most simulation efforts revolve around producing a set of mathematical equations that model the process. The ultimate goal of these is to produce a model which can do one of two things.

First, the exact models can be manipulated to predict the results of changing factors in the process. This type of simulation can then be used to conduct tests which, if successful, can then be tried in the actual process. This type of simulation needs to understand the complex equations and reactions happening at the molecular level.

These models can then be integrated into advanced control models which provide automated controls for the industrial process. Having accurate automated controls can significantly reduce costs by fine tuning the use of materials and optimizing production. As a result, the majority of simulations seem to be about adding intelligence to the process.

The second use of these simulations is to train new operators in the industry by providing a realistic interface to a modelled industrial process. It is vital that operators receive training that does not put the process and the people around the process at risk, and that extensive costs are not incurred during the training phase of an operator's career. This type of simulation does not require a complex type of understanding of the process - rather, it needs an intimate understanding of the history of the process. The operator needs to learn what has happened or could have happened.

The problem with simulations based purely on mathematical formulae is that they generally try to reduce the industrial model to a *scientific model.* The human element, the instincts and experience of the operators, is largely ignored. That is, simulations generally do not take the operator in mind, and since the operator is an intrinsic part of any industrial process the simulations have a tendency to fail, produce sub-optimal results or be only correct in specific conditions.

As noted in the paper by Weymouth and Sztrimbely (1990) it is the operators that can tell you how things are run. Their strategy was to incorporate operators, computer staff and engineering together in a process they called *knowledge engineering*. This knowledge was combined with artificial intelligence (AI) techniques to create a decision making model, capable of scheduling different events within the process. Weymouth and Sztrimbely also noted that a primary concern was to get the experience of the older operators in a usable form for the less experienced ones.

In order to simulate real-world processes I needed to understand the process to be simulated. In order to accomplish this goal a non-trivial process was selected for study.

Section 1.2: Continuous Digester Industrial Theory

The non-trivial process I selected to study was the Kamyr continuous digester. The digester process was available at my work site. It is generally accepted as a complex process. The following sub-sections discuss the kraft pulping process and provides an outline of some of the simulation work done in this field.

Section 1.2.1 : The Continuous Digester Process

Kraft pulp is a porridge of wood fiber which has all the lignin (the bonding agent of wood) removed through a chemical process called pulping. The remaining fiber can be meshed together and bleached to make a very bright white product. Kraft pulp is used in the production of paper products ranging from tissue paper to Kodak picture backing. This process has been in existence since the late 1800's, and has grown out of a great deal of empirical study.

To learn about this process I first examined the general methodology behind the process of making kraft pulp (Smook,1994). This process extracts the lignin from wood using a highly caustic chemical cooking process which combines pressure, chemical and heat. The chips are soaked in active alkali chemical, called *white liquor*, and forced through a vessel called a digester. In the Kamyr continuous digester this is a single unit - other kraft processes use *batch* processing which involves multiple tanks.

As the chips move through the digester it passes through heating zones, cooking zones, cleaning/washing zones and finally through a blow unit which sends the de-ligined pulp to the next phase (diffusing). Each zone pushes the cooking chemical through an internal tube out through a screen which permits the chemical through but not the wood chips. The chemical is extracted out through special drains, cleaned and returned to the process.

The amount of time that the chemical and temperature is exposed to chips inside the digester determines the quality of the product. The quality of the product is a measurement of lignin, called a K-number (in the European market a *kappa* number is used). This value cannot be measured inside the digester, so other techniques have been developed; most modeling and simulation studies with digesters focus on solving this problem (this is discussed more in section 1.2.2). A simple diagram of the chip to digester process is given in figure 1.1 (Smook,1994):

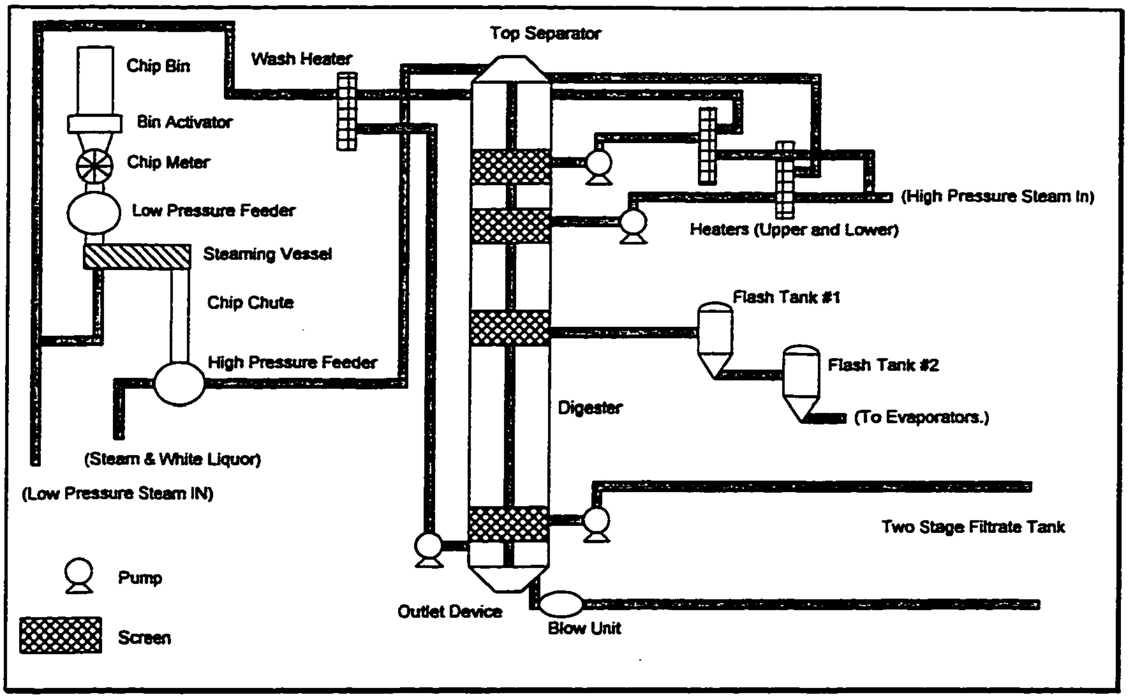


Figure 1.1 : Kamyr Continuous Digester System

The different components in the digester process are:

Chip Bin - This container stores chips from the wood yard, and insures that chip supply during wood yard downtime.

Bin Activator - The activator assures a uniform flow of chips from the chip bin.

Chip Meter - A rotating star feeder with seven pockets possessing a measured volume of chips per revolution. The chip meter speed determines the first factor of production (chip flow).

Low Pressure Feeder - A rotating star feeder that acts as a seal against the pressure in the steaming vessel.

Steaming Vessel - A sealed screw conveyor, providing the initial steam bath of the chips. Its purpose is to raise the chip temperature to approximately 250 deg. F.

Chip Chute - The chute provides a passage from the steaming vessel to the liquor pool in the high pressure feeder.

High Pressure Feeder - The high pressure feeder combines the chips with the steam/white liquor mixture and sluices the mixture to the top separator.

Top Separator - A sealed screw conveyor, which evenly moves the sluiced chips into the digester vessel. It includes a level indicator which alarms if the digester is full (normally a digester runs between 60 - 80% full).

Heaters - Heaters are used to increase the temperature of re-circulated liquor. The wash heater takes liquor from the bottom of the digester, re-heats it and injects the liquor at the top using low pressure steam. The upper and lower heaters are similar, in that they reheat cold liquor, but take the liquor from the middle areas of the digester.

Flash Tanks - These tanks reduce the steam/chemical temperature to room temperature. Waste chemical extracted here is sent to the recovery process.

Outlet Device - Provides a scraper at the bottom of the digester to uniformly release cooked chips to the blow unit.

Blow Unit -The blow unit passes cooked chips to the high density storage tank, ensuring temperature and pressure factors do not damage wood fibers.

Filtrate Tank - As part of the cleaning process, dilution and filtrate are added to the chips. The washed out chemicals/chip solution is filtered out and passed onto the diffusion washer, and onto the high density storage tank.

The chemical reactions that take place in the digester are a result of the active alkali chemical, usually referred to as white liquor. This chemical is a combination of sodium hydroxide (NaOH) and sodium sulfide (Na_2S) (Smook,1994). As the chips travel through the digester the chemical digests the wood lignin, leaving the wood fibers necessary to make paper products. The chips flow from the top of the digester (top separator) to the bottom (outlet device). Through this flow is a water/liquor wash which breaks down the lignin. Spent (used) liquor is removed via the flash tanks and the filtrate tank. This spent (or black) liquor is sent to the evaporators, where the recovery of re-usable chemical can be done.

An interesting note in the behaviour of the digester is that the flow of chips is controlled through pressure and the rotation per minute of the chip feeder. The flow of cooking chemical is through the chips - the chemical washes around the chips. Gravity and pressure forces the heavier, chemical filled chips down to the outlet device. The chips are forced up against screens along the sides of the digester, squeezing out the chemical. An extraction screen is a fine mesh (5 mm holes) that the chips are pushed against. The chips, except for some very small particles, cannot pass through the screen. The extracted chemical can be cleaned and reused. An extraction screen is given below:

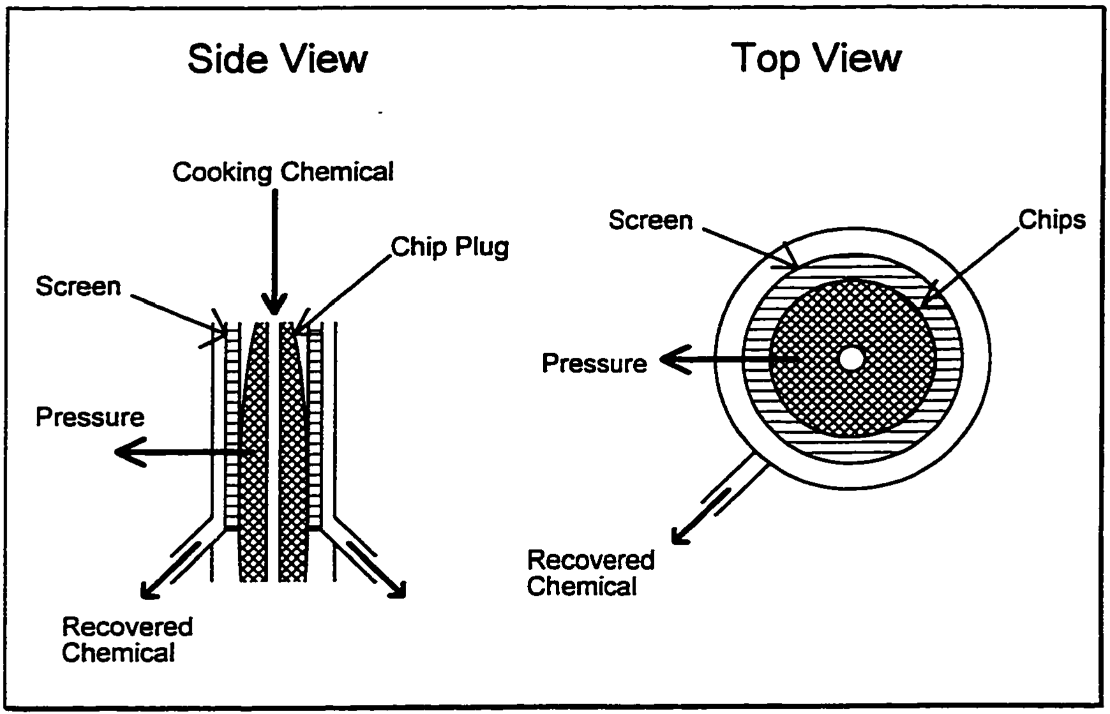


Figure 1.2 : Digester Extraction Screens

The importance of the screens cannot be overlooked. If a screen becomes plugged in the digester both temperature and pressure will be impossible to control. This will result in poor quality product and low productivity. Excess screen plugging will stop the liquid flow through the chips from the center to the outside of the chips plug, reducing the cooking (or washing in the case of extraction or washing zone) of the exterior region and destroying the interior chips.

By discussing the digester operation with the operators and supervisors of a typical disgester, an approximate timeline was developed:

TIME (hours)	EVENT	NOTES
0:00	chip bin	Storage of chips to insure steady supply.
0:01	agitator	Shakes chips into chip meter for uniform distribution.
0:02	chip meter	Pressure seal. RPM determines production. 7 compartments which rotate and deposit chips.
0:03	low pressure feeder	Steam injected into chips. Pressure raised to 40-60 #.
0:04	steaming vessel	Gases/air forced out of chips. Temperature raised to 240-250 deg. F. Moisture uniformly distributed.
0:08	high pressure feeder	Adds cooking liquor to chips. Increases pressure to 180# at top separator.
0:10	top seperator	Pressure at 180#. (sealed in) Temperature 250 deg F. Screen used to extract liquor.
1:00	impregnation zone	Chips soak up liquor. Pressure at 165#. Temperature at 240 deg F.
3:00	upper cooking	Screens extract liquor for re-heating. Temperature at 310 deg F. Cooking (lignin breakdown) begins.
4:00	lower cooking	As upper, but temp 330 deg F. K-number based on temperature here. + 1 deg F == -0.5 K-number.
5:00	cooking zone	chips left to cook (temperature increases +8 deg)
7:00	extraction	Temp cooled to 280-300 deg F. Liquor (chemical) extracted. Chemical sent to flash tanks, recovery.
7:30	washing zone	Chips rinsed with filtrate (dirty water). Pressure 240# Temperature 265 deg F Overflow filtrate extracted using screens.
8:00	scraper	Breaks up chips for uniform distribution.
8:01	outlet device	Cool wash (temperature 170-190 deg F.). Pressure lowered - 240 drops to 90 rapidly. (called blow effect).
8:05	diffuser washer	Washing continues. Pressure : 40 #
8:15	atmospheric diffuser	Washing, pressure reduced to atmosphere.
8:30	blow tank	Provides storage/feed for bleaching process.

Figure 1.3 : Digester Process Timeline

The wood chips pass from zone to zone, first being impregnated with cooking chemical (white liquor), then being heated, then allowed to cook, and finally the chemical is removed and the wood pulp blown out the bottom of the digester. In order to understand the different zones in the digester process, examine the following diagram illustrating the zones and time line events:

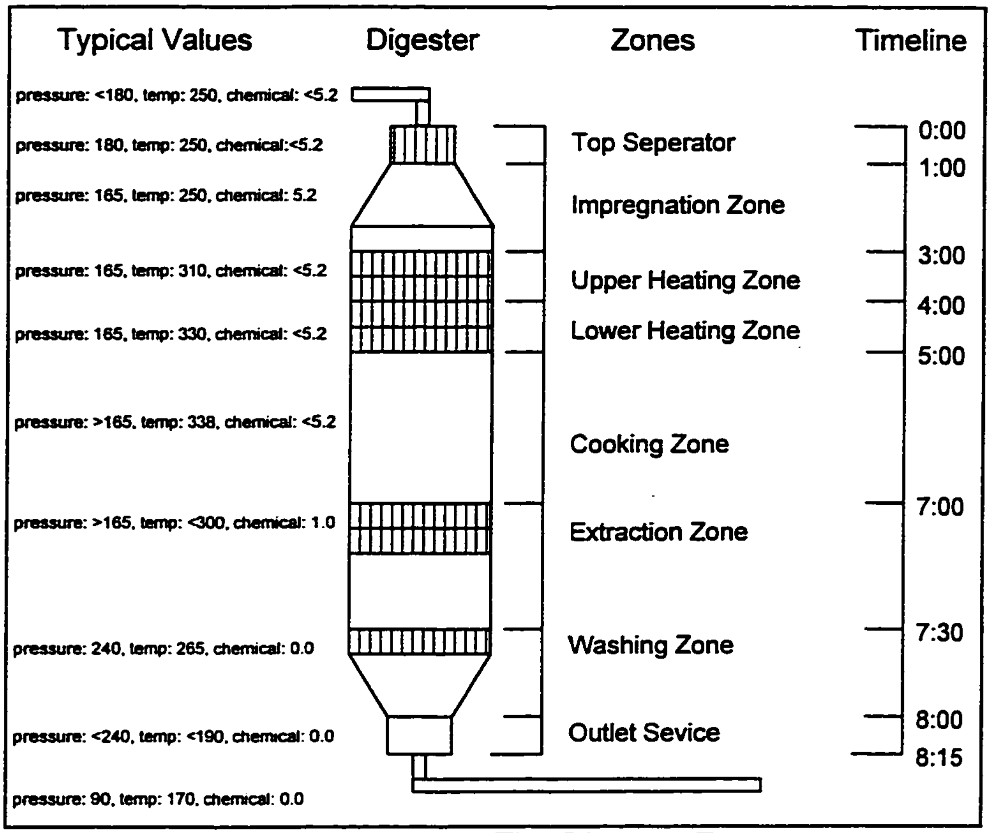

Typical Values
Digester
Zones
Timeline
pressure: <180, temp: 250, chemical: <5.2
pressure: 180, temp: 250, chemical:<5.2
pressure: 165, temp: 250, chemical: 5.2
pressure: 165, temp: 310, chemical: <5.2
pressure: 165, temp: 330, chemical: <5.2
pressure: >165, temp: 338, chemical: <5.2
pressure: >165, temp: <300, chemical: 1.0
pressure: 240, temp: 265, chemical: 0.0
pressure: <240, temp: <190, chemical: 0.0
pressure: 90, temp: 170, chemical: 0.0
Top Seperator
Impregnation Zone
Upper Heating Zone
Lower Heating Zone
Cooking Zone
Extraction Zone
Washing Zone
Outlet Sevice
0:00
1:00
3:00
4:00
5:00
7:00
7:30
8:00
8:15

Figure 1.4 : The Digester Zones

The formulae and human elements of the digester process are discussed in more detail in section 5.2. Even this preliminary work shows how a great deal of operator experience is used to control the digester process. One of the top concerns is that this experience is passed onto the next generation of operators.

Section 1.2.2 : Continuous Digester Simulations

Generally, digester simulations focus on mathematically modeling the digester process. The Kamyr equations are available (see section 5.2) and can be used to predict the quantity of production, but not the *quality* of production. That is, the production rate can be determined by how fast wood chips are fed into the system, but the amount of lignin digestion that takes place is harder to predict.

The first modeling method examined was by Allison, Dumont, Novak, and Cheetham (1989) who examined the exposure time the wood chips experience within the digester. A digester is a closed vessel, full of the cooking liquor. It is very difficult to determine the actual level of the chips, and thus difficult to determine how high the chips are (the size of the *chip plug*). In this study Allison et al used data collected from *strain gauge meters* to calculate the position of the chip plug. These meters have a blade which sticks out at a right angle from the digester's internal wall. By measuring the pressure or strain put on these gauges the position of the chip plug can be approximated. In this study the premise was that the exposure time would determine the quality, and this time could be determined by using strain gauges and a complex algorithm. This method does work, assuming that all other factors can be kept constant. Human intervention was still required when unusual instances occurred. The study gave the example of a chip plug hang-up, when sections of the chip plug get stuck on the extraction screens, thus never reaching the gauges and giving the false reading that no chips are coming down. This type of instance is exactly why the human being needs to be part of the solution.

Another interesting approach to modeling the digester was based on a database of information. This study by Michaelsen, Christensen, Lunde, Lundman,and Johansson (1992) focuses on a quality control variable (the kappa number) which is a measure of the quality of pulp. The model tries to keep the kappa number constant, allowing the other factors in the model to change. (This is similar to the keeping the H-Factor constant in section 5.2. The premise is that since production is governed by chip flow the only issue is quality of product. The kappa number is an European measurement of pulp quality.)

This study uses a complex linearization (a set of partial derivatives) combined with feedback from the control system to model the digester. - the algorithm uses older information to correct itself over time. The theory is that the model will become more accurate over time.

This particular study was interesting because of the database of information that was retained. The complexity of the mathematical model restricted the model to the site being modelled, rather than being able to expand to the general case. This approach of adapting with respect to history has a major flaw. If the process changes significantly a new set of non-trivial equations will be required. Complex processes such as the Kamyr continuous digester require a more advanced type of solution in order to facilitate a correct model.

Section 1.3: Knowledge Base Simulation

After examining the environment to be modelled - an environment where people are part of the simulation, where poorly understood chemical reactions take place and there is a constant effort to improve the process to remain competitive - I decided that the best solution to investigate would be the use of a database of behaviours, combined with the simpler Kamyr equations all the while keeping the operator in the process. This decision was influenced by Nielson (1991) who discusses the three instances where a math model may fail:

1. A poorly understood decision process.
2. A human in the loop - an operator as a required part of the process.
3. Situations where experimenting with the decision making process are made frequently.

All three of these apply to the complex digester process. To resolve this complex problem a different sort of simulation is required; a *knowledge-based* simulation is the answer.

The knowledge based event simulator requires that a database of information about the process is used. This information can be stored in many different formats. The most useful format encountered was the use a simple programming language to describe the different behaviours in the model. Hu and Rozenblit (1991) use a Lisp like language to describe their rule database. This technique should allow the most flexible rule database because anything should be describable using a full language.

A possible implementation of a knowledge based simulation would include the traditional simulation modules: event processing, a user interface and a simulation state. Additionally a knowledge base, a processor dedicated to implementing the rules in the knowledge database would be required to evaluate any general rules, probabilities or dynamic formulae that occur. An information flow diagram illustrates how information would be passed to the different parts of the simulation:

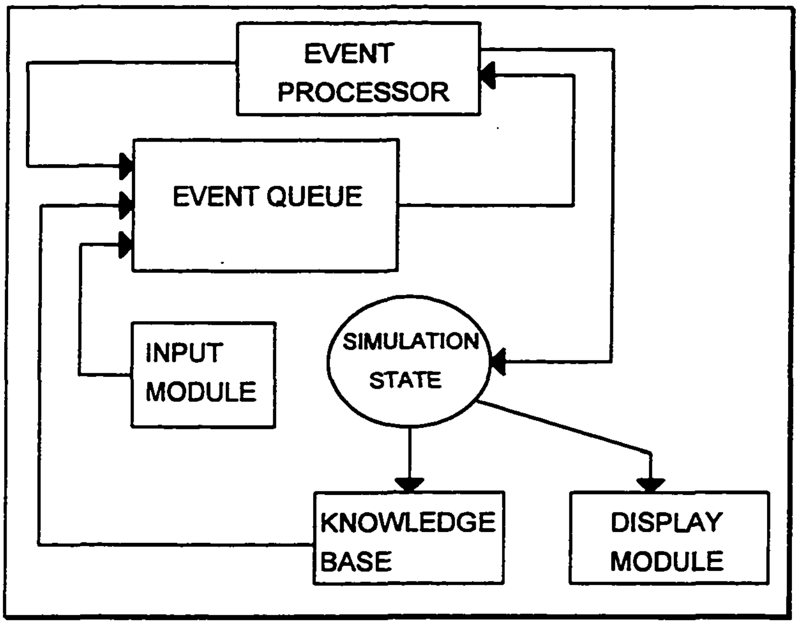


Figure 1.5 : An Event Driven Simulation with a Knowledge Base

As every quantum of time passes, the event processor examines the queue. Events that are scheduled to be executed (to a maximum number, to ensure the user input module does not spend too much time suspended) are removed from the queue and processed. At this time the event processor will read and write to the variables stored in the simulation state to reflect any changes. Once the simulation state has been updated, the event processor will add any new events that are created by the event currently being processed.

After the event processor has completed, the rules processor will evaluate the simulation state using a knowledge database. This process is implemented by taking values stored in the simulation state, checking if particular conditions exist and creating an event for each condition that requires it. The knowledge database should be dynamic, to allow run-time adjustments, using a simple internal language capable of primitive Boolean and mathematical operations.

Each module or processor should be discrete both in concept and design. If practical, each component should be implemented as a separate task in a multitasking operating system. The interaction that does take place between components uses an object message passing system with clearly defined responsibilities. The final result of the knowledge based simulation should be a database capable of working with experienced operators.

Section 1.4: Motivation and Thesis Outline

After examining the industry process of creating kraft pulp I decided that I could either build a complex simulation of a particular digester or develop an engine that could be used to produce knowledge based simulations. I felt that a simulation engine would be more useful both to industry and to the development of my computer science skills. I call this simulation development environment the Simulation Engine.

My goal was to create a graphic, PC based, simulation development environment capable of supporting knowledge based simulations. The focus of these simulations should be trainer oriented instead of predictive, since these simulations will be used to assist in the educating of new operators. The simulation should be capable of supporting simple math models/formulae, but contain enough intelligence that it can demonstrate qualitative behavior of complex processes as well.

The development environment must support dynamically configurable simulations allowing experienced operators to reconfigure simulations quickly and easily. The design should allow room to develop concepts more in depth as more information becomes available. The environment should allow simplifications so that work can focus on the known factors and not be halted on the unknowns.

The Simulation Engine takes full advantage of the Windows 3.1 operating system, utilizing its limited multitasking ability to breakdown the simulation engine design. It implements inter-process communication using native Windows protocols and interfaces with other programs not a part of the simulation. The engine provides a simple developer's graphical interface, which is configurable by non-computer oriented personnel. Since it was designed to run on a PC platform, it is cost effective. In order to test the knowledge based simulation generator simple digester simulations were developed.

The Simulation Engine conceptualization is described in chapter 2. Here the concept of a knowledge based simulation development environment is molded into an object-oriented message passing model. Chapter 3 discusses the various programming tools used to build the actual program such as Borland C++, the DDE(Dynamic Data Exchange) protocol and the Windows operating system.

Chapter 4 explains some of the more in depth implementation details of the Simulation Engine. The key concepts include the language the knowledge database is implemented in, the graphic tags that interface to the operator, the variable handler used to store the simulation data and the DDE client/server class developed to allow inter-process communication.

Actual simulations are presented in chapter 5, including the development of a simple pump simulation used to explain the development process. This chapter also discusses interfacing to other Windows applications and gives an example of how this might be done.

The final chapter discusses the problems with the Simulation Engine, the desirable enhancements and the possible future applications of the Simulation Engine.

Chapter 2: OVERVIEW OF THE SIMULATION ENGINE

This chapter examines the simulation engine model. It defines the roles and responsibilities of each of the major components required in the design phase. It also introduces the message passing mechanism required between the different tasks. This chapter is meant to give an overall understanding of the simulation engine.

Section 2.1: The Object-Oriented Model

The simulation engine breaks down the process of simulating into four objects. Each of the entities can be classified according to the different classes defined by Budd (1991). The simulation model is illustrated in figure 2.1.

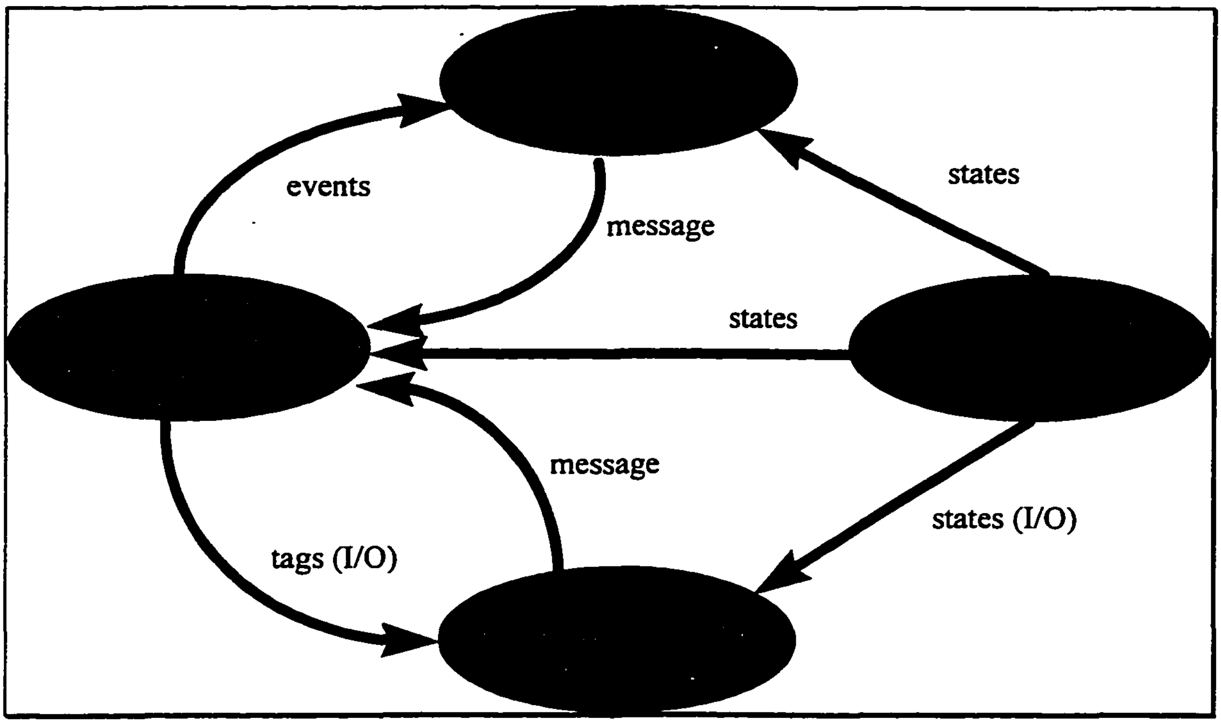


Figure 2.1: The Simulation Engine Model

The first object in the model is the Simulation State. This entity is a storage object or data manager. It keeps track of the values or states that represent the simulation. Other programs, including tasks not a part of the simulation engine, can request specific simulation states. These tasks can send assignment messages to the simulation state to change stored values.

The next object is the Input/Output handler. This program is responsible interacting with the user. It provides output onto a view screen in response to I/O messages received from other tasks. Input is provided using mouse and keyboard by interfacing with Window's messaging system (this is covered in more detail in section 3.1). This entity is a combination of source and viewer object. The reasons for this hybrid are covered in section 2.4.

Most of the I/O messages come from the Knowledge Base. This object is responsible for processing the rules that specify the simulation. It receives messages from other tasks which trigger it to load, parse and execute different script files. These scripts can then generate more communication among the different entities. This class can viewed as a storage class (it stores the different scripts) or a facilitator as it interfaces between the other tasks.

Some script files need to be run on regular intervals. This is accomplished using the Event Handler. The Event Handler message's are called events. As each event is received this object stores them in a queue of script files to be executed. When a timed interval occurs, it checks each item in its queue and executes them by sending a message to the Knowledge Base as required. This class generates information and can be viewed as another source object.

By working together these entities can simulate both simple and complex processes. The aspects of storage, interface, process and automation have been taken into account. The result is a highly flexible simulation engine.

Section 2.2: Message Passing System

The simulation engine uses a standardized messaging system between its object-tasks. To understand how the simulation engine works, we need to understand the different types of messages that are passed from task to task.

Exactly how the messages are passed will be discussed in more detail in section 4.5. Each message consists of two text strings (normally referred to as the *topic* and *item*). This section covers the structure and purpose of those strings.

The simplest message in the simulation engine is the *state*. This message comes from the Simulation State and consists of a variable name and a floating point value (stored in a text string). It is sent out in response to a request; the client process sends a request in the form of a variable name to the Simulation State and the Simulation State responds with a *state* message.

The state message is complemented by the *assignment* message. This message sends the variable name in the first string and a floating point value in the second. The Simulation State can then store the new value and make it available for the other tasks in the simulation.

The Event Handler and I/O Handler exchange messages with the Knowledge Base. If the first string is a *message* the Knowledge Base will queue the contents of the second string (which should be a script filename) to be executed. If the first string is anything else, we assume a state has arrived. The name is derived for the first string and the data from the second. Note that the string "message" can never be a variable name because of this.

The message *event* is the first non-trivial message. Valid messages are shown in figure 2.2.

First String	Second String
queue	<ID string>, <script name>, <iteration count>
pulse	nil
remove	<ID string>

Figure 2.2: The Event Message

When an event message has a first string of *queue*, the Event Handler updates or adds a job to the queue list, based on the unique ID string. If the first string is *remove*, the job with the corresponding ID string will be removed from the job queue. Finally, if the string is *pulse*, the Event Handler will process its job queue.

The most complex communications in this model is to the I/O Handler. State messages arrive and are matched with the first string with any tags storing that variable. The value will be updated to the contents of the second string and the user display is refreshed. More complex I/O messages, usually received from the Knowledge Base as part of a script executing, force changes in the screen's display by altering display elements called tags (discussed in section 4.2). Figure 2.3 shows what the messages look like:

First String	Second String
refresh	<tag ID> or refresh
smartgraphic	<tag description>
simplegraphic	<tag description>
simpletext	<tag description>
simpleregion	<tag description>
niltag	<tag ID>
flushtags	nil
smarttext	<tag description>
smartpopuptext	<tag description>
simplepopuptext	<tag description>
changeregion	<tag ID>,<new x,y,dx,dy>
changeregioncolor	<tag ID>,<color>
changegraphic	<tag ID>, <bitmap filename>
<default>	<data to be displayed>

Figure 2.3 : The Tag I/O Message

For more information on the various tag descriptions see appendix III. Most of the first string possibilities create different tags. The exceptions are *refresh*, which forces the I/O Handler to request a state message for a particular tag; *niltag*, which removes a tag; *flushtags*, performs a *niltag* on all display elements; *changegraphic*, allows us to update/animate a graphic image; and the default, if none of the others apply the first string is assumed to be a tag ID and the second is assumed to be data to be displayed by that display element.

The messaging system is critical to the function of the simulation engine. In order for the simulation to run, protocols must be agreed between the different tasks in the simulation.

Section 2.3: The Details of the Model's Objects

Section 2.3.1: The Simulation State

The Simulation State is an object which manipulates the values or states of the simulation. Other tasks in the simulation send requests for states to this program, which looks up the value and returns it. This object also accepts assignments, which it uses to alter its stored variables. How the simulation works is described in figure 2.4.

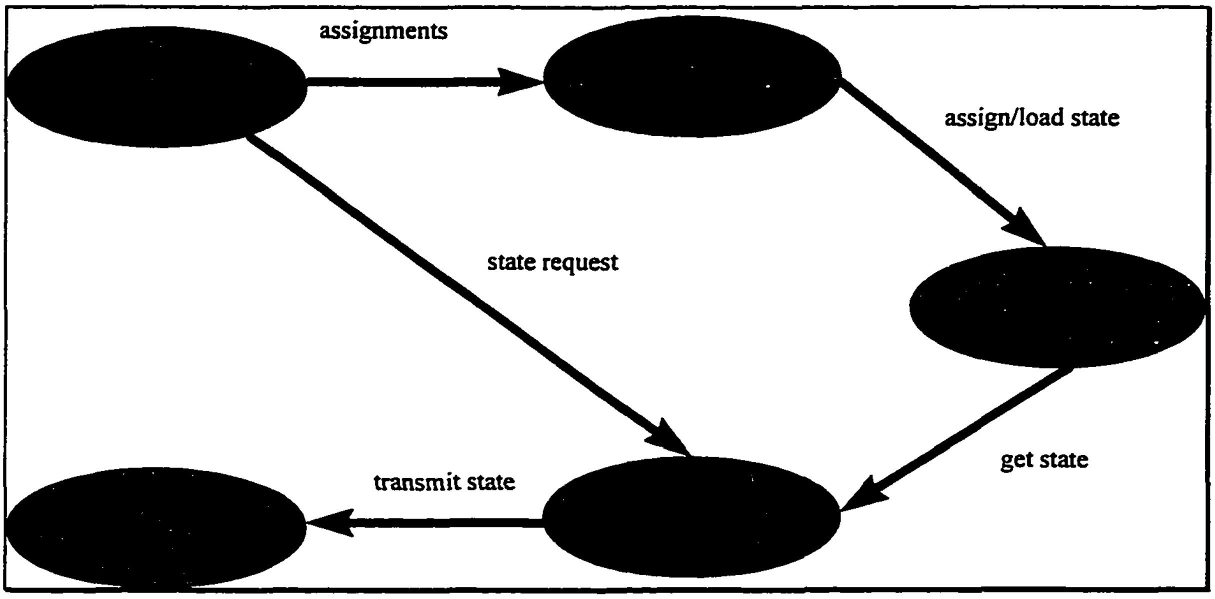


Figure 2.4 : The Simulation State

The core of the simulation state is a variable handler which maintains simulation variables. States can be assigned (to existing) or loaded (new states). These values can be retrieved using a *get* message. This entity is discussed in more detail in section 4.3.

Of all the tasks in the Simulation engine the Simulation State is the simplest. Like all simulation tasks it is capable of peer-to-peer type of communications, but the nature of state storage allows it to behave as a pure server type entity. Future enhancements to the simulation entity will require the task to take a more active, peer-to-peer type role.

Section 2.3.2: The I/O Handler

The user interface is made up of two important components, the output section and the input section. These two sections are combined by the end user to create the Input/Output Handler. The user is the source/sink object which takes messages from the Output Section and generates messages to the Input Section. Figure 2.5 illustrates the I/O Handler model.

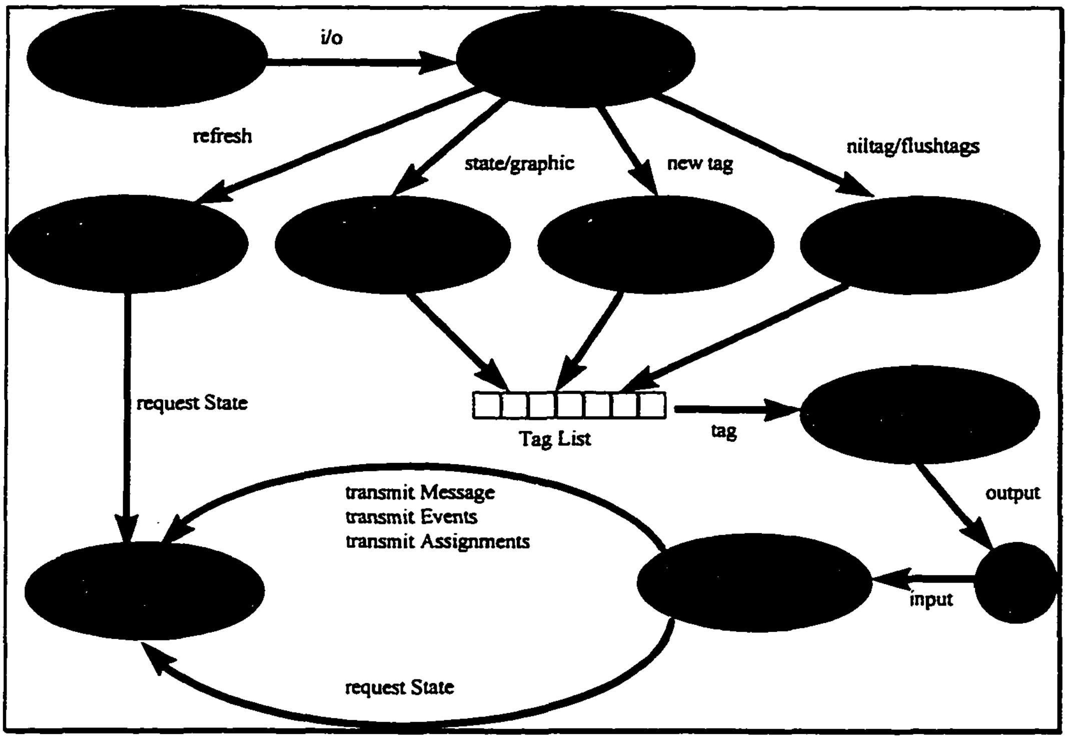


Figure 2.5 : The I/O Handler

The original simulation model separated the I/O Handler into two tasks, the Input Handler and the Output Handler. Although this is a reasonable idea, the restrictions of Windows programming required that the display screen and input routines be contained in the same program, thus the input and output was combined to create the I/O Handler - a hybrid object that both facilitates and views information.

The I/O Handler interacts with users via an object called a Tag (discussed in detail in section 4.2). A tag is an entity which can be displayed (as a value or picture) and utilised to get user input. A list of tags is maintained by the I/O Handler. This list represents the current state of the user's interaction with the simulation. The Output section will refer to this list to re-draw the screen and to generate state requests. By selecting a tag the user can generate messages, events, and assignments. These are communicated with the other tasks in the simulation using the DDE OUT section.

The other processes in the simulation communicate with the I/O Handler by sending i/o to the DDE IN section. This information is passed to the Process Data section, which deciphers the type of i/o. The I/O Handler receives two types of communications. The first type of i/o is a requested state. This information is passed to the Update Tag section, which searches and updates the correct tag in the tag list. The second type is specific instruction from the Knowledge Base. For instance, a refresh instruction will force the I/O Handler to request a state or states. This request will cause a requested state to arrive, which will cause an update. Generally, the instructions will make the Input section add, update, and remove tags in the list.

Section 2.3.3: The Event Handler

The Event Handler is designed to repeat regular, timed events in the simulation. The idea was to offload the Knowledge Base by automating the execution of routine or iterative tasks. These tasks would need to happen via a *pulse* type message. The Event Handler is modelled in figure 2.6.

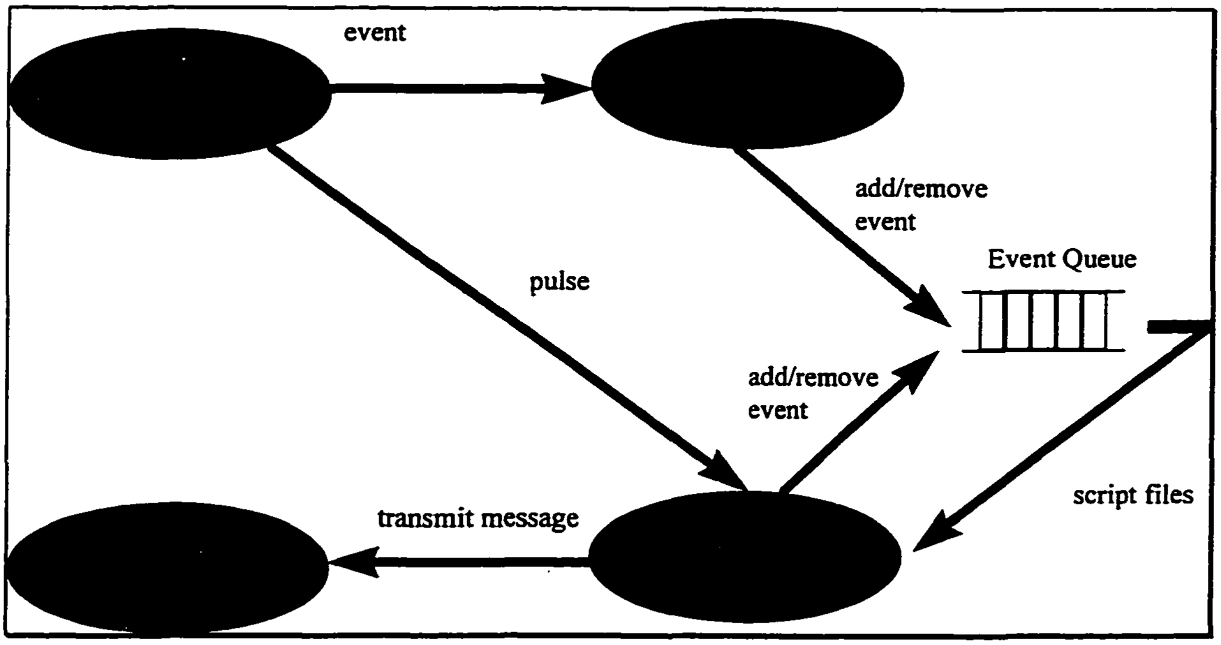


Figure 2.6 : The Event Handler

The Event Handler receives two types of instructions. The first is an event to be either scheduled for execution or to be removed. Based on this, the event is either added to the event queue or removed. The other type of instruction, pulse, tells the event handler to process the event queue. The execute section reads the next event to be processed, transmits a message to the Knowledge Base if required, and either re-submits the event to the queue or removes it.

Section 2.3.4: The Knowledge Base Script Language.

A *knowledge base* is a set of rules (or knowledge) stored in a format capable of being interpreted by the computer(Oren,1991). In the simulation engine, the Knowledge Base entity is the task associated with interpreting the rules that represent the process being simulated. It communicates with the other tasks in the simulation using the DDE IN and DDE OUT entities. The Knowledge Base model is shown in figure 2.7.

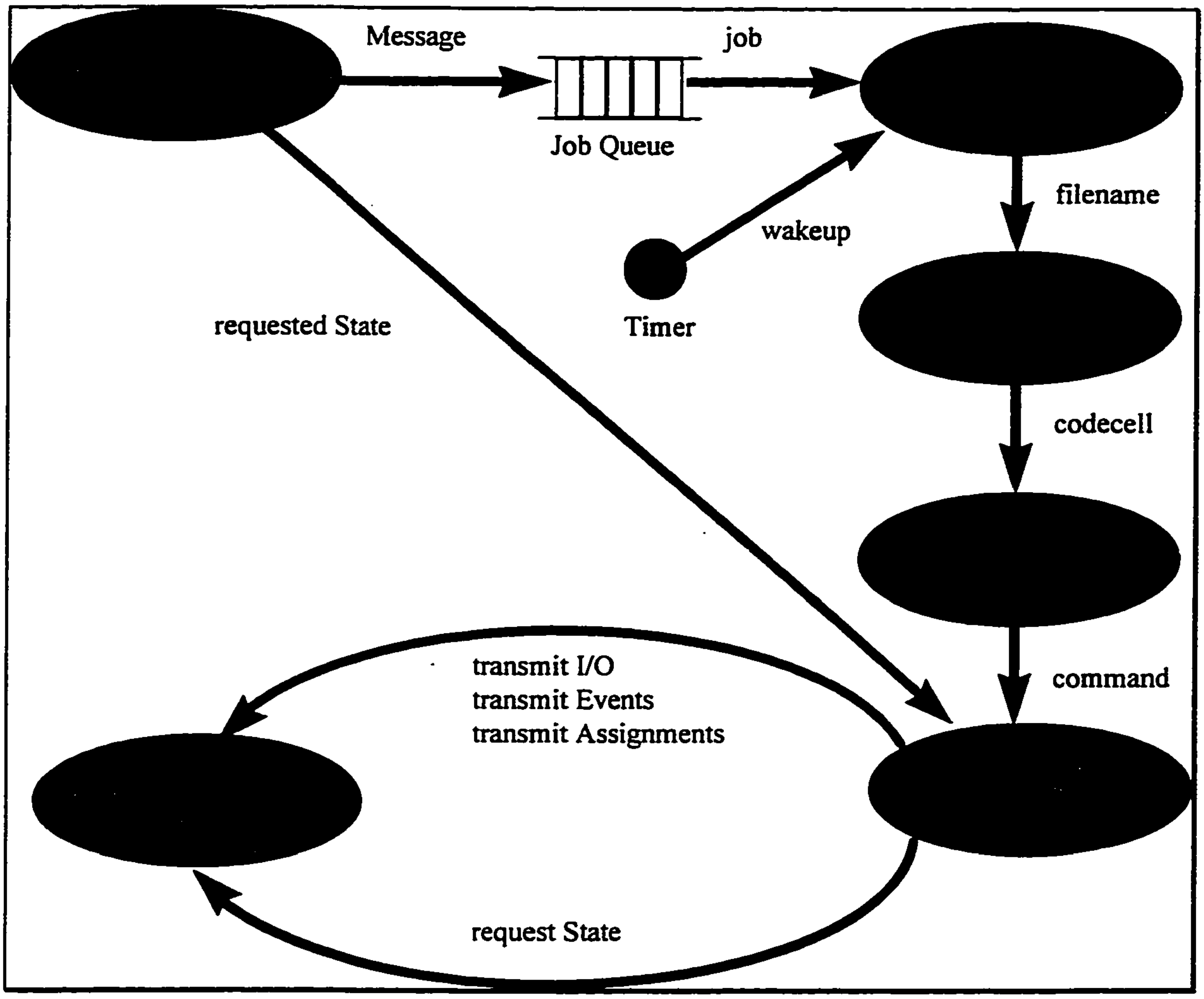


Figure 2.7 : The Knowledge Base

The heart of the Knowledge Base is a job queue which stores jobs until they can be executed. As messages arrive via the DDE IN section they are converted to jobs and stored in this queue. A timer regularly sends a "wakeup" signal which causes the "update simulation" section to read jobs from the queue and pass them on as script filenames to the loader/parser section. The loader/parser section reads the file, creates a parsed object called a codecell, and invokes the interpreter. The interpreter executes each command. By running each command the rules of the simulation being modelled are executed. Some of the high level commands can request and transmit information via the DDE OUT section which communicates with the rest of the simulation engine. Other commands can call more scripts, allowing models to be built from the top down - less knowledge to more knowledge. The script language is covered in more detail in section 4.1.

The Knowledge Base is meant to be used as an discrete event (message) driven model. The messages that arrive from other simulation tasks represent events (such as user i/o, or value change) that affect the simulation. The Knowledge Base runs the appropriate script - it applies the correct rule for the situation.

It is possible to do continuous modeling by utilizing the timer. By using a timer the Knowledge Base can provide a quantum unit of time (based on a constant number of wake-ups), which can be used by the Simulation Engine for timed events. This quantum, or simulation heartbeat, can be used to create a continuos simulation by sending out regular pulse messages to the other simulation tasks. In order to facilitate this, a special script is always run when the heartbeat is active. It is automatically queued to run at each heartbeat. All timed events can be triggered from this script.

The high level language allows the simulation designer to develop a simulation with complex rules that can represent quantitative and qualitative events within the model. This allows us to build rules which represent the "soft" concepts (Rothenberg,1991). These can be hard to define (e.g. getting warmer, usually around 10%) and require the flexibility that a knowledge base can offer. The Knowledge Base program provides the tools required to implement a set of rules. Combined with the other simulation engine tasks, it is a key component to the design of the entire simulation system.

Chapter 3 : THE WINDOWS PROGRAMMING ENVIRONMENT

This chapter examines the influencing factors of the chosen programming environment, Windows 3.1. It discusses how the actual operating system influenced the Simulation Engine's design. Programming libraries used in the development of the simulation engine, OWL (Object Windows Library) and CLASSLIB (class library), are described. Finally, the DDE (dynamic data exchange) protocol used in the simulation engine is introduced. This chapter provides the groundwork for understanding chapter 4.

Section 3.1: The Windows Operating System Environment.

The simulation engine was originally designed to work in a Windows 3.1 environment using the Win16 API. The completed project runs in Windows 95, Windows for Workgroups or Windows 3.1, but still depends on the basic principals found in Windows 3.1.

The Windows operating system is a non-preemptive multi-tasking operating system. This means that the operating system can not interrupt a process to force a fair time slice. Instead, program design in Windows 3.1 is based around voluntary release of system resources. The downside of this approach is that if a process becomes locked up it will lock all of the operating system. (It should be noted that Windows 3.1 will terminate such processes if a CTRL-ALT-DELETE keyboard message is issued. Previous versions of Windows could not even do this. The Windows NT platform uses true preemptive behavior, finally bringing safe multitasking to the PC Windows).

To understand the Windows environment we must first understand the relationship between the application and its window. The application interfaces between the window and the operating system. The window links the user to the application.

The window has associated with it a procedure which defines how the window will react to the communications from the application. As the application receives and deciphers information tokens called *messages* from the operating system, they are dispatched to the window procedure. This procedure then reacts according to the program design.

Messages can be originated from hardware (mouse, keyboard) or other applications. The messages are processed by the operating system, and passed on to the program in one of two ways. First, the message may be *posted*. A posted message is left in an application message queue to be read when the program gets a chance. The second method is a *send*. A message which arrives using a send goes directly to the window process, bypassing the application. This is only used for high-priority communications. The message passing process is modelled in figure 3.1. (Yao,1994).

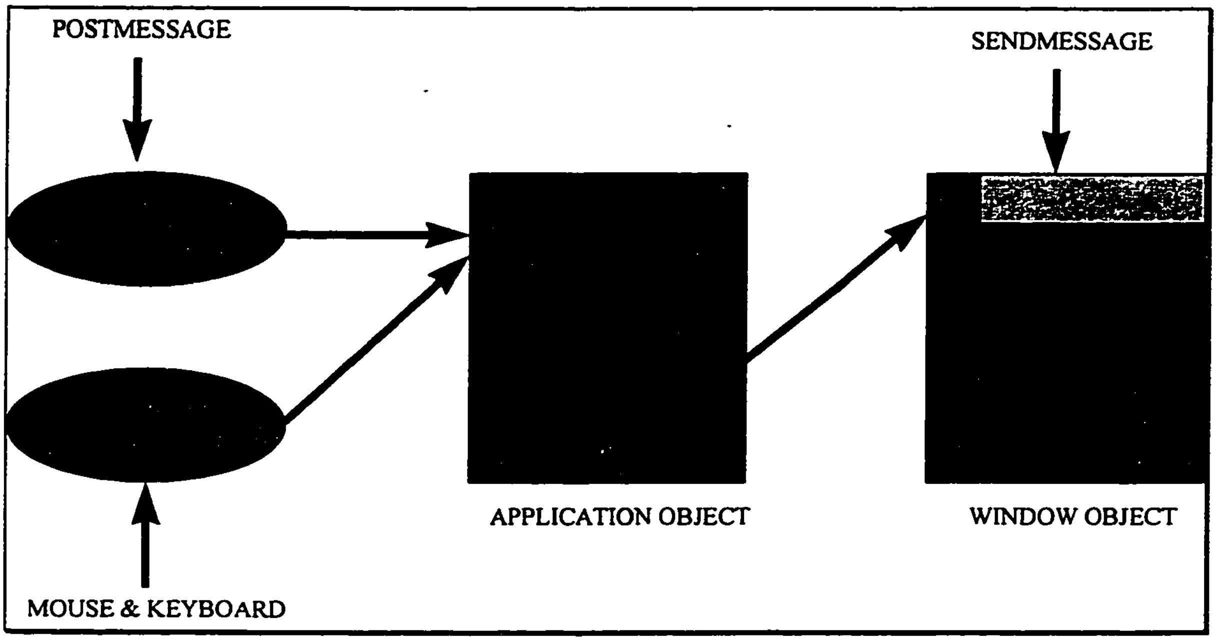


Figure 3.1 : Windows Message Passing System

The application's GetMessage loop releases control to the operating system after each message is processed. This loop is in every Windows program, and typically looks like:

```
while (GetMessage(&msg, 0, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

The GetMessage function will only return false when a quit message is received. At this point the loop will end and the program will terminate.

The key to the process is the message. A message is a structure made up of six parts. From the Borland C++ windows.h file we can see the structure:

```
typedef struct tagMSG
{
   HWND       hwnd;
   UINT       message;
   WPARAM     wParam;
   LPARAM     lParam;
   DWORD      time;
   POINT      pt;
} MSG;
```

The *hwnd* is a window handle. As explained earlier, a window handle identifies a window uniquely. In this case it is the window targeted to receive this message. The *message* is an unsigned integer (either 16 or 32 bit depending on which applications programmer interface you are using) which identifies the type message (the details of the different types of messages is out of scope for this paper). The *wParam* and *lParam* variables are data - the type of data depends on the type of message. The variables *time* and *pt* identify the time the message was created and the position of the mouse at that time.

Because of a message driven, non-preemptive nature of the Windows operating system the style of programming is a bit different. A windows program amounts to a set of "traps" which capture different messages. Even when using a timer messages are generated and responded to. The environment is very easy to develop in, but the influence on the simulation engine is substantial. The simulation engine is a reactive type program; it waits for user input, it waits for timed events. This lends itself very nicely to event driven typed applications, which in turn assists the simulation engine in creating event driven simulations.

Section 3.2: The Object Windows Library

This section covers in general terms the use of the Object Windows Library. The OWL (Object Windows Library) simplifies the process of creating windows applications. The process of creating a windows application requires the programmer to create a application loop which reads the different Windows messages and a window function which interacts with the user. As well, the application needs to register its name, icon, cursor etc. For example, look at this code taken from (Ammeraal,1993) Windows Wisdom for C and C++ Programmers (page 5):

```
#include <windows.h>

long FAR PASCAL _export WndProc(HWND hWnd, UINT message,WPARAM
        wParam,LPARAM lParam);

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpCmdLine,
        int nCmdShow)
{
    char szAppName[]="wtest";
    WNDCLASS wndclass;
    HWND hWnd;
    MSG msg;
    int xScreen = GetSystemMetrics(SM_CXSCREEN),
       yScreen = GetSystemMetrics(SM_CYSCREEN);

    if (!hPrevInstance)
    {
        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc = WndProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = hInstance;
        wndclass.hIcon = LoadIcon(NULL,IDI_APPLICATION);
        wndclass.hCursor = LoadCursor(NULL,IDC_ARROW);
        wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName = NULL;
        wndclass.lpszClassName = szAppName;
        if (!RegisterClass(&wndclass)) return FALSE;
    }
    hWnd = CreateWindow( szAppName, "Windows Test",
            WS_OVERLAPPEDWINDOW, 0, 0, xScreen, yScreen,
            NULL, NULL, hInstance, NULL);
        ShowWindow(hWnd,nCmdShow);
        UpdateWindow(hWnd);

        while (GetMessage(&msg,NULL,0,0)) {
         TranslateMessage(&msg);
         DispatchMessage(&msg); /* message is sent to the Window */
        }
        return msg.wParam;
}

/*
*     Window Procedure
*/
long FAR PASCAL _export WndProc(HWND hWnd, UINT message,WPARAM
wParam,LPARAM lParam)
{
        switch (message)
        {
              case WM_DESTROY :
                    PostQuitMessage(0); break;
              default:
                    return DefWindowProc(hWnd, message, wParam, lParam); break;
        }
        return 0L;
}
```

This program creates a window with the title "Windows Test" and waits for the a *WM_DESTROY* message. The programmer is required to a significant amount of work for very little return.

The OWL library (version 2.0) hides this complexity by providing a detailed class hierarchy of 128 classes (Yao,1994). The philosophy of OWL is to create two main objects; first, an application object descendant of *TApplication* and secondly a window object descending from *TWindow*.

The *TApplication* class takes care of the non-window functionality. These include registering the window class, icon, cursor etc. Note that the programmer can override any or all of these if necessary.

The *TWindow* class provides for user input and output. This class comes with over 55 descendants, including *TDialog, TControl*, and *TFrameWindow*. These are further extended to allow buttons, checkboxes and other custom design user interface items. The application object actually expects a

TFrameWindow, which includes basic window items (e.g. scrollbars), but can be made to work with any descendent of *TWindow*.

The OWL library also simplifies the program entry point. Instead of using the complex *WinMain* it uses a more familiar looking *OwlMain*, which is similar to the traditional C++ *main*. This function creates an instance of the application class and runs it. The application object creates an instance of the window class and monitors the windows messaging system. Messages are read in and dispatched to the window object, which can have member functions that respond to the messages. A simple OWL based program, one which produces similar results as the previous example, is given:

```
#include <owl\owlpch.h>


// —— window object
class MyWin : public TFrameWindow
{
        public:
                MyWin(char *title) : TFrameWindow(0,title) {
                        Attr.W = GetSystemMetrics(SM_CXSCREEN);
                        Attr.H = GetSystemMetrics(SM_CYSCREEN);
                }
                ~MyWin(){}

};

// —— application object
class MyApp : public TApplication
{
        public:
                MyApp() : TApplication() {}
                void InitMainWindow() {
                        SetMainWindow( new MyWin("Windows Test"));

                }

};

// —— main entry point
int OwlMain(int, char **)
{
        return MyApp().Run();
}
```

This program is not only smaller than the non-library version, it is also much more flexible. The programmer can focus on design of the application and concentrate more on the program task ahead of him. The cost of using the OWL library can be measured in size of the executable. The non-library program is 32 Kbytes where the OWL program is 175 Kbytes. While significant, this size is more than justified in the increase of programmer productivity. The simulation engine project spanned both version 1.0 and 2.0 of the OWL library, requiring one major re-write. The functionality and simplicity provided by the OWL library allows the programmer to focus on the design of the program, and forces the message handling details of the Windows operating system into a background role.

Section 3.3: CLASSLIB - Container Classes.

This section discusses the Borland C++ CLASSLIB, which provides container classes.

The types of containers are (Borland,1993b): *Array, Association, Bag, Binary Tree, Dequeue, Dictionary, Double-listed list, Hash table, List, Queue, Set, Stack* and *Vector.* These classes provide a extensive set of tools that can be used in moving from conceptual model to actual computer program.

In order to implement these containers on specific classes, Borland C++ makes use of the *template* concept provided by C++. A template looks like a class with member class substituted with a pseudo-argument. For example consider the following example (Stroustrup,1991):

```
template<class T> class vector {
          T * v;
          int sz;
public:
          vector(int);
          T& operator[](int);
          T& elem(int i) { return v[i]; }
          // ...
};
```

This template provides a class capable of storing an array (or vector) of elements of some class. If the programmer required an array of integers, he would use:

```
vector<int> int_array(20);
```

The resulting vector object would store, or contain, integers. Just as easily, we could create a more complex class and create a vector from them.

The Borland CLASSLIB library uses this functionality to build many containers implemented in many different ways. Depending on the fundamental data structure (FDS) the programmer chooses for implementing the abstract data type (ADT) member functionality will vary. Not every combination of FDS/ADT is available (although the programmer can expand upon these). Those that are provided are shown in figure 3.2 (Borland,1993c page 223).

			ADT			Sorted		
FDS	Stack	Queue	Dequeue	Bag	Set	Array	Array	Dictionary
Vector	x	x	x	x	x	x	x	
List	x							
DoubleList			x	x				
Hashtable								x
Binary tree								

Figure 3.2 : CLASSLIB Container Classes

In order to make use of an abstract data type, Borland uses a specific naming convention. The declaration works by combining ADT "as" FDT. For instance if you wanted an array implemented as a vector to contain floating points, the declaration would look like:

```
TArrayAsVector<float> x(100);
```

The result would be an array of floating points. The advantage of this is that the array could be of a more complex class. (Sections 4.4 and 4.3 give such examples used in the Simulation Engine).

The prefix to the ADT indicates other attributes the template has. The *T* indicates that the template is a member of a Borland library (Borland uses this notation extensively in all their libraries). Other prefixes, all which may be present or absent at the same time, are listed in figure 3.3.

M	User supplies the memory-management container
I	Indirect container
C	Counted container
S	Sorted container

Figure 3.3 : CLASSLIB ADT Prefixes

These indicate whether the user's class will have member functions for copying itself (M), whether container is actually storing pointers to objects (I), whether the elements in the container should be counted (C) , and whether the elements should be sorted(S). Depending on these selections, the programmer will be required to build different functionality into their class.

With some learning, these container classes can reduce the workload of the programmer. The alternative is to build each container class individually. This time consuming process is worthwhile only in situations where special behaviours are required. When a simple queue or array is required, the CLASSLIB can help considerably.

Section 3.4: Dynamic Data Exchange for Inter-Process Communication.

The Simulation Engine uses the DDE (Dynamic Data Exchange) protocol for communicating between the different processes that are a part of the Simulation Engine. This section discusses about the protocol itself and how it is implemented using Windows DDEML link library. The DDE protocol is native to the Windows environment and is currently losing favor to the more popular OLE (object linking and embedding) protocol. The DDE protocol was chosen because it is easily implemented and provided the basic client/server communication required to complete the Simulation Engine. If another protocol had been used, it likely would not be OLE - rather it might make sense to use one of the IP based protocols, such as UDP. This would allow for easy communication between platforms; extending the usefulness beyond the Windows/PC environment to include UNIX, VMS, OS400, etc.

The DDE protocol is implemented on the Windows messaging system (covered in more detail in section 3.1). The protocol provides a means of establishing a connection, asking for action, sending/receiving information and closing connection. Each individual conversation can be viewed as a client/server type relationship, since the originator (or client) starts and controls the conversation with the server. Multiple conversations can be started between processes; thus a peer-to-peer relationship can be accomplished by having each process initiate a conversation (Clark, 1992).

A conversation is initiated by using the Windows *SendMessage()* function to send a WM_DDE_INITIATE message to the target application window. This message can be lost (application is either not there or not DDE capable), rejected (the application is not interested in starting a conversation) or accepted. If the message is accepted both processes create DDE handles for the conversation. This handle provides the processes access to global memory objects allowing data to flow between them.

The client process can then request a specific task be executed by using a WM_DDE_EXECUTE message. The server process will receive the message, get the required data (typically a text string data) and perform the task. A response WM_DDE_ACK is returned, either a "YES" message indicating a successful execution of the command or a "NO" message indicating failure.

The client process can request data using the WM_DDE_REQUEST message. The server extracts the data from the global memory and either sends a WM_DDE_DATA message (your data is now available in the global memory) or a WM_DDE_ACK. The second message will return a "YES" ; I have this information but cannot give it to you right now, or a "NO"; I do not know what is being asked for.

The client can transmit data to the server by loading the global data and sending a WM_DDE_POKE message. The server process receives the message and returns a WM_DDE_ACK message. The response will be a "yes"; I got and understood your request, or a "no"; what are you talking about.

An additional way for the client to get data is to send a WM_DDE_ADVISE message. This tells the server to send WM_DDE_DATA messages whenever the server feels it is a good time (normally when the data changes). A WM_DDE_ACK will be returned with a "no" if the advise loop is rejected. Otherwise the server will continue to send messages until it receives a WM_DDE_UNADVISE message.

Finally, to conclude a conversation a WM_DDE_TERMINATE message is sent. This instructs the server (or client) to clean up the shared memory and shutdown the DDE link. This message must be sent before an application exists or memory will be lost.

The dilemma with the DDE protocol is the amount of programming and details involved in using it. A simpler approach, which takes care of communicating with the Window's messaging system, is required. In order to accomplish this, Windows 3.1 comes with a dynamic link library called the Dynamic Data Exchange Management Library (DDEML). This library provides functions and design conventions for the DDE protocol. It insulates the programmer from working with global memory and takes responsibility of garbage collection.

The key to the DDEML programming conventions is the use of a *CALLBACK* function. This function is registered to the application using a special function *DdeInitialize()*. Once this function is loaded it will be called whenever a DDE type message arrives for the application. The function will be passed a message identifier which tells the CALLBACK function what type of message has arrived. These identifiers correspond to the underlying DDE messages:

XTYP_EXECUTE	WM_DDE_EXECUTE
XTYP_REQUEST	WM_DDE_REQUEST
XTYP_POKE	WM_DDE_POKE
XTYP_AVSTART	WM_DDE_ADVISE
XTYP_ADVSTOP	WM_DDE_UNADVISE

The difference is that the identifiers come with the handles to the data. These can then easily be converted to their text strings using other functions provided by the DDEML.

The DDEML encourages the use of special conventions. These include the concepts of *services*, *topics* and *items*. A service name is a string that the DDEML maps to an application. Identifying the DDE server process is more easily done by using the service name, rather than identifying the window to post the message to. Within each service several topics can be identified, so that a conversation is picked by selecting a service and a topic. The actual entity being discussed is referred to as the *item*.

The DDE protocol, as implemented using the applications programmer's interface DDEML, provides a mechanism for inter-process communication. Further it is a widely used protocol, available in most word-processing and spreadsheet applications. The DDE protocol is a valuable part of the Simulation Engine.

Chapter 4: IMPLEMENTATION OF THE SIMULATION ENGINE

This chapter is intended to outline and describe in some detail how the major parts of the Simulation Engine were implemented. Each task in the simulation engine has an implementation issue: the Knowledge Base has a Lisp like language, the Simulation State uses a Variable Handler, the I/O Handler's Tag class hierarchy, and the Event Handler's event class. As well, the DDE client/server class used by all the tasks in the simulation engine is discussed.

Section 4.1: The Knowledge Base Script Language.

In order to create a Knowledge Base, a method was required of describing the pieces of knowledge. In order to accomplish this, a script language was developed. To simplify the programming task, the script language is based on Pure Lisp (Pratt,1984) which has been extended to allow program to program communication using the Dynamic Data Exchange protocol (DDE). The use of a simple programming language allows the simulation designer a great deal of flexibility. The script language can communicate changes of value, graphic displays, etc. to the other programs in the simulation engine. Formulae, functions and even random events can be generated. By allowing the language to call other script files, the programmer can build and expand upon the simulation. The script language allows for a precise description of the rules that make up the simulation.

The simulation script language has very similar syntax to Lisp-like languages. Many of the commands of Lisp have been implemented, as well as some extensions to allow inter-process communication and high-level floating point arithmetic. It is not the purpose of this section to discuss the Lisp programming language; a detailed breakdown of the implemented commands can be located in Appendix II.

The script language is implemented using two classes: *Loader* and *CodeCell.* A Loader object interfaces to a text file containing script code. This class has all the member functions required to read in, parse, and return a pointer to a *CodeCell* object. A skeleton of the loader class is given below:

```
class Loader
{
        private:
                ifstream *fp;
                char filename[30];
                // ... other data items used in the parsing process...
        public:

        Loader(char * _filename);
        Loader();
        ~Loader();

        //  the parser
        //
        CodeCell * Parse();
        // ... and other member functions used in the parsing process
};
```

The resulting *CodeCell* object contains both the parsed byte code instructions as well as the member functions for interpreting them. A skeleton of the *CodeCell* class is given below:

```
class CodeCell
{
        private:
                char *data;
        public:
                CodeCell *CAR, *CDR;
                int type;

                // constructors
                //
                CodeCell(int _type =0);
                CodeCell(char *_data, int _type=0);
                CodeCell(CodeCell *r);


                ~CodeCell();

                void AssignData(char *_data);
                void AssignType(int _type);
                void Dump(string * core);

                CodeCell * Interp(CodeCell * env);

                CodeCell * Eval(CodeCell *env);
        CodeCell * Apply(CodeCell *f,CodeCell * args, CodeCell *env);
        // ... predicates and primitive functions of the script language ...
};
```

Each *CodeCell* can either be a LIST/CELL or an ATOM. Most Lisp implementations include two other types, FUNCTION and NUMBER (Pratt 1984). For the purposes of the script language, the usage within the script file is sufficient to determine whether an ATOM is a number, function or variable. For example, if an atom is the first entry in a list it is assumed to be a function. If an atom begins with a numeric character it is assumed to be a number.

In a similar fashion, we can consider variables and functions to be similar types. A variable can be equivalent to a LIST or an ATOM, a function definition is a LIST starting with a LAMBA function. This approach requires that a variable name cannot be used as a function name or vice-versa. The advantage is that both *functions* and variables are made persistent by implementing a *CodeCell* object called *env*. This object belongs to the Knowledge Base's window object. It is passed to the *Interp()* member function call, so that functions and variables will be available to other scripts.

The *Interp()* member function can travel through a *CodeCell* tree and execute its content. This structure is defined by the data, type and its CAR and CDR *CodeCell* pointers. For example, figure 4.1 shows the structure of the Lisp instruction ***"(SETQ X 10.2)".***

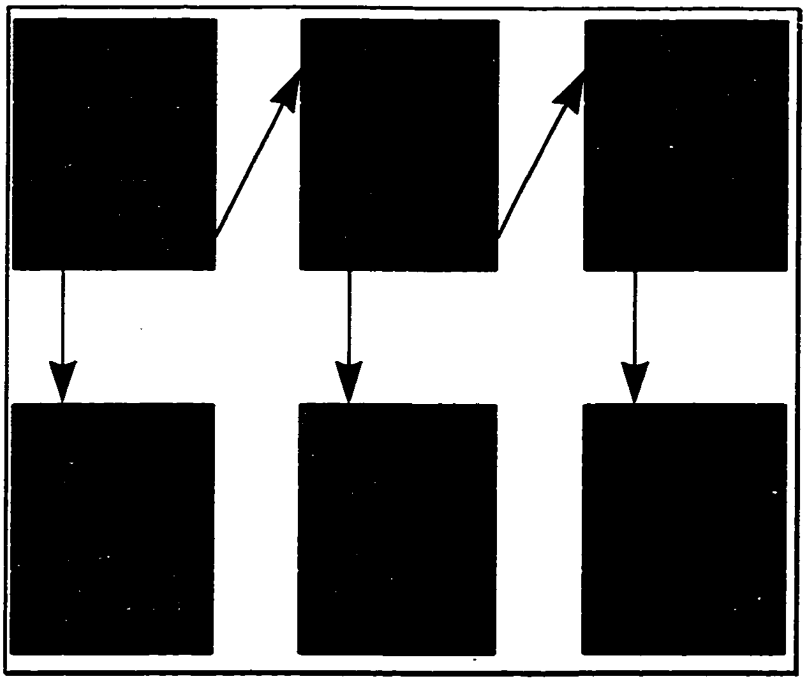


Figure 4.1 : An Example CodeCell Structure

Once the Loader object has created the tree of *CodeCell* objects it can then be executed. The returned pointer to *CodeCell* object calls its *Interp()* member function. This member function takes the argument *env.* The *Interp()* member function returns a pointer to *CodeCell* object, representing the result of the execution. This is typically a *t,* meaning success, or a *nil* , meaning failure. This information is extracted using the *Dump()* member function into a string variable which can be displayed or written to a log file.

Since the script program is interpreted line by line, a while loop is used to process the entire file. Consider this code segment which is based on the Knowledge Base Window object's *Message* member function:

```
Loader * x;              // interface to file "filename"
CodeCell * runme;        // pointer to CodeCell to be run
CodeCell * result;       // pointer to the result of running "runme"
string core;             // a string for storage

x = new Loader(filename);
// while there are lines to parse
while ((runme = x->Parse()) != NULL) {
        if (!runme->null()) {
```

```
                    result = runme->Interp(env);
                    core = "" ;
                    result->Dump(&core);
                    if (core != "") logfile->write((char *)core.c_str(),core.length());
               delete result, runme;
               }
          }
          delete x;
```

When the Knowledge Base program is initially run, a prebuilt script file is parsed and executed. This file contains definitions for built in functions as well as many simple functions written in the script language. These are stored in the *env* object for access by other scripts. As each subsequent script is run or re-run it can add to or replace entries in the env object. While these values are available to the Knowledge Base, they are not available to the Simulation Engine or other DDE compatible programs. Values that are required by the I/O Handler or Event Handler should be transmitted to the Simulation State. These values can be read in when required by the script file, and transmitted back when required. These commands are DDESETQ and DDESENDQ and are outlined in Appendix II.

A communication issue arises when the script file requires a state value from the Simulation State. In this case the program makes the requests, then goes into a waiting state. This is accomplished by monitoring a global variable where the response will be deposited. The polling is done at high speed, but permits other programs to continue execution. The program segment is provided below:

```
do {
     This->anyRequest(This->targetConv[conv],v->data);      /
     //   This code is used to gracefully wait until data has been
     //   returned. It will LOCK UP if the state handler is not
     //   around to accept the request for data.
     //
     while (global_flag==0) {
          while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }
     }
     global_flag =0;
}
```

The script language is a complete language capable of inter-process communication, mathematical manipulation and problem description. The script language could be replaced by another high-level language by maintaining the message passing rules (section 4.5). Using the Pure Lisp language as a guideline allowed for a simple to use and implement script language.

Section 4.2: The Input/Output Tag Object

As mentioned in Section 2.3.2, the Tag object is assigned the task of interacting with the user. In order to develop a versatile system, the Tag object is only the parent object of an hierarchy of descendants. Each descendant is specialized for a particular type of interface. At the most general level, a Tag object has a x, y, dx, dy display region and an identifier or *tagname* which is unique to that tag. The Tag object class is given below:

```
class Tag
{
        TRegion *pos;
        char  *tagname;

public:
        int TAGTYPE;

        Tag(TRect _tr,char *_tagname);
        ~Tag(){ delete pos; delete tagname;}

        virtual void Do(TWindow *parent) {}
        virtual void Paint(TDC &dc) {}

        TRect GetRgnBox() { return pos->GetRgnBox(); }
        char *GetTag() { return tagname; }

        int Contains(TPoint x) { return pos->Contains(x); }
        int Match(char *ts) { return (!strcmp(tagname,ts)); }
};
```

At the topmost level, the tag provides member functions for checking tagname match, checking if a point is contained within the region, returning the tagname and display rectangle and two virtual function, Do and Paint. The Paint member function is provided to be overridden with a function capable of displaying the

Tag. Note that parent object Tag has no displayable form. The virtual function Do is provided to act upon being selected by the user. In this way the tag object provides both input and output functionality to its descendants. The use of virtual functions allows the descendants to be stored in a list of type Tag. Each descendent may use the Do and Paint functions as best fits its responsibility. Thus the Window object for the I/O Handler contains a simple member function that reacts to all mouse-button-up window events:

```
void MyWin::EvLButtonUp(UINT id, TPoint& p)
{
        char s[100];

        for (int i=0;i<200;i++) {
                if (taglist[i] && taglist[i]->Contains(p)) {
                        taglist[i]->Do(this);

                }
        }

}
```

Six types of tags have been developed by inheriting from class Tag. The hierarchy tree of available tag types is given in figure 4.2.

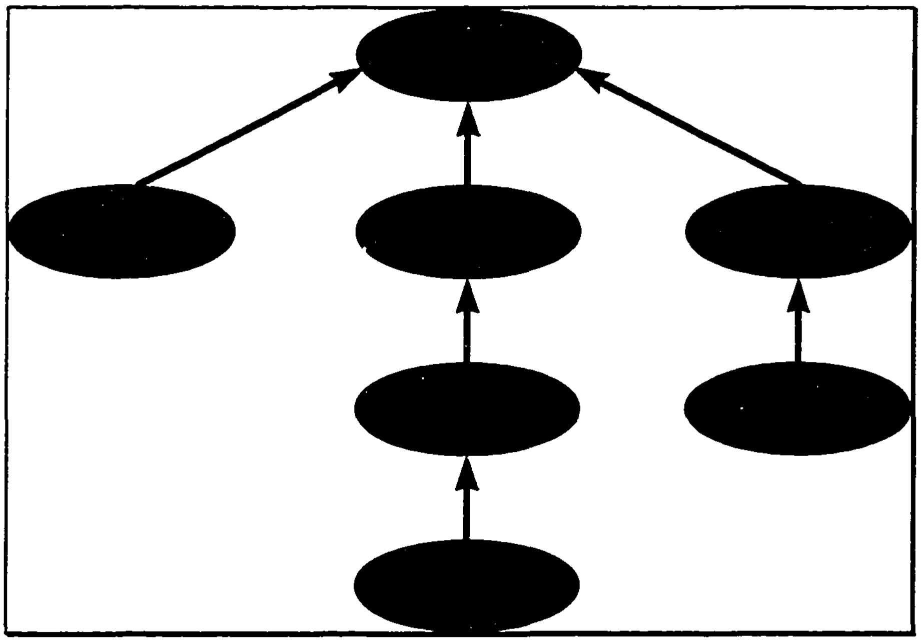


Figure 4.2 : The Tag Inheritance Tree

Each of these classes adds to the parent it inherited from. Consider the inheritance of PopupText. Its branch in the hierarchy tree begins with SimpleText, which adds the ability to display text on the screen. This text cannot be changed other than by deleting the tag. The next step in the branch is SmartText, which creates a selectable text on the screen and makes the text dynamic. Not only can the text be changed, selecting this tags region will cause a message to be sent to the Knowledge Base, allowing the user the ability to directly cause the simulation rules to be invoked. From this PopupText adds a popup dialog that allows the user to input a value for a state. Not only can this value be changed and send messages to the Knowledge Base but also it can be used to input data into the simulation.

The other tags follow a similar pattern. All the possible tag configurations and how to use them are described in Appendix III.

Section 4.3: The Variable Handler Class

The Simulation State provides variables, or states, to the other tasks in the simulation engine, as well as any DDE compatible software (this is covered by section 5.3). The simulation states consist of floating point values and related variable names. Access, storage and manipulation of these states are accomplished by using a class called the *VarHandler*. This class was originally designed as a sorted structure array containing a string and a float. In its original format, one *dirtyBIT* variable was used to determine whether the state had been written to. This boolean flag would be set to TRUE if the value had changed since last being accessed and FALSE otherwise. As the complexity of the simulation engine increased, and as tasks other than the I/O Handler accessed the Simulation State, the implementation of the dirtyBIT become too complex and its usefulness declined in the wake of the power of the Knowledge Base. Future versions of the Simulation State (refer to section 6.3) may require the dirtyBIT to reduce unnecessary interprocess communication.

The next generation of the VarHandler class was simplified by utilizing a *Dictionary* class member object called *dict* to store and sort variable names and their corresponding floating point values. The class is given below:

```
class VarHandler {
private:
          Dictionary *dict;
public:
          VarHandler();
          ~VarHandler();

          int add(char *s,float x);
          int assign(char *s,float x);
          float get(char *s);
          float remove(char *s);

};
```

In order to implement the Dictionary class, a CLASSLIB (as discussed in section 3.3) template, *TDictionaryasHashTable* was used. This template takes a storage type class and builds a dictionary type from it. It requires that the storage class has a member function *HashValue()* which returns a hash based sort key. This was accomplished by using another class provided by Borland's CLASSLIB, *string,* which has a member function *hash()*. The resulting class, *HashString,* is then associated with type float, the class the dictionary will lookup. The code used to produce the Dictionary type is provided below:

```
class HashString : public string {
  public:
          HashString() : string() {}
          HashString(const char* s) : string(s) {}
          unsigned HashValue() const { return hash(); }
};
typedef TDDAssociation<HashString,float> ClassData;
typedef TDictionaryAsHashTable<ClassData> Dictionary;
```

The resulting class can *add, assign, remove* and *get* values in the Simulation State. When a state is called for which is undefined (i.e. not in the dictionary) the VarHandler returns a special value, FLOATNULL, which other programs accessing the Simulation State must recognize as a NULL value. This value is defined as -1e37, an unlikely number in any simulation.

The VarHandler is a good example of a class greatly simplified by using existing libraries. Although it and the State Handler are the simplest entities in the Simulation Engine, they are very important. The values stored in the Simulation State represent the entire simulation. The whole process of storage and retrieval is taken care of - as well as providing an access point to third party programs which may have a use for these values.

Section 4.4: The DDE Client/Server Class

Each of the tasks in the Simulation Engine use the DDE protocol to communicate (also see section 3.4) and each requires a window interface to the user. To simplify this process, the TFrameWindow class provided by Borland C++ is used to build a descendent class, called MyClientServer. The resulting class includes all the necessary window functions and most of the DDE protocol functionality. The remaining functionality must be a part of the application part of the program - not the window. This is handled using a common function, which is included in each of the tasks.

To begin with, we examine a portion of the MyClientServer header file:

```
class MyClientServer : public TFrameWindow {

...
public:

...
        MyClientServer::MyClientServer(char *title,
                                       char *servicename,
                                       char *topicname,
                                       char *targetservers[],
                                       char *targettopics[],
                                       int _numberoftargets);

...
        void anyRequest(HCONV hConv,char *item);
        void anyTransmit(HCONV  hConv,char *item,char *s);
        void anyAdvise(HCONV  hConv,char *item);
...
        virtual void ProcessData(int flag,int conv,char *topic,char *item,char *data);
```

```
            virtual void LoadDataReq(char *topic,char *item);
    ...
    };
```

This class inherits from TFrameWindow which is a descendent of TWindow, which allows MyClientServer to be a window class (refer to section 3.2). This class can then be inherited from to create a window class for each particular program. When the object is first constructed it should provide a title for the screen, the name of the DDE service to be provided, the name of the topic for this service, an array listing the servers to connect to, another array containing the topics to connect to in the corresponding servers, and finally a number_of_targets variable.

Using this information, the Client/Server object will create DDE handles for providing server/topics and for connecting to the required DDE servers. These connections will be made at run time. These handles are of HCONV types; these are used in the member functions to allow communication. These functions include: *anyRequest()*, which sends a request to the targeted server; *anyTransmit()*, sends a Poke type message; *anyAdvise()*, which sends a request to be added to a advise loop.

Two virtual member functions are provided to allow descendants to override the pre-built functionality. *ProcessData()* receives the incoming data once it has been decoded from the DDE handles. This allows the descendent to be concerned with the text message and not be bothered with the details of the protocol. By overriding this function the descendants provide their own responses to incoming communications.

The second virtual member function is *LoadDataReq()* which is called with a topic and item and is expected to load class variable *DataReq*. This variable will be used by *ClientServer* functions to communicate with a client task. As with *ProcessData()*, this function works strictly with text strings - once overridden by the descendant class this function will simply use topic and item to determine what value is required in *DataReq* and copy it there. The *ClientServer* will then take responsibility to create a DDE handle and respond to the requesting client.

Part of the DDE functionality must be provided by the application part of the program (recall that in windows every program is part application and part window - refer to section 3.1 and 3.2). This is done by using a CALLBACK function which is registered by the application class. This means any incoming DDE communications will first be captured by the function, which will then call the appropriate Client/Server member functions. The CALLBACK function looks like:

```
HDDEDATA FAR PASCAL _export
MyApp::CallBack(WORD type,WORD wFmt,HCONV hConv,HSZ hsz1,HSZ hsz2,
                        HDDEDATA hData, DWORD, DWORD)
{
        int i;
                char t1[100],t2[100];

        switch (type) {

        case XTYP_ADVREQ:
                if (This->MatchTopicAndItem(hsz1, hsz2,t1,t2))
                  return This->DataRequested(wFmt);
                return 0;

        case XTYP_XACT_COMPLETE:
                        This->anyReceivedData(hConv,hData,hsz1,hsz2);
                return 0;

        case XTYP_ADVSTART:
                if (This->MatchTopicAndItem(hsz1, hsz2,t1,t2)) {
                        for (i=0;i<=MAX_LOOP;i++) {
                                if (This->Loopstr[i]==0) {
                                        This->Looptop[i] = hsz1;       // store topic
                                        This->Loopstr[i] = hsz2;      // store item
                                        This->Loopcon[i] = hConv;
                                        This->Loop++;
                                        return (HDDEDATA)1;
                                }
                        }
                }

                return 0;

        case XTYP_ADVSTOP:
                if (This->Loop && This->MatchTopicAndItem(hsz1, hsz2,t1,t2)) {
                for (i=0;i<MAX_LOOP;i++) {
                        if (This->Loopcon[i] == hConv) // conv match
                        if (DdeCmpStringHandles(hsz1, This->Looptop[i]) == 0)
                        if (DdeCmpStringHandles(hsz2, This->Loopstr[i]) == 0) {
                                This->Loop–;
                                This->Loopcon[i] = 0;
                                This->Loopstr[i] = 0;
                                This->Looptop[i] = 0;
                                return (HDDEDATA)1;
                        }
                }
                }
                ::MessageBox(GetFocus(),"Error! ","DDE SERVER",MB_OK);
                return 0;

        case XTYP_CONNECT:
                for (i=0;i<CONV_MAX;i++) {
                        if (!This->HConv[i]) {
                                This->OpenMouth = i;
                                return (HDDEDATA)1;
                        }
        }
                return 0;

        case XTYP_CONNECT_CONFIRM:
                This->HConv[This->OpenMouth] = hConv;
                break;

        case XTYP_DISCONNECT:
                 for (i=0;i<CONV_MAX;i++)
                        if (hConv == This->HConv[i])
                                This->HConv[i]=0;
                                        break;

        case XTYP_ERROR:
                This->MessageBox("A critical DDE error has occured.", This->Title,
                                        MB_ICONINFORMATION);
                break;

        case XTYP_EXECUTE:
                return DDE_FNOTPROCESSED;

        case XTYP_POKE: {
                This->anyReceivedData(hConv,hData,hsz1,hsz2);
                return (HDDEDATA)DDE_FACK;
         }

        case XTYP_REQUEST:
                if (This->MatchTopicAndItem(hsz1, hsz2,t1,t2))  {
                        This->LoadDataReq(t1,t2);
                  return This->DataRequested(wFmt);
                }
                return 0;

        case XTYP_WILDCONNECT:
```

```
                    return This->WildConnect(hsz1, hsz2, wFmt);

                case XTYP_REGISTER:
                    break;

               default:
                    break;
           }
           return 0;

          }
```

This CALLBACK function is based on (Borland,1993a). The case arguments match the different DDE messages (oulined in Section 3.4). The variable **This->** is used to refer the window object. It is assigned during the construction of the window class. Since it is a global it can be used in the CALLBACK function to access the Client/Server member functions.

Working together, the DDE CALLBACK and the Client/Server class provide the functionality for simple DDE communication. Note however that the DDE protocol is not strictly enforced - use of meanings of server, topic and item have been modified to allow simpler implementation. Future revisions would clean up and complete the DDE class to insure complete compatibility (see section 6.3).

Chapter 5 : DEVELOPING SIMULATIONS

Using the Simulation Engine to model a process is relatively easy so long as the designer is aware of the strengths and weaknesses of the program. The Simulation Engine has a powerful Lisp-like language which can describe anything from high-level mathematical formulae to random events to approximations. This language is tied to either manual or automatic *pulse* events which can be used to simulate timed events. The user interface is based on graphic elements or *tags*, which can be linked to data entry and script files, allowing for excellent user control and manipulation of the modelled process. Virtually speaking, any process can be modelled.

The pitfalls of this system become apparent when the design leaves the original design paradigm; that complex processes should be modelled based on a combination of operator experience and mathematical formulae. If a process can be modelled successfully using equations, then the Simulation Engine and its Lisp-like script language would not be the best choice. If these equations were influenced by unknown factors (factors that can not be mathematically modelled with ease), then the Simulation Engine will allow you to combine them. For a more complete discussion of the applications, limitations and some planned improvements to the Simulation Engine, refer to chapter 6.

Section 5.1: Building A Simple Simulation

This section describes the steps of building a simple simulation. The process to be simulated is the use of a pump filling a storage tank. To add some complexity to the process, we will assume that the substance being pumped contains a sticky particulate which quickly gathers and lowers the efficiency of the process. We will indicate the tank level graphically and with text as well as giving the current pump efficiency, tank overflow and the number of simulated minutes. A graphic will be used to indicate when the pump's efficiency has dropped below fifty percent, and a tag to flush the pump line will be added.

The key to this simulation is the pump's efficiency. It can be determined (by examining the gathered knowledge on the pump) that its "clean" rate is 500 liters per minute and that its efficiency drops about ten percent per minute after being flushed. (Note: these values are meant as examples only.) In this simple simulation the key rule is easily identified. A safe rule-of-thumb is that the efficiency of the pumping process will be reduced 8-12 percent every minute, unless a line purge has just been done.

The two entities in the simulation are a pump and a tank. These sub-processes need to be modelled in order to complete the main process. In order to do this, I have adapted a naming convention for naming these entities. Each entity has a main name, eg. tank or pump. This is followed by an underscore and a unique number, which is followed by another underscore and a specific variable name(i.e. pump_01_eff). When calling a prebuilt script which models a certian type, the convention simply puts the id number in a local Lisp variable called *id.* The script can build the necessary DDE query strings, extract the data from the Simulation State, and process the generic code. For example, here is a simple pump model based on rate and efficiency:

```
;     pump.scr
;           requires variable id to be set
;           returns pump_out
;
;   build dde query strings
;
(setq pumps 'pump_)
(strcat pumps id)
(setq s1 pumps)
(setq s2 pumps)
(strcat s1 '_eff)
(strcat s2 '_rate)
;
; get data
;
(ddesetq 0 pump_eff100      s1)
(ddesetq 0 pump_rate     s2)
(setq pump_eff (/ pump_eff100 100))
;
;   calculate new values
;
(setq pump_out ( * pump_rate pump_eff))
```

At this point the generic pump script is very simple. As our understanding of pump behavior improves we will modify this one file - and improve the entire simulation. Here is a code fragment which illustrates the use of this script file:

```
(setq id '01)
(load 'pump.scr)
```

The *id* variable is loaded with the id number of the pump being accessed. A similar approach is used for the tank entity:

```
;     tank.scr
;           requires variable id to be set
;           returns level
;
; build strings for dde query
;
(setq tanks 'tank_)
(strcat tanks id)
(setq s1 tanks)
(setq s2 tanks)
(setq s3 tanks)
(strcat s1 '_level)
(strcat s2 '_overflow)
(strcat s3 '_max)
(ddesetq 0 old_tank_level      s1)
(ddesetq 0 old_tank_overflow   s2)
(ddesetq 0 tank_max            s3)
;
;   calculate new values
;
(setq tank_level ( + old_tank_level tank_intake ))
(cond ((> tank_level tank_max)
         (setq tank_overflow ( + old_tank_overflow ( - tank_level tank_max )))) (t nil))
)
(cond ((> tank_overflow 0) (setq tank_level tank_max) (t nil)))
;
;   send out new values
;
(ddesendq 0 tank_level    s1)
(ddesendq 0 tank_overflow s2)
(setq level tank_level)
```

This approach is simple and allows incremental refinement. (An improved approach to building the query strings is required and is discussed in section 6.3).

With the sub-processes modelled, including any sub-processes they might contain, and any complex mathematical formulae required, the rules-of-thumb can be examined. In this example there is one rule; "the efficiency of the pumping process will be reduced 8-12 percent every minute, unless a line purge has just been done". A line purge resets the efficiency to 100 percent. In order to accomplish this, a state will be created, "pump_01_purged", which will be set to an "1" if a purge just occurred and a "0" if a purge has not happened. The rule can then be implemented in the "lub.scr" script file, which happens once every simulation *pulse*. (This is assuming that each *pulse* will represent one minute.) The script would then look like:

```
;
;    pump_eff.scr
;
;    script calculates the efficiency of pump_01
;
;
;
(ddesetq 0 pe 'pump_01_eff)
(ddesetq 0 pl 'pump_01_purged)
(cond ((< pl 1.0) (setq pe (* pe (/ (random 88 92) 100)))) (t nil))
(ddesendq 0 pe     'pump_01_eff)
(ddesendq 0 0.0    'pump_01_purged)
```

The corresponding script file, "purge.scr", will be required to set the "pump_01_purged" state to a value of one. This script will also be responsible for resetting the current pump efficiency:

```
;    purge.scr
;
;    purge pump_01's line
;
(ddesendq 0 1.0 'pump_01_purged)
(ddesendq 0 100.0 'pump_01_eff)
(load 'alarm.scr)
(ddesendq 2 'refresh 'refresh)
```

Unlike the efficiency script, this routine will need to be triggered by the operator by using a special *tag* linked to the "purge.scr" file.

When the I/O Handler sends a *StartSimulation* message to the Knowledge Base the script "init.scr" will be executed. This script allows the simulation designer a single opportunity to initialize the simulation states. This same script should load another script responsible for loading the initial screen. In this case a initial script could look like:

```
;     init.scr
;
;     send out initial values
;
(ddesendq 0 0 'quantum)
;
;     pump_01
;
(ddesendq 0 100.0  'pump_01_eff )
(ddesendq 0 500.0  'pump_01_rate)
(ddesendq 0 1.0    'pump_01_purged)
;
;     tank_01
;
(ddesendq 0 0.0     'tank_01_level)
(ddesendq 0 5000.0 'tank_01_max)
(ddesendq 0 0.0     'tank_01_overflow)
;
;     load initial screen
;
(load 'tanksim.scr)
```

This script also initializes the system variable *quantum*. This state represents the number of pulses the Simulation Engine has generated. In this specific case, we are using one *quantum* to represent one minute.

With these script files built, all that remains is the construction of an interface for the operator. To minimize the time spent doing this, it is a good idea to build background graphics. A background image can be drawn using a paintbrush program (e.g. Paint Shop Pro) which can draw .BMP files and provide x, y coordinates. Once the layout of the display screen is constructed, locate and record the x, y, dx, dy values where you would like to place the different dynamic graphic elements (e.g. analog outputs, changeable graphics). The remaining image is loaded at run time and the dynamic images placed on top of the background image. For the simple pump simulation, figure 5.1 shows the background file “tanksim.bmp”.

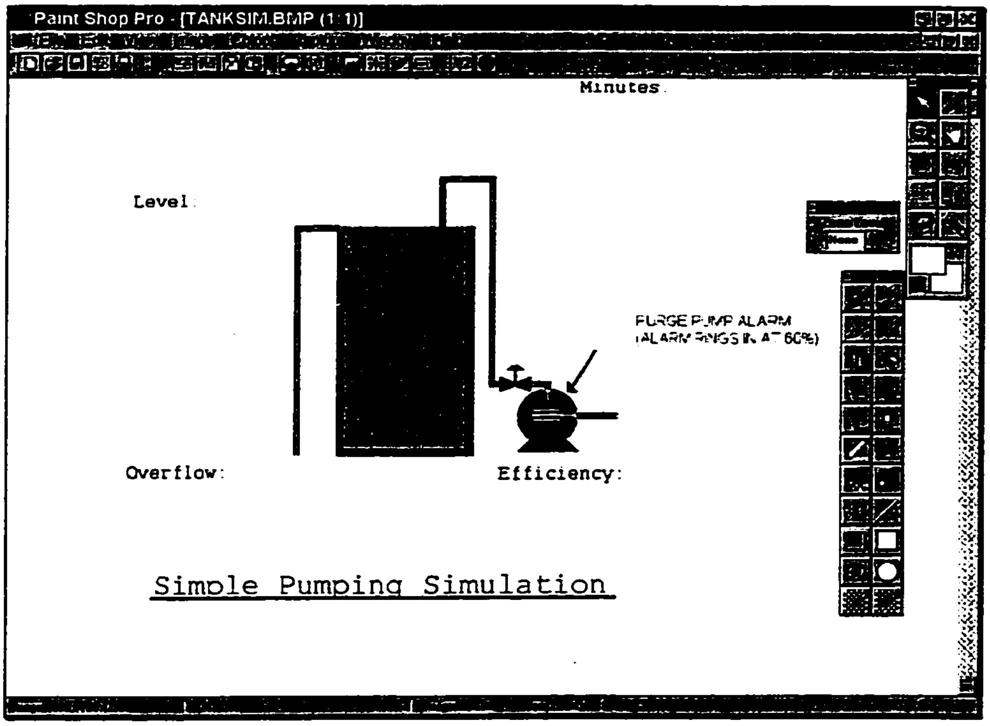


Figure 5.1 : Building A Simulation Background

The missing numeric displays (analog outputs) are tank_01_level, tank_01_overflow, pump_01_efficiency and minutes (quantum). Graphic elements that are missing include an efficiency alarm, a pulse button, a stop button, and a purge button. A simple animation was added to represent the current tank level using a region fill inside the tank. All these missing display elements, or *tags*, are initialized in "tanksim.scr", which is called from "init.scr" when the simulation initially runs. It is given below:

```
;        tanksim.scr
;
;        load tags for the simulation
;
(ddesendq 2 'flushtags 'flushtags)
;
(ddesendq 2 '0,0,1000,700,tanksim.bmp,tanksim 'simplegraphic)
(ddesendq 2 '700,0,100,25,0,quantum,nil 'smarttext)
;
(ddesendq 2 '900,10,50,50,rarrow.bmp,heart_beat,lub.scr 'smartgraphic)
(ddesendq 2 '950,10,50,50,stop.bmp,stop_beat,exit.scr 'smartgraphic)
(ddesendq 2 '800,350,200,75,buttup.bmp,button_01,button.scr 'smartgraphic)
(ddesendq 2 '600,250,200,75,alarmok.bmp,alarm_01,alarm.scr 'smartgraphic)
(ddesendq 2 '665,420,100,25,EFF%:,pump_01_eff,pump_01_eff,nil 'smartpopuptext)
(ddesendq 2 '220,125,100,25,0,tank_01_level,nil 'smarttext)
```

```
(ddesendq 2 '250,420,100,25,0,tank_01_overflow,nil 'smarttext)
(ddesendq 2 '356,170,130,235,0,200,0,tank_level 'simpleregion)
;
(load 'lub.scr)
(ddesendq 2 'refresh 'refresh)
```

The type of tags used define the operator interface. For example, consider the *smartpopup* tag "pump_01_eff". Not only will this tag display values as they arrive from the simulation state and run a script file if selected, it provides the operator a pop-up dialog which allows him to change the "pump_01_eff" state directly.

Once these tags are loaded the script file forces one pulse to run by loading the "lub.scr" script file. This file runs automatically whenever a simulation heartbeat or pulse occurs. In this simple example a *smartgraphic* called heart_beat will run the "lub.scr" whenever the operator selects it. This file will call all the other files, and will signal the Event Handler with a *pulse* message. It is given below:

```
;     lub.scr
;
;     heart_beat
;
(ddesetq 0 x 'quantum)
(ddesendq 1 'pulse 'pulse)
(setq x (add1 x))
(ddesendq 0 x 'quantum)
;
;     modify pump_01 efficiency
;
(load 'pump_eff.scr)
;
;     calculate pump output
;
(setq id '01)
(load 'pump.scr)
(setq tank_intake pump_out)
;
;     calculate new tank level
;
(setq id '01)
(load 'tank.scr)
(load 'tankani.scr)
;
;     calculate any new alarms
;
```

```
(load 'alarm.scr)
(ddesendq 2 'refresh 'refresh)
```

This script first does the overhead work such as incrementing the *quantum* state and sending out the *pulse* message. It can then process the simulation.

This simple simulation calculates the current pump efficiency, calculates the pump output, the new tank level, checks for new alarms and finally sends a generic screen refresh to the I/O Handler. Most of the scripts called from here were already built; "pump_eff.scr", "pump.scr", "tank.scr" are described above. The result of their calculations are sent to the Simulation State which will send them out to the I/O Handler for updating. The new scripts are designed to add to the operator interface. They are "tankani.scr" and "alarm.scr".

The script "tankani.scr" manipulates the current tank level into x, y, dx and dy values and animates a region on the operator's display to reflect the new level.

It is given below:

```
;
;    tankani.scr
;
;    Animate the tank filling up
;
(setq s1 '356, )
(setq dy_fact (/ tank_level 20))
(setq y_fact (- 405 dy_fact))
(setq xtra (trunc y_fact))
(strcat s1 xtra)
(strcat s1 ',130, )
(setq xtra (trunc dy_fact))
(strcat s1 xtra)
(strcat s1 ',tank_level)
(ddesendq 2 s1 'changeregion)
```

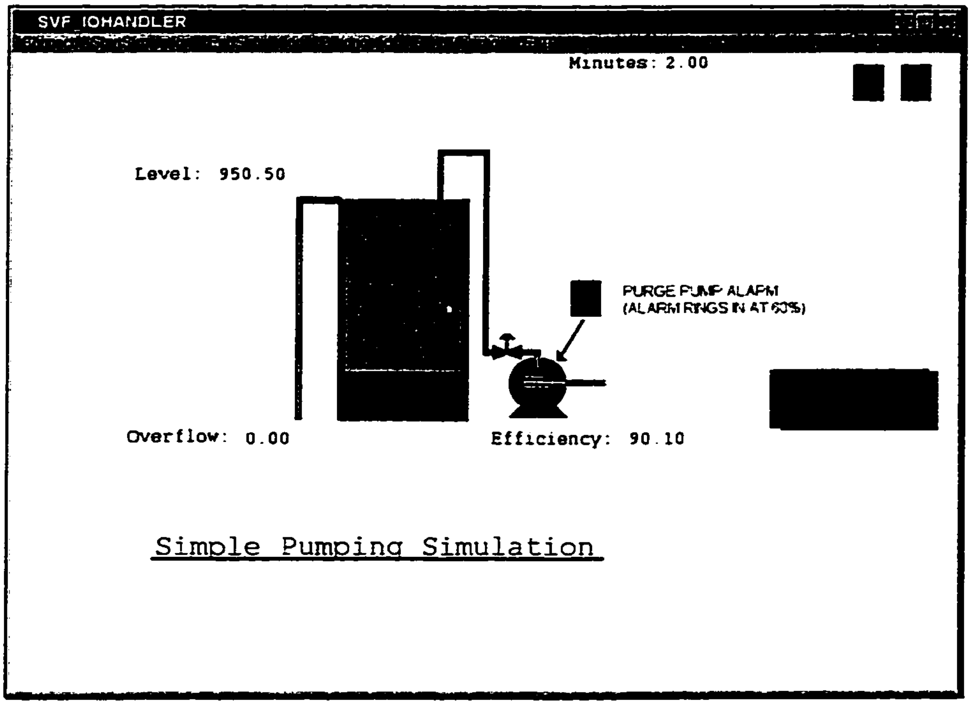
The script "alarm.scr" sets the tag "alarm_01" to the appropriate alarm graphic, depending on the pump's efficiency. In this example the alarm will be set if the efficiency is less than 65%. The code is given below:

```
;
;    purge pump alarm
;
(ddesetq 0 eff 'pump_01_eff)
(cond ((< eff 65.0 ) (ddesendq 2 'alarm_01,alarm.bmp 'changegraphic))
    ( (t (ddesendq 2 'alarm_01,alarmok.bmp 'changegraphic)) )
)
(ddesendq 2 'alarm_01 'refresh)
```

Independent to the "lub.scr" file is the script that is executed when the tag "button_01" is selected. This script file is called "button.scr" and looks like:

```
;
;     button.scr
;
(ddesendq 2 'button_01,buttdn.bmp 'changegraphic)
(ddesendq 2 'button_01 'refresh)
(ddesendq 2 'button_01 'refresh)
(ddesendq 2 'button_01 'refresh)
(ddesendq 2 'button_01,buttup.bmp 'changegraphic)
(ddesendq 2 'button_01 'refresh)
(load 'purge.scr)
```

This script "blinks" the button the operator selects. Multiple refreshes force a noticeable delay. Once the little animation is complete, the "purge.scr" file is loaded (described above) and pump efficiency is reset to 100%. The final product is given in figure 5.2.



Minutes: 2.00

Level: 950.50

Overflow: 0.00

Efficiency: 90.10

Figure 5.2 : A Simple Pump Simulation

This simulation is only meant to be a simple example. The process it was (loosely) modelled after is actually much more complex. The purging process does not necessarily work all the time, "best two out of three" was how one operator described it. The pump's efficiency is never at 100%, even after a perfect purging. Other factors that could be taken into account are the pressure, density, temperature and "stickiness" of the fluid. By breaking the simulation into smaller script files the simulation can be extended and developed as new information is gathered.

Section 5.2 : The Basic Kamyr Model of a Continuous Digester

This section discusses the design of a Digester simulation based on the manufacturer's specifications. The goal of this simulation is to develop a graphic representation of the Digester process and to introduce a basic set of "rules-of-thumb" into the model.

One of the biggest problems encountered in industry today is the training of employees responsible for the operation of expensive, dangerous equipment. In the production of kraft pulp, proper control of pumps, tanks, steam flows and chemical flows are critical to production and safety. An 'operator' must be fully competent *before* he/she takes command of the process. Unfortunately, traditional training requires that the operator train on active equipment under supervision of an experienced operator. The trainee may not experience any emergency situations; since this is the actual process, it is not economical to lose production or risk lives for training purposes. What is required is the ability to produce simulations of the process which can be used to expose and train employees.

Usually any such simulation is based on an accurate mathematical formula. In the case of an industrial based system, this is not always possible. Specifically, the digester process has proven to be very difficult to model. In order to properly simulate such an ambiguous system requires a combination of known mathematical formulae and a collection of rules and probable happenings. The *perfect* digester model is outlined in the Kamyr Digester Operating Manual (1993). This set of equations provides an excellent starting point in the modeling of the system. The three main equations are:

1. Production (PROD) based on chip input.

R - Chip Meter (RPM).
F - Filling Degree of Chip Meter (%).
V - Chip Meter Volume (cubic ft per revolution).
W - Chip Meter Volume (lbs/cubic ft).
Y - Yield, %Bone Dry Pulp to %Bone Dry Wood.

PROD = (R x F x V x W x Y) / 12500

This is implemented using the script "prod.scr":

```
;
; Production (PROD) based on chip input
;
(ddesetq 0 r    'chip_meter_rpm)
(ddesetq 0 f    'chip_meter_fill%)
(ddesetq 0 v    'chip_meter_volume)
(ddesetq 0 w    'chip_weight_ovendry)
(ddesetq 0 y    'pulp_from_chip_yield)
(ddesetq 0 tot  'total_prod)
(ddesetq 0 oldp 'hourly_prod)
(setq new_hprod (hour_prod r f v w y))
(setq new_tot (+ tot new_hprod))
(ddesendq 0 new_hprod  'hourly_prod)
(ddesendq 0 new_tot    'total_prod)
(ddesendq 0 (* 24.0 new_hprod) 'target_prod)
(ddesendq 2 'refresh   'refresh)
```

2. Chemical Flow (W) based on target production (PROD).

A - Active Alkali (cooking chemical) (%AA).
The Active Alkali (or alkali charge) is usually 12 to 14% for softwood and 8 to 10% for hardwood. (Smook, 1994 p. 81) It is set by the operator.
S - Chemical content (lbs/cubic ft).
PROD - Production (tons / day).
Y - Yield, %bone dry pulp to %bone dry wood.

(a) calculate bone dry pulp: BDP = (PROD x 0.9 x 2000) / 24.0
(b) calculate bone dry wood: BDC = (BDP x 100.0) / Y
(c) calculate lbs/hr active alkali charge: AAC = (BDC x A) /100.0
(d) calculate chemical flow: W = (AAC) / S (cubic ft / hour)

This is implemented using the script "chem.scr":

```
;     chem.scr
;
;     Chemical Flow (W) based on target production (PROD)
;
(ddesetq 0 A 'active_alkali%)
(ddesetq 0 S 'chemical_content)
(ddesetq 0 Y 'pulp_from_chip_yield)
(ddesetq 0 HPROD 'hourly_prod)
(setq PROD (* HPROD 24.0))
;
(setq BDP (/ (* PROD (* 0.9 2000.0)) 24.0))
(setq BDC (/ (* BDP 100.0) Y))
(setq AAC (/ (* BDC A) 100.0 ))
(setq W   (/ AAC S))
; W is now in ft cubed per hour; convert to gallons per minute
; (GPM is the preferred unit to display)
(setq WGPM (/ (* W 7.48) 60))
(ddesendq 0 WGPM 'chemical_flow)
(ddesendq 2 'chemical_flow 'refresh)
```

3. Temperature based on production (PROD) and cooking time.

T - Time in hours chips will cook for.

RR - Relative rate of cooking, referenced in a table by temperature.

TEMP - Temperature required. According to Smook (1994), using temperatures higher than 180 deg C (356 deg F) will damage the product.

H-Factor - Ratio value of relative rate of cooking (lignin breakdown) vs. temperature. Notice that H-Factor is an experimentally developed value which acts as a quality control value. The operator can control time and temperature based on keeping the H-Factor constant. The H-Factor is proportional to a value called the K-Number, an actual measurement of quality in the pulp. If the operator can keep the H-Factor constant the desired quality of product will be achieved. H-Factor is calculated:

H-Factor = T x RR (at a specified temperature)

For example:

Let T = 1.8 and temperature be 320 deg F.
From relative rate table find that RR at 320 is 397.8.

H-Factor = 1.8 x 397.8 = 716.04.

(a) to increase production (PROD) to a new production (PROD_NEW).

T_NEW = (PROD)/(PROD_NEW) x T

(b) calculate a new relative rate (RR_NEW) to keep H-Factor constant.

RR_NEW = (H-Factor)/(T_NEW)

(c) lookup new relative rate up on relative rate table to find temperature(TEMP). The relative rate vs temperature is implemented using script "ratetemp.scr" :

```
;     ratetemp.scr
;
;     Return the temperature (TEMP) based on Relative Rate (RR)
;     inputs required : RR
;     outputs         : TEMP
;
(cond ((> RR 53.1) (setq TEMP 280)) (t nil)))
(cond ((> RR 56.0) (setq TEMP 281)) (t nil)))
(cond ((> RR 59.1) (setq TEMP 282)) (t nil)))

... [temp 283 to 354 omitted]...

(cond ((> RR 1968.8) (setq TEMP 355)) (t nil)))
(cond ((> RR 2056.7) (setq TEMP 356)) (t nil)))
(cond ((> RR 2148.3) (setq TEMP 357)) (t nil)))
(cond ((> RR 2243.7) (setq TEMP 358)) (t nil)))
(cond ((> RR 2343.1) (setq TEMP 359)) (t nil)))
```

The calculations are done in script file "termp.scr" :

```
; temp.scr
;
;     Temperature based on production (PROD) and cook_time
;     (should be called when a change of production happens)
;
(ddesetq 0 T 'cook_time)
(ddesetq 0 RR 'relative_rate)
(ddesetq 0 HPROD 'hourly_prod)
(ddesetq 0 HFACTOR 'h_factor)
(setq PROD_NEW (* HPROD 24.0))
(ddesetq 0 PROD_OLD 'target_prod)
(setq T_NEW (* (/ PROD_OLD PROD_NEW) T))
(setq RR (/ HFACTOR T_NEW))
; get TEMP via rate-temp  script
(load 'ratetemp.scr)
(ddesendq 0 TEMP 'temperature)
(ddesendq 0 T_NEW 'cook_time)
(ddesendq 0 RR 'relative_rate)
(ddesendq 0 (* T_NEW TEMP) 'h-factor)
(ddesendq 2 'temperature 'refresh)
(ddesendq 2 'cooktime 'refresh)
(ddesendq 2 'relative_rate 'refresh)
(ddesendq 2 'h-factor 'refresh)
```

As indicated earlier, these formulae are theoretical only. In practice they have proven to be a better guide than a strictly enforced rule. Generally speaking, an operator will try operate at a maximum production rate at a maximum temperature (about 330 deg F). In order to increase the process beyond the original formula, operations will manipulate all the variables in ways not accounted for by the formula. For example, one of the digesters I studied was rated at 800 tons/ day. It produces an average of 1100 tons per day - by using operator skill and knowledge, production has been increased 300 tons per day. This is discussed more in section 5.3.

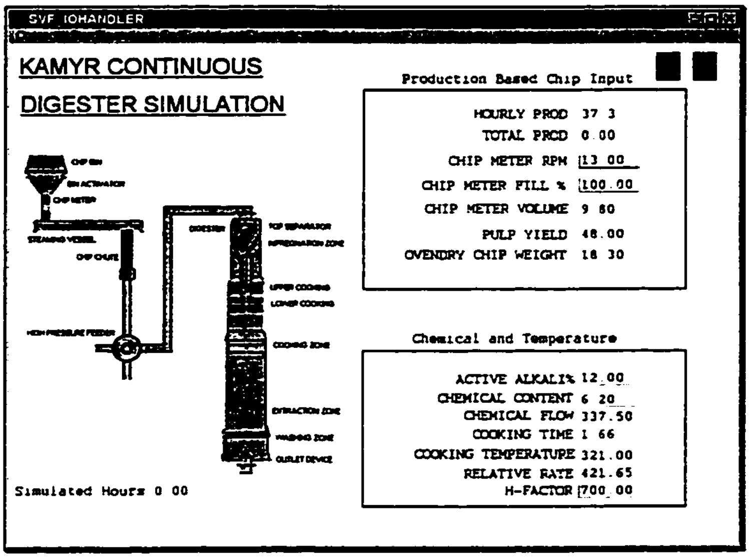
Having built the general scripts for dealing with the three main formulae, the design of the operator interface could begin. It was decided that the operator should see all the values and be allowed to input chip meter speed, chip meter fill percentage, chemical content, active alkali percentage and be allowed to adjust the H-Factor. These input points were linked to the corresponding scripts. Since a change in production affects all the values in the simulation, a new script was built, “hourprod.scr”:

```
;      hourprod - calculate the hourly production
;
(ddesetq 0 r 'chip_meter_rpm)
(ddesetq 0 f 'chip_meter_fill%)
(ddesetq 0 v 'chip_meter_volume)
(ddesetq 0 w 'chip_weight_ovendry)
(ddesetq 0 y 'pulp_from_chip_yield)
;
; assign hourly production the new target value
;
(setq x (hour_prod r f v w y))
(ddesendq 0 x 'hourly_prod)
;
; find out the chemical and temperature changes required
;
(load 'chem.scr)
(load 'temp.scr)
(ddesendq 0 (* x 24.0) 'target_prod)
(ddesendq 2 'refresh 'refresh)
```

The background for the user’s interface included a simple drawing of the process. Based on the actual control screens, this drawing was to contain output tags, alarms etc. Due to time constraints this work was not done; instead a simpler text display was created beside the drawing. The I/O Handler receives its initial tags from "chipinit.scr":

```
;     chipinit.scr
;     initialize the chip screen
;
(ddesendq 2 'flushtags 'flushtags)
(ddesendq 2 '0,0,1000,700,kamyr.bmp,tag1              'simplegraphic)
(ddesendq 2 '200,625,100,20,0,quantum,nil               'smarttext)
(ddesendq 2 '900,10,50,50,rarrow.bmp,heart_beat,lub.scr  'smartgraphic)
(ddesendq 2 '950,10,50,50,stop.bmp,initx,reset.scr       'smartgraphic)
(ddesendq 2 '800,90,50,25,0,hourly_prod,hourprod.scr    'smarttext)
;
(ddesendq 2 '800,120,100,25,0,total_prod,nil             'smarttext)
(ddesendq 2 '800,155,100,25,0,chip_meter_rpm,chip_meter_rpm,hourprod.scr
'smartpopuptext)
(ddesendq 2 '800,190,100,25,0,chip_meter_fill%,chip_meter_fill%,hourprod.scr
        'smartpopuptext)
(ddesendq 2 '800,225,100,25,0,chip_weight_ovendry,nil     'smarttext)
(ddesendq 2 '800,260,100,25,0,pulp_from_chip_yield,nil   'smarttext)
(ddesendq 2 '800,290,100,25,0,chip_meter_volume,nil      'smarttext)
(ddesendq 2 '800,465,100,25,0,active_alkali%, active_alkali%, chem.scr
        'smartpopuptext)
(ddesendq 2 '800,495,100,25,0,chemical_content,chemical_content,chem.scr
        'smartpopuptext)
(ddesendq 2 '800,520,100,25,0,chemical_flow,nil          'smarttext)
(ddesendq 2 '800,545,100,25,0,cook_time,nil              'smarttext)
(ddesendq 2 '800,575,100,25,0,temperature,nil            'smarttext)
(ddesendq 2 '800,600,100,25,0,relative_rate,nil          'smarttext)
(ddesendq 2 '800,625,100,25,0,h_factor,h_factor,temp.scr 'smartpopuptext)
```

The resulting simulation is shown in figure 5.3.



KAMYR CONTINUOUS
DIGESTER SIMULATION

```
Production Based Chip Input

           HOURLY PROD  37 3
            TOTAL PROD  0 00
       CHIP METER RPM  13 00
      CHIP METER FILL %  100.00
     CHIP METER VOLUME  9 80
           PULP YIELD  48.00
  OVENDRY CHIP WEIGHT  18 30

Chemical and Temperature

        ACTIVE ALKALI%  12.00
      CHEMICAL CONTENT  6 20
         CHEMICAL FLOW  337.50
          COOKING TIME  1 66
   COOKING TEMPERATURE  321.00
         RELATIVE RATE  421.65
              H-FACTOR  700 00

Simulated Hours 0 00
```

Figure 5.3 : The Kamyr Continuous Digester Simulation

In this simulation only the most basic Kaymr rules were implemented. Further expansion of this simulation is possible using some of the known characteristic behaviours of the pulping process. As an example, consider the following basic K-number guidelines:

- A one percent.Active Alkali change will change K-number by 2 units
- A one degree (F) temperature change will change K-number by 1.5 units
- A one RPM chip meter change will change K-number by 1 - 1.5 units

These rules will have to be incorporated to affect the predicted K-number value. When combined with other qualitative rules, such as :

- plugged extraction screens will stop chemical flow and reduce temperature
- if a high percentage of chips are pins (small, toothpick-like chips) screens are more likely to plug

The simulation should be able to examine incoming chips and predict screen plugging probability. Once the screens become plugged, the simulation should account for a reduction in temperature. The reduced temperature will then increase the K-number, which will result in a lower quality, less cooked final product. These rules begin the process of producing a complex simulation; the Simulation Engine allows for future growth as input from operators and process engineers is gathered.

Rather than proceed with building in the rest of the complex rules into this simulation, it was identified that the operators would be able to build a set of rules to calculate the various inputs based on their experience. This model is used in section 5.3.

Section 5.3 : The Avenor ISO Model

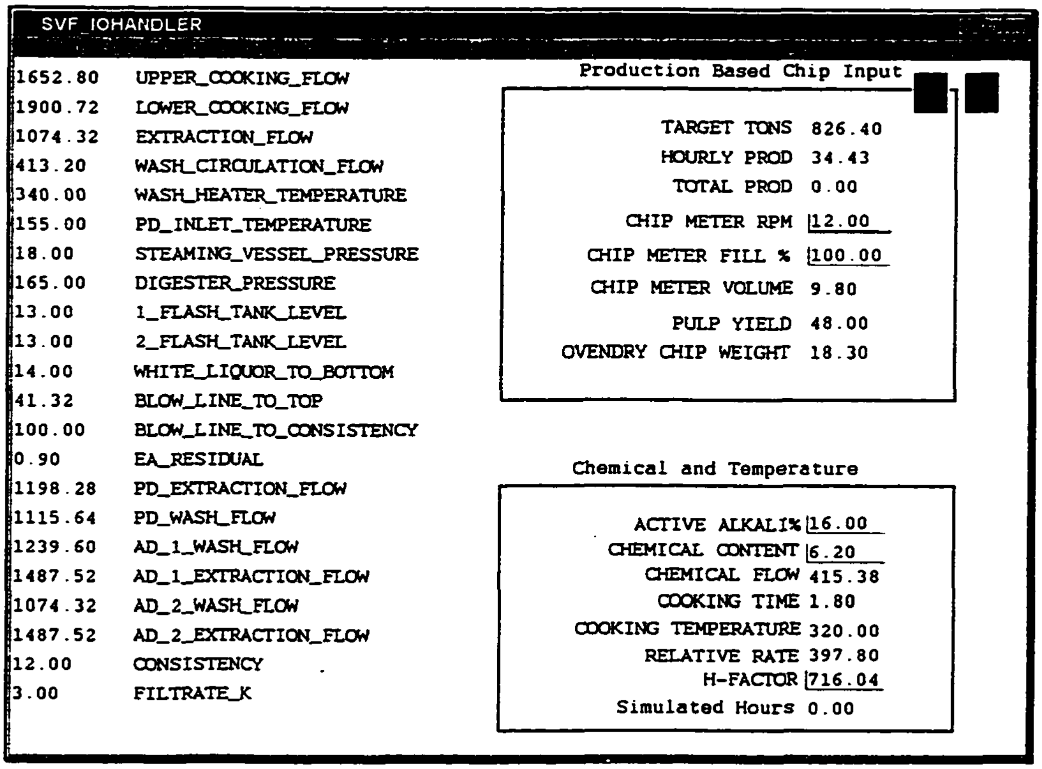
This section extends the basic Kamyr model of a digester by integrating information gathered by the operating staff at a particular mill site. The actual numbers being used in the formulae will not be replicated in respect of the company's privacy. This does not compromise the idea, however.

When examining the original simulation model it becomes evident that many variables are not taken into account. Some of these variables include:

UPPER_COOKING_FLOW
LOWER_COOKING_FLOW
EXTRACTION_FLOW
WASH_CIRCULATION_FLOW
WASH_HEATER_TEMPERATURE
PD_INLET_TEMPERATURE
STEAMING_VESSEL_PRESSURE
DIGESTER_PRESSURE
1_FLASH_TANK_LEVEL
2_FLASH_TANK_LEVEL
WHITE_LIQUOR_TO_BOTTOM
BLOW_LINE_TO_TOP
BLOW_LINE_TO_CONSISTENCY
EA_RESIDUAL
PD_EXTRACTION_FLOW
PD_WASH_FLOW
AD_1_WASH_FLOW
AD_1_EXTRACTION_FLOW
AD_2_WASH_FLOW
AD_2_EXTRACTION_FLOW
CONSISTENCY
FILTRATE_K

A good operator will set these based on the targeted production. (Recall that targeted production is based on the current chip input variables given in section 5.2.) Rather than expand the original model with additional formulae from Kaymr, it was decided to use the operating parameters that the operating staff had calculated. These are a set of standardized "rules" which use the targeted production to find these unknowns. This simulation will simply extend the previous one by providing these calculated values on the display screen instead of the operating diagram.

In order to produce this new simulation, the operating parameter rules were first built into a standalone simulation which simply allowed the operator to enter the target production. This model acted as a simplified expert system; the operator enters the production value and the model returns the values the operator should use. This model was then merged with the existing Kamyr model, allowing the target tons to be based on the chip variables. The resulting simulation is shown in figure 5.4.



1652.80	UPPER_COOKING_FLOW
1900.72	LOWER_COOKING_FLOW
1074.32	EXTRACTION_FLOW
413.20	WASH_CIRCULATION_FLOW
340.00	WASH_HEATER_TEMPERATURE
155.00	PD_INLET_TEMPERATURE
18.00	STEAMING_VESSEL_PRESSURE
165.00	DIGESTER_PRESSURE
13.00	1_FLASH_TANK_LEVEL
13.00	2_FLASH_TANK_LEVEL
14.00	WHITE_LIQUOR_TO_BOTTOM
41.32	BLOW_LINE_TO_TOP
100.00	BLOW_LINE_TO_CONSISTENCY
0.90	EA_RESIDUAL
1198.28	PD_EXTRACTION_FLOW
1115.64	PD_WASH_FLOW
1239.60	AD_1_WASH_FLOW
1487.52	AD_1_EXTRACTION_FLOW
1074.32	AD_2_WASH_FLOW
1487.52	AD_2_EXTRACTION_FLOW
12.00	CONSISTENCY
3.00	FILTRATE_K

Production Based Chip Input

TARGET TONS	826.40
HOURLY PROD	34.43
TOTAL PROD	0.00
CHIP METER RPM	12.00
CHIP METER FILL %	100.00
CHIP METER VOLUME	9.80
PULP YIELD	48.00
OVENDRY CHIP WEIGHT	18.30

Chemical and Temperature

ACTIVE ALKALI%	16.00
CHEMICAL CONTENT	6.20
CHEMICAL FLOW	415.38
COOKING TIME	1.80
COOKING TEMPERATURE	320.00
RELATIVE RATE	397.80
H-FACTOR	716.04
Simulated Hours	0.00

Figure 5.4 : Operating Parameter Simulation

The circular graphic in the upper right was provided to indicate when the Knowledge Base was working. It changes colour (RED, ORANGE, YELLOW, and GREEN) depending on which script file is executing. The result tells the operator that the system is busy, not to use the calculated numbers yet.

The modifications included adding the new variables to the "init.scr" script, updating the screen tags so that the variables would be displayed, copying and modifying the background image, and adding a script file that does the actual calculations for the operating parameters. A partial script is given below:

```
;          stpd.scr
;
;          This script performs the calculations as determined by the operating staff.
;
(ddesetq 0 stpd 'stpd)
(ddesendq 0 (* 2 stpd) 'UPPER_COOKING_FLOW )
```

```
(ddesendq 0 (* 2.3 stpd) 'LOWER_COOKING_FLOW )
(ddesendq 0 (* 1.3 stpd) 'EXTRACTION_FLOW )
... [calculations] ...
(ddesendq 0 (* 1.3 stpd) 'AD_2_WASH_FLOW )
(ddesendq 0 (* 1.8 stpd) 'AD_2_EXTRACTION_FLOW )
;
(ddesendq 2 'UPPER_COOKING_FLOW  'refresh)
(ddesendq 2 'LOWER_COOKING_FLOW  'refresh)
(ddesendq 2 'EXTRACTION_FLOW  'refresh)
... [transfer data to IO/HANDLER] ...
 (ddesendq 2 'AD_2_WASH_FLOW 'refresh)
(ddesendq 2 'AD_2_EXTRACTION_FLOW 'refresh)
(ddesendq 2 'stpd 'refresh)
```

Along with a new I/O Handler screen initializing script was written to add new tags. Note that no new input tags were added, since all the inputs were in place from the Kamyr model. The script is partially listed below:

```
;     avenor.scr
;
;     Revised simulation to include operating variables
;
(ddesendq 2 'flushtags 'flushtags)
(ddesendq 2 '0,0,1000,700,avenor.bmp,digesterbmp 'simplegraphic)
;
;
;
(ddesendq 2 '120,10,400,25,UPPER_COOKING_FLOW,isovar1 'simpletext)
(ddesendq 2 '120,40,400,25,LOWER_COOKING_FLOW,isovar2  'simpletext)
(ddesendq 2 '120,70,400,25,EXTRACTION_FLOW,isovar3  'simpletext)
(ddesendq 2 '120,100,400,25,WASH_CIRCULATION_FLOW,isovar4  'simpletext)
...[screen text]...
 (ddesendq 2 '120,610,400,25,CONSISTENCY,isovar22  'simpletext)
(ddesendq 2 '120,640,400,25,FILTRATE_K,isovar23  'simpletext)
;
(ddesendq 2 '0,10,400,25,0,UPPER_COOKING_FLOW,nil  'smarttext)
(ddesendq 2 '0,40,400,25,0,LOWER_COOKING_FLOW,nil  'smarttext)
...[input/output tags]...
 (ddesendq 2 '0,610,400,25,0,CONSISTENCY,nil  'smarttext)
(ddesendq 2 '0,640,400,25,0,FILTRATE_K,nil  'smarttext)
;
; from the original Kamyr model
;
(ddesendq 2 '900,10,50,50,rarrow.bmp,heart_beat,lub.scr   'smartgraphic)
...[refer to section 5.2] ...
 (ddesendq 2 '800,625,100,25,0,h_factor,h_factor,temp.scr  'smartpopuptext)
(ddesendq 2 '800,655,100,20,0,quantum,nil                'smarttext)
```

The result of these changes is to produce a model which can be used both to train and to act as an expert system. Further expansions to this simulation could include introducing random events such as screen plugging, poor chips and other anomalies which would then let the operator attempt to identify the problem based on the reported values. Another possibility is to experiment with different formulae that the operations group feel may work. Generally speaking, the simulation should now be turned over to the operations group and their process engineers to explore particular avenues. The Simulation Engine has proven to be a powerful and flexible tool.

Section 5.4 : Linking To Other Windows Applications

This section discusses how the Simulation Engine can be used to communicate information to other Window applications.

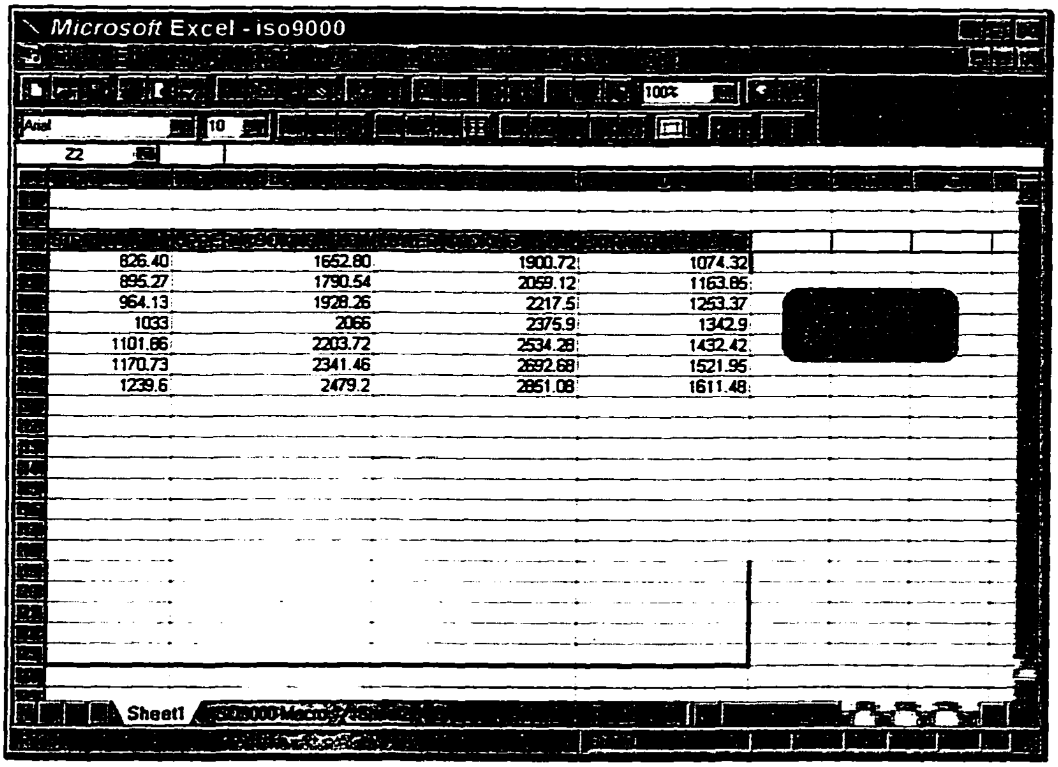
During the design phase of the Simulation Engine the decision was made to use the DDE protocol to communicate between the different processes that make up the Simulation Engine. This functionality permits other Window's applications to read data from the Simulation State.

As an example of this functionality, consider the popular PC spreadsheet application, EXCEL. This spreadsheet is capable of DDE inter-process communication and thus is capable of using the Simulation State as a DDE server. Assume that the task at hand is to record some of the calculated values that the simulation in Section 5.3 describes. A macro would need to be developed in EXCEL that would connect and gather data. As an example I have written one:

```
'
' OPERATINGVALUES Macro
'
'  This macro demostrates how the simulation engine data can be
'  accessed using the DDE protocol. It selects 4 data items from
'  the simulation state and collects them as rows of data.
'
'  Macro recorded 7/5/96 by Steven Falcigno
'
```

```
'
Sub OPVALS()

'
' First establish a DDE conversation
'
  channelNumber = Application.DDEInitiate( _
   app:="SVF_SIMULATION_STATE", _
   topic:="states")

'
' Increase the target row for outputing data
'
  oldx = Worksheets("Sheet1").Cells(1, 26)
  oldx = oldx + 1
  Worksheets("Sheet1").Cells(1, 26).Formula = oldx
  oldx = oldx + 2

'
' Issue requests to the SIMULATION_STATE for data.
' Once the data has arrived, write it to the spreadsheet.
'
  returnList = Application.DDERequest(channelNumber, "stpd")
  Worksheets("Sheet1").Cells(oldx, 1).Formula = returnList(1)
  returnList = Application.DDERequest(channelNumber, "UPPER_COOKING_FLOW")
  Worksheets("Sheet1").Cells(oldx, 2).Formula = returnList(1)
  returnList = Application.DDERequest(channelNumber, "LOWER_COOKING_FLOW")
  Worksheets("Sheet1").Cells(oldx, 3).Formula = returnList(1)
  returnList = Application.DDERequest(channelNumber, "EXTRACTION_FLOW")
  Worksheets("Sheet1").Cells(oldx, 4).Formula = returnList(1)


'
' Terminate the DDE conversation
'
  Application.DDETerminate channelNumber

End Sub
```

The language is reasonably complicated, but once a macro is written it is a simple enough process to link it a push-button on the spreadsheet itself. The information gathered here can be used in statistical analysis or in comparing the simulation to the real world process. Figure 5.5 shows the output of the above macro after seven iterations of changing chip RPM from 12 to 18 (which changes the target production from 826 tons to 1239 tons):



826.40	1652.80	1900.72	1074.32
895.27	1790.54	2059.12	1163.85
964.13	1928.26	2217.5	1253.37
1033	2066	2375.9	1342.9
1101.66	2203.72	2534.28	1432.42
1170.73	2341.46	2692.68	1521.95
1239.6	2479.2	2851.08	1611.48

Figure 5.5 : Excel Spreadsheet Communicating With The Simulation Engine

This simple example shows how the Simulation Engine could be linked to other Window's examples. This is a reasonable thing to expect, since this is how the Simulation Engine was constructed. Each process is a discrete Windows program which uses DDE to communicate. The result is that the Simulation Engine can easily talk to any Windows application supporting the DDE protocol.

Chapter 6 : CONCLUSIONS

The Simulation Engine was designed to be a virtual machine for building simulations that would require complex rule based events. It was to be built in a popular environment which could be inexpensively acquired. The Simulation Engine was meant to demonstrate knowledge in simulation design theory, object-oriented design and implementation using C++, inter-process communication protocols, and high level language development. In building this program over thirty non-trivial programming "experiments" were performed to test and learn the Windows/C++ programming environment. These range from the simple creation of a Windows application to the use of screen graphics to the advanced concepts of DDE protocol manipulation to the use of the Borland CLASSLIB to build advanced data structures. The final programs that make up the Simulation Engine were not written in one attempt, but were assembled from the preceding pieces, inherited from pre-built classes and expanded into their final form. The result is a highly advanced and flexible system of programs.

I believe that my Simulation Engine, although not perfect, is a very good example of advanced Computer Science system development. The remainder of this chapter discusses the future of the Simulation Engine by identifying its flaws, possible enhancements and a possible practical future as a simple control system.

Section 6.1: Practical Limitations to the Simulation Engine

Several issues have been identified with this Simulation Engine, both with the fundamental design and with the actual implementation. This section identifies these items, and recommends possible solutions.

Section 6.1.1 : Design Limitations

The Simulation Engine attempts to distribute the workload over several discrete processes. Two of these processes, the Simulation State and the Event Handler could be made superfluous. The Simulation State acts as a storage object which can be accessed from *any* other process in the system. With respect to the Simulation Engine, this functionality could be completely implemented by the Knowledge Base's high level language. That is, variables assigned in the Knowledge base retain their values until they are over-written or until the simulation ends. With some modification, the Knowledge Base could communicate any changes to the I/O Handler, thus rendering the Simulation State unessential. In order to maintain the flexibility that the Simulation State offers - allowing programs such as Excel to access to the data in the simulation - the Knowledge base would need to be extended. As a result the Knowledge Base's purpose would become confused. Instead, I recommend that the Simulation State be extended to have more intelligence and a higher functionality.

The Event Handler, however, requires some significant changes to continue to be a part of the Simulation Engine. Its original purpose, to allow scheduling of jobs to run a specific number of times, is easily done by the Knowledge Base. In order to retain the Event Handler in the Simulation Engine, it needs the functionality to completely process the job without putting further load on the Knowledge Base. That is, without calling the Knowledge Base's script language interpreter. This can be accomplished either by limiting the Event handler's functionality to specific tasks - e.g. variable ramping, addition, subtraction, etc. or by duplicating the complexity of the Knowledge Base's script language. I suggest that the Event Handler's functionality can be carried out by the Knowledge Base and that the Event Handler can be removed from the Simulation Engine.

Section 6.1.2 : Implementation Limitations

The choices made in the design of the Simulation Engine have forced some implementation sacrifices. Some of these issues are easy to resolve by extending the system (adding memory or drive space) and some will require more effort.

The DDE protocol uses a significant amount of system memory - even when the protocol is idle. The Simulation Engine generates a great deal of inter-process communication. The result is that the base PC computer to support the Simulation Engine needs to have at least 16 Megabytes of memory (I would recommend 24 as preferred). The choice of protocol is discussed in more detail in section 6.2.

Another memory intensive part of the Simulation Engine is the script language itself. Based on a Lisp interpreter I wrote in my fourth year of Computer Science, it was chosen because it was reasonably straight forward to implement and was quite capable of describing complex rules. The implementation is highly recursive and as a result requires a great deal of stack memory. As well, the script language does not implement any of the traditional iterators, such as *while*, *for* and *goto* loops. The result is a dependence on recursive style programming within the scripts. This consumes more memory. The language needs to be replaced with a graphic-oriented, iterative based solution. This is also discussed further in section 6.2.

The I/O Handler's tag objects are also limited by memory. The actual number of tags are limited by a statically defined amount, currently set to two hundred. This value can be changed as required by recompiling. A better solution would be a dynamic array with memory checking.

The I/O handler has a bigger problem. Some types of tags have not yet been developed - more region based tags and a greater diversity of input dialogs would add to the usability of the Simulation Engine. Some ideas include building a generic dialog system - allowing dialogs to be designed by the operator and integrated into the simulations. The designs would include graphic animated changes of set points. This functionality would require a significant programming effort.

The final major limitation encountered is the design of the simulation screens themselves. The creation of background .BMP files and the positioning of tags on them is a tedious and time consuming task. A better solution would be to spend some time building a graphic package that allows the design of the static background and the placement of dynamic tags on it. This package would then output the necessary script files that would load the tags at run time.

Many of these "limitations" are directed towards the improvement or enhancement of the Simulation Engine. The fundamental idea behind the Simulation Engine was to produce a system capable of simulating complex, rule based processes. The program requires fine tuning, a better protocol solution, and an improved language for rule implementation. Other issues, such as simplified screen development and better dialogs, are important to improving the ease of use of the Simulation Engine. In the next section, I will try to recommend how some of these enhancements could be implemented.

Section 6.2: Enhancements to the Simulation Engine

Two key items will improve the Simulation Engine, the move to a TCP/IP based protocol and a change in implementation language. These items will allow an improvement not only to the development of simulations but also to the general performance of the Simulation Engine itself.

The first to consider is the inter-process protocol. The biggest advantage of using the DDE protocol is that it integrates nicely with most Windows based protocols. In order to maintain this functionality, it would be desirable to maintain some level of DDE (or the more modern Windows protocol OLE) protocol support in the Simulation State. This would continue the ability to communicate information to programs such as Microsoft Excel.

The primary protocol would become TCP (Transmission Control Protocol) which establishes *streams* between different virtual sockets which would allow information to pass back and forth. The messaging system between the different Simulation Engine processes would remain intact; the fundamental changes would be made to the *ClientServer* class discussed in section 4.5. Little or no changes would be required to the other classes in the Simulation Engine.

The key advantage would then be the ability of the Simulation Engine to receive and send data to any program that can connect to it. This connection would be established in a fashion similar to the existing DDE, instead of a conversation being established a connection would be made. The advantage is that the TCP protocol can be used to communicate over a network to any platform that supports it. This means that systems running VMS, UNIX, OS/400 type operating systems on mid-range computers can communicate information to the Simulation Engine.

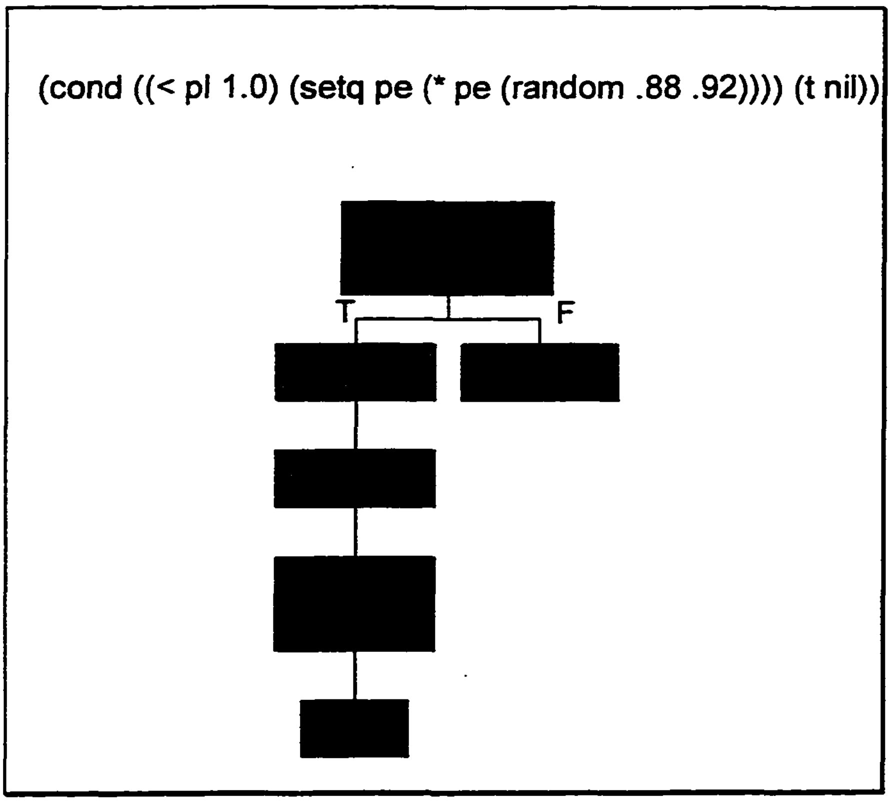
As an example, consider an industry using Sun midrange computers for control and AS400 computers for business applications. In this instance the Simulation Engine could gather real cost information from the business computer and actual field values from the control systems. This information could then be used to improve the simulation's realistic value. And since the DDE protocol is maintained in the Simulation State, information could be collected and reported on directly from an Excel spreadsheet. There are some very clear advantages to using a TCP-like protocol.

The other desirable enhancement to the Simulation Engine is an easier language for designing simulations. As discussed in section 6.1.2, the Lisp-like script language is not simple enough for development by operators. The key seems to be a movement to a graphical type language interface.

The problem with graphical, flowchart like languages is the amount of drawing a program takes up. Consider something as simple as :

```
(cond ((< pl 1.0) (setq pe (* pe (random 88 92)))) (t nil))
```

This example was taken from the simple pump simulation in section 5.1 and represents a straightforward instruction in a set of a about ten instructions. To graphically represent this, one might draw the flow chart shown in figure 6.1.



```
(cond ((< pl 1.0) (setq pe (* pe (random .88 .92)))) (t nil))
```

Figure 6.1 : Example of a Graphical Instruction

This approach is easier to learn and design in. It would require a graphical type interface to it, so that each instruction could be selected and edited. A program listing would be a listing of English instructions. The simulation designer would select the instruction to edit, which would then give a graphic flow chart similar to Figure 6.1 above. The above example was taken from the "pump_eff.scr" script program:

```
;
;     pump_eff.scr
;
;     script calculates the efficiency of pump_01
;
;
;
(ddesetq 0 pe 'pump_01_eff)
(ddesetq 0 pl 'pump_01_purged)
(cond ((< pl 1.0) (setq pe (* pe (/ (random 88 92) 100)))) (t nil))
(ddesendq 0 pe     'pump_01_eff)
(ddesendq 0 0.0    'pump_01_purged)
```

In the new graphic format, the overview listing would look like figure 6.2.

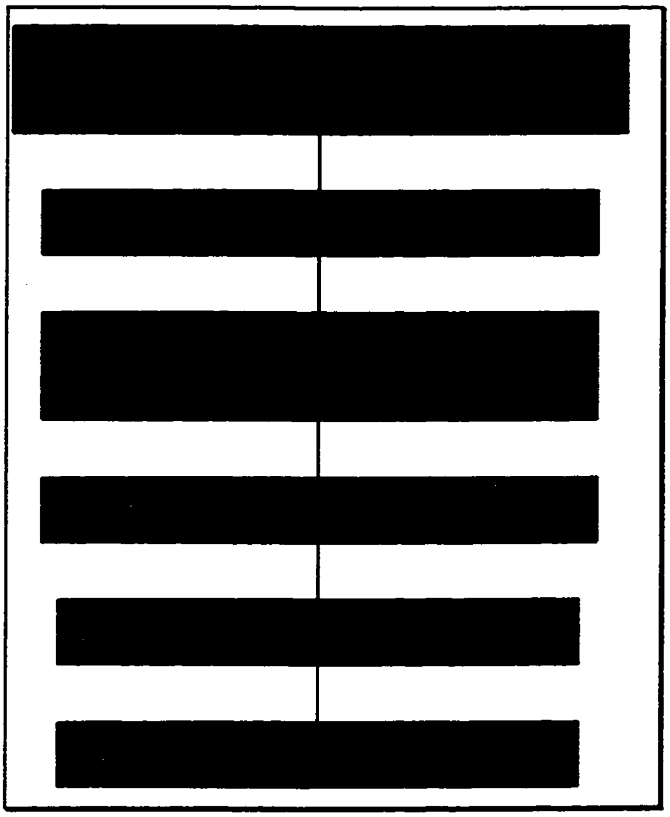


Figure 6.2 : A Graphic Script Overview

The programmer could then select the plain text description and go to a flow chart such as the one in Figure 6.1. This style of graphic programming would be easy to learn. The interface would be time consuming to design, but the result would be an advanced programming development environment.

Finally, a system of automatically "learning" from the operator's responses could be developed. In such a process the Simulation Engine would record the decisions the operator makes in response to specific situations. These would be converted into script files and integrated into the simulation to create an increasingly intelligent simulation.

Section 6.3: The Future of the Simulation Engine

The Simulation Engine was designed to give the appearance of a control system, to simulate processes in order to allow operators to learn about the process. In building this Engine, I have created a very flexible design that can be extended beyond the realm of simulation.

Recall the original model given in Figure 2.1. Each of the processes in the Simulation Engine is an independent process, using a common protocol for communicating. The addition of a process that talked to the actual control systems would be very easy to do, and would allow the Simulation Engine to become a real time Man Machine Interface (MMI).

As an example of this, consider the requirement to interface with a Remote Termination Unit (RTU). These devices are commonly used to terminate field I/O signals (typically 0-14 mA) and bring these values into an interface card. This card will then have a serial (RS232) type interface which can be connected to modem, PC or other such devices.

In order for the Simulation Engine to become a true control system it would need to be able to communicate with an RTU. Since there is a serial link, a program would need to be developed that could communicate to a serial interface to the RTU. Fortunately, most RTUs not only support multiple protocols, the specifications for many of protocols is industry standard (e.g. DNP 3.0 [Distributed Network Protocol]). The task of writing a program to communicate with a RTU is achievable.

If this program were then put into the *ClientServer* window class the other programs in the Simulation Engine would be able to communicate with the RTU. The result would change the basic model to figure 6.3.

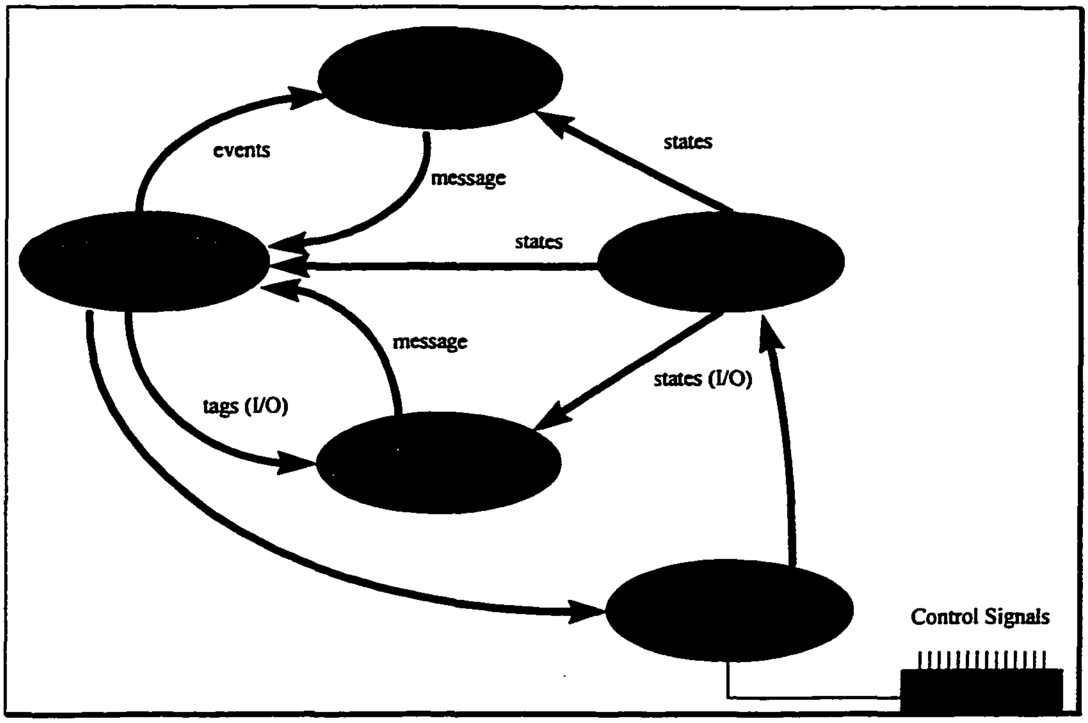


Figure 6.3 : Simulation Engine RTU Add-on Model

The RTU could then receive instructions from the Knowledge Base that could be based on higher level intelligence. The operator would have access to the outputs from the field and the ability to make changes in the field.

This evolution to a simple Windows based MMI would be a productive growth of the Simulation Engine. Even if control was not desired, most RTU-like devices have multiple communications ports - one could be used simply to gather data to create more refined solutions. Finally, the fact that the whole process can be done on an inexpensive, PC based platform - capable of communication to software packages such as Excel - makes this extension very desirable.

Section 6.4 : Final Summary

This thesis has examined an approach to developing Knowledge Based simulations in the microcomputer environment.

Simulations developed under the Simulation Engine are easy to use. The graphical type interface allows the operator to manipulate the process being simulated using simple mouse button selections. The data from the simulation can be effectively gathered using commonly used applications (e.g. Microsoft Excel). The actual development of simulations requires design skill in a high level language, but is far easier than working from first principles. The result is a simulation design system which is straightforward to use and configure.

The Simulation Engine has limited portability within the popular Windows operating systems including Windows 3.1, Windows for Workgroups and Windows 95. This effectively restricts the Simulation Engine to operating in a PC based environment, although the simulation data can be moved to other platforms by taking full advantage of the client/server nature of the Simulation State

The interactive nature of the Simulation Engine allows for rapid prototype development. Since the script files used to build the simulation can be modified at run time the simulation designer and the end user can work together on a live system to adjust the simulation. The resulting rapid feedback reduces the time between considering the change and making the change.

The Simulation Engine, with its rapid prototype features, client/sever architecture and ease of use is still a growing product. By introducing an easier to use script language the gap between simulation designer and end user will be eliminated. Additional programs can be added to the system to interface with real world equipment allowing the Simulation Engine to gather and use live industrial data.

The Simulation Engine is a valuable tool for teaching industrial processes. Not only can it mathematically model an industrial process, the Simulation Engine can graphically represent the process to simplify the learning process. The Knowledge Base nature of the Simulation Engine permits modeling of unusual or difficult to understand occurrences within the process. Properly designed, simulation developed with the Simulation Engine could be used to build predictive models of complex industrial processes. These could be used to make production choices and expert decisions.

The Simulation Engine is cost effective due to its platform being an IBM compatible running Windows 3.1, and the development time to create and support simulations. With the simplification of the script language it will be possible to use the end user to build the simulations exactly how they should be. This eliminates the need for a computer oriented person.

The Simulation Engine is the result of a carefully planned object-oriented approach. Before the platform or operating system were chosen the objects required to make this engine work were identified and defined. The message passing system was designed for the object-oriented model rather than the environment. Once the design stage was complete, it was determined that it was possible to implement the Simulation Engine in a cost-effective platform (PC) using a multitasking operating system (Windows 3.1). The model was modified to take full advantage of its native environment, but remained consistent to the original object-oriented model.

The result was an object-oriented, easy to use, cost effective solution to the problem of designing simulations for industrial processes.

Appendix I: INSTALLING THE SIMULATION ENGINE

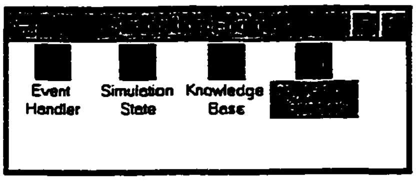
Installing and setting up the simulation engine is a relatively simple process. The Simulation Engine executables have been compressed to approximately 734 kilobytes, but will expand to just over seven megabytes once installed. This archive has been put on an installation diskette. This disk should first be put in the installation diskette drive A: (or any other drive by simply substituting the drive letter in the following). Create a directory on your hard drive (for example: *C:\> MKDIR C:\SIMENG*). Make this your current directory and execute the following command: *C:\SIMENG> A:\PKUNZIP -d A:\SIMENG.ZIP*. This will create the following sub-directories under the current directory:

AVENOR	- The simulation built for the Avenor Kraft Pulp Mill from section 5.1.3
BMPS	- Directory where simulation graphics are stored
KAMYR	- The simulation built on the Kamyr equations from section 5.1.2
SCRIPT	- The directory where the active scripts are stored
TANKSIM	- The simple pump/tank simulation from section 5.1.1

The current directory will contain the following executables:

EVENT.EXE	- The Event Handler
IOHAND.EXE	- The Input/Output Handler
KNOW.EXE	- The Knowledge Base
STATE.EXE	- The Simulation State

For installations using Windows 95 a batch file, GO.BAT is included. This batch file will execute all four of the programs in the Simulation Engine. Windows 3.1 users will need to create icons and execute the programs independently. A typical group would look like:



Once the installation is complete, you can proceed to create or install a simulation. Create the graphics for the simulation and copy them into the BMPS directory. Create the script files and copy them into the SCRIPT directory. I recommend putting an extra copy in a separate directory as was done for AVENOR, KAMYR and TANKSIM. For help on building simulations refer to chapter 5.

If a simulation is already installed, delete the installed simulation files by using *DEL BMPS*.BMP* and *DEL SCRIPT*.SCR* before installing the new files. Be very careful not to delete any other files in these directories as they may be required by the Simulation Engine. (At this time there is only one file, *PREBUILT.KEY*, which contains some pre-built scripts, that must not be deleted.)

In order to execute your simulation, execute each of the programs that make up the engine (Windows 95 users can use the GO.BAT file provided). Once the programs are running, go to each window and select the *CONNECT* menu option. (The Simulation State does not currently have a *CONNECT* option.) Maximize the Input/Output Handler and select the *STARTSIMULATION* menu option. The Simulation Engine should proceed to execute your INIT.SCR file and execute the simulation.

At this stage in its life, the Simulation Engine does not check for file existence. As well, it only does simple language syntax checking in the script files. Future versions will clean up the user interface, providing more room for user error. Until then the use of the Simulation Engine must be done carefully. The system should not cause any harm, but should be considered beta and as-is at this time.

Appendix II: THE SCRIPT LANGUAGE

The Knowledge Base uses a script language to implement database rules. This language is based on simple Lisp, as a result it contains many of the same type of commands. Despite this, there are many extensions and omissions, allowing the script language to focus on simulation development. All the script commands and syntax are listed below, using the greater-than and less-than symbols (<>) to indicate a user entered value.

-

This is the subtraction operator. The second argument is subtracted from the first.

Syntax:

```
( - <number> <number> )
```

!=

This is the 'not equal to' operator. It returns 't' if the two numbers are not equal.

Syntax:

```
( != <number> <number>)
```

*

Multiplication operator. Returns the multiplied result of two numbers.

Syntax:

```
( * <number> <number> )
```

/

Division operator. The second argument is divided from the first.

Syntax:

```
( / <number> <number>)
```

^

Exponential operator. The first argument is raised to the power of the second.

Syntax:

```
(^ <number> <number>)
```

+

Addition operator. The numbers are added together.

Syntax:

```
(+ <number> <number>)
```

<

Less-than operator. Returns 't' if the first number is less than the second, else it returns 'nil'.

Syntax:

```
( < <number> <number> )
```

<=

Less-than-or-equal-to operator. Returns 't' if the first number is less than or equal to the second, else it returns 'nil'.

Syntax:

```
( <= <number> <number>)
```

==

Equal-to operator. Returns 't' is the arguments are equal, else it returns 'nil'.

Syntax:

```
( == <number> <number>)
```

>

Greater-than operator. Returns 't' if the first argument is greater than the second, else it returns 'nil'.

Syntax:

```
( > <number> <number>)
```

>=

Greater-than-equal-to operator. Returns 't' if the first argument is greater than or equal to the second, else it returns 'nil'.

Syntax:

```
(>= <number> <number>)
```

abs

Absolute function. Returns the absolute value of a number.

Syntax:

```
( abs <number>)
```

add1

Increments a number by one.

Syntax:

```
(add1 <number>)
```

append

Appends a list or atom to a list. Fails to work if first argument is not a list.

Syntax:

```
(append <list> <list or atom>)
```

apply

Applies a function to a list of argument.

Syntax:

(apply <function_name> <list>)

atom

Returns 't' if its argument is an atom, else it returns 'nil'.

Syntax:

(atom <list or atom>)

caar

Return the CAR of the CAR of argument. (see **car** for definition of CAR).

Syntax:

(caar <list or atom>)

cadr

Returns the CDR of the CAR of the argument. (see car,cdr for definitions of CAR,CDR resp.).

Syntax:

(cadr <list or atom>)

car

Returns the CAR of a list. The CAR of a list is the first entity in a list, which can be a list, an atom, or a nil.

Syntax:

(car <list or atom>)

cdr

Returns the CDR of a list. The CDR of a list is the list minus its first element. The result will either be a nil or a list - never an atom.

Syntax:

(cdr <list or atom>)

concat

Merges the first argument into the list of the second argument. If the second argument must be a list.

Syntax:

```
(concat <list or atom> <list>)
```

cond

Evaluates the second argument of a sub-list if the first argument is true. Many expressions may be listed but only the first true expression's adjacent script is evaluated.

Syntax:

```
(cond (  (<boolean expression> <list of valid script commands>)
         (<boolean expression> <list of valid script commands>)
         ....
      )
)
```

Example:

```
(cond (  (> RR 53.1) (setq TEMP 280) )
         (t   nil)
      )
)
```

cons

Adds the second element to the first argument's list. The first argument must be a list.

Syntax:

```
(cons <list> <list or atom>)
```

consp

Returns 't' if the argument is a list, else it returns 'nil'.

Syntax:

```
(consp <list or atom>)
```

cos

Retums the cosine of the argument. Note that the result is in radians.

Syntax:

(cos <number>)

cubed

Retums the argument multiplied by itself three times.

Syntax:

(cubed <number>)

ddesendq

Sends an atom to one of the peer programs using the DDE protocol. (see following sections for details on this command). The first argument identifies the peer being communicated with (0 = Simulation State, 1 = Event Handler, 2 = I/O Handler). The data being sent is a message to the DDE peer. This is expanded on in Appendix III.

Syntax:

(ddesendq <peer number> <atom containing data> <atom containing DDE item>)

ddesetq

Requests data from a peer program using the DDE protocol and assigns it to a local variable. The first argument identifies the peer being communicated with (see **ddesendq**).

Syntax:

(ddesetq <number> <local variable name> <atom containing DDE item>)

defun

Defines functions that can be called at a later date.

Syntax:

```
(defun <function name> <list containing local variables>
<list of valid script commands>
)
```

Example:

```
(defun cubed (x) (* x (* x x )))
```

e

The constant e.

eql

Returns 't' if two atoms are equal, else returns 'nil'. Both arguments should be atoms.

Syntax:

```
(eql <atom> <atom>)
```

equal

Returns 't' if two items (lists or atoms) are equal.

Syntax:

```
(equal <list or atom> <list or atom>)
```

eval

Evaluates a list. Must be a list and the first entry of this list should be a defined function.

Syntax:

```
(eval (list containing valid script))
```

exp

Returns e to the power of the argument.

Syntax:

(exp <number>)

first

Alias for **car**.

if

Returns second argument if the first is true, otherwise it returns the third argument.

Syntax:

(if <list or atom evaluates to 't' or 'nil'> <list or atom> <list or atom>)

isconst

Returns 't' if argument is a literal value (a number or a string), otherwise it returns 'nil'.

Syntax:

(isconst <atom>)

islambda

Returns 't' if argument is the name of a function, otherwise it returns 'nil'.

Syntax:

(islambda <atom>)

isvar

Returns 't' if argument is the name of a variable or function, otherwise it returns 'nil'.

Syntax:

(isvar <atom>)

last

Returns the last member of a list (which may be a list, atom or nil).

Syntax:

(last <list>)

length

Returns the length of a list.

Syntax:

(length <list>)

list

Puts both arguments in one list.

Syntax:

(list <list or atom> <list or atom>)

listp

Returns ‘t’ if the argument is a list, otherwise it returns ‘nil’.

Syntax:

(listp <list or atom>)

ln

Returns the natural logarithm of the argument.

Syntax:

(ln <number>)

load

Reads and executes a script file. When execution completes, returns to original file.

Syntax:

(load <filename>)

log

Returns the base 10 logarithm of the argument.

Syntax:

(log <number>)

member

Returns ‘t’ if the first argument is a member of the second argument.

Syntax:

(member <list or atom> <list>)

not

Returns ‘t’ if the argument is ‘nil’, otherwise it returns ‘nil’.

Syntax:

(not <atom>)

nth

Returns the nth member of a list.

Syntax:

(nth <number> <list>)

nthcar

Returns the nth CAR of a list.

Syntax:

(nthcar <number> <list>)

nthcdr

Returns the nth CDR of a list.

Syntax:

(nthcdr <number> <list>)

null

Returns 't' if the argument is 'nil', otherwise it returns 'nil'.

Syntax:

(null <atom or list>)

numberp

Returns 't' if the argument is a number, otherwise it returns 'nil'.

Syntax:

(numberp <atom>)

PI

The constant PI.

plusp

Returns 't' if the argument is a positive number, otherwise it returns 'nil'.

Syntax:

(plusp <atom>)

quote

Returns the value of the list or atom. This stops the interpreter from processing the contents of a list or atom. The symbol ' is used as a short form.

Syntax:

(quote <list or atom>)

Or:

'<list or atom>

random

Returns a random number from the first argument to the second. Arguments must be valid numbers.

Syntax:

(random <number> number>)

rest

Alias for CDR.

reverse

Returns the reverse of a list.

Syntax:

(reverse <list or atom>)

second

Returns the second entry in a list. This is the CDR of the CDR of a list. The argument must be a list.

Syntax:

(second <list>)

setq

Assigns the first argument the value stored in the second argument. The first argument becomes a local variable.

Syntax:

(setq <variable name> <value>)

sin

Returns the sine of the argument. The result is given in radians.

Syntax:

(sin <number>)

sqrt

Returns the square root of the argument.

Syntax:

(sqrt <number>)

strcat

Appends the value of the second argument to the first argument. The result is retumed in the first argument, which must be a variable.

Syntax:

(strcat s1 "_tank01")

sub1

Returns the argument less one.

Syntax:

(sub1 <number>)

tan

Returns the tangent of the argument. The result is given in radians.

Syntax:

(tan <number>)

zerop

Returns 't' if the argument is zero, otherwise it returns 'nil'.

Syntax:

(zerop <number>)

Appendix III: TAG MESSAGES

The Knowledge Base has several messages it can communicate to the I/O Handler using the *I/O tags* discussed in section 4.2. Since these messages use the DDE protocol they are sent using the *ddesendq* command (covered in Appendix II). The following is a summary of all the messages the I/O Handler will currently accept:

changegraphic

Modify the image file stored by a *smartgraphic* tag.
Syntax:

```
(ddesendq 2 '<tagname>,<image_file> 'changegraphic)
```

flushtags

Delete all existing tags.
Syntax:

```
(ddesendq 2 nil 'flushtags)
```

niltag

Delete a specific tag.
Syntax:

```
(ddesendq 2 '<tagname> 'niltag)
```

refresh

Issue a request to the State Handler for fresh data. It can either request all tags or specific tags. The tag must be a *smarttext* tag or a *smartpopuptext* tag, otherwise the refresh is ignored.
Syntax:

```
(ddesendq 2 'refresh 'refresh)
(ddesendq 2 'refresh '<tagname>)
```

simplegraphic

Create a *simplegraphic* tag.
Syntax:

```
(ddesendq 2 '<x>,<y>,<dx>,<dy>,<image_file>,<tag_name>
'simplegraphic)
```

simplepopuptext

Create a *simplepopuptext* tag.
Syntax:

```
(ddesendq 2
'<x>,<y>,<dx>,<dy>,<text_data>,<tag_name>,<target_tag or
nil>,<script_file or nil>
'simplepopuptext
)
```

simpleregion

Create a *simpleregion* tag.
Syntax:

```
(ddesendq 2 '<x>,<y>,<dx>,<dy>,<tag_name> 'simpleregion)
```

simpletext

Create a *simpletext* tag.
Syntax:

```
(ddesendq 2 '<x>,<y>,<dx>,<dy>,<text_data>,<tag_name> 'simpletext)
```

smartgraphic

Create a *smartgraphic* tag.
Syntax:

```
(ddesendq 2 '<x>,<y>,<dx>,<dy>,<image_file>,<tag_name>,<script_file or
nil>
'smartgraphic
)
```

smartpopuptext

Create a *smartpopuptext* tag.
Syntax:

```
(ddesendq 2
        '<x>,<y>,<dx>,<dy>,<text_data>,<tag_name>,<target_tag or
        nil>,<script_file or nil>
        ‘smartpopuptext
)
```

smarttext

Create a *smarttext* tag.
Syntax:

```
( ddesendq 2 '<x>,<y>,<dx>,<dy>,<text_data>,<tag_name>,<script_file or
nil>
 'smarttext
)
```

References

Allison, B.J. & Dumont G.A. & Novak, L.H & Cheetham W.J. (1989). Adaptive-Predictive Control Of Kamyr Digester Chip Level Using Strain Gauge Level Measurements. In Pulp and Paper Reports PPR 739. Point Claire, P.Q.: Pulp and Paper Research Institute of Canada.

Ammeraal, L. (1993). Windows Wisdom for C and C++ Programmers. Rexdale, ON: John Wiley & Sons Ltd.

Borland C++. (1993a). DDEML Example. In Borland C++ 4.0 for DOS, Windows and Windows NT [Computer Software]. Scotts Valley, CA: Borland International Inc.

Borland C++. (1993b). Library Reference (Version 4.0). Scotts Valley, CA: Borland International Inc.

Borland C++. (1993c). Programmer's Guide (Version 4.0). Scotts Valley, CA: Borland International Inc.

Budd, T. (1991). An Introduction to Object-Oriented Programming. Don Mills, ON: Addison-Wesley Publishing Company.

Clark, J. D.(1992). Windows Programming Guide to OLE/DDE. Carmel, IN: Sams.

Hu J. & Rozenblit J. W. (1991). KAR for Design Model Development. In P.A. Fishwick & R.B. Modjeski (Ed.), Knowledge-Based Simulation: Methodology and Application (pp. 77-94). New York, NY: Springer-Verlag.

Kamyr, (1993). Digester Update (7th ed.). Ridge Falls, NY: Kamyr Inc.

Michaelsen R. & Christensen T. & Lunde G.G. & Lundman G. & Johansson K. (1992) Model Predictive Control Of A Continuos Kamyr Digester At SCA-Nordliner. Sweden. Paper presented at the conference Control Systems '92 Dream vs Reality: Modern Process Control in the Pulp and Paper Industry.

Nielson, N.R. (1991). Applications of AI Techniques to Simulation. In P.A. Fishwick & R.B. Modjeski (Ed.), Knowledge-Based Simulation: Methodology and Application (pp. 1-19). New York, NY: Springer-Verlag.

Oren, T.I. (1991). Dynamic Templates and Semantic Rules for Simulation Advisors and Certifiers. In P.A. Fishwick & R.B. Modjeski (Ed.), Knowledge-Based Simulation: Methodology and Application (pp. 53-76). New York, NY: Springer-Verlag.

Pratt, T. W. (1984). Programming Languages (2nd ed.). Eaglewood Cliffs, NJ: Prentice-Hall Inc.

Rothenberg, J. (1991). Knowledge-Based Simulation at the RAND Corporation. In P.A. Fishwick & R.B. Modjeski (Ed.), Knowledge-Based Simulation: Methodology and Application (pp. 133-161). New York, NY: Springer-Verlag.

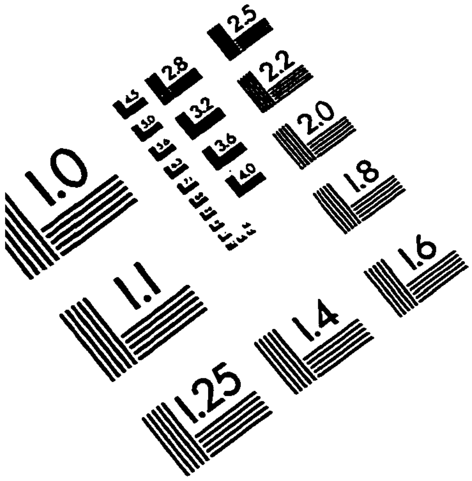
Smook, G.A. (1994). Handbook for Pulp and Paper Technologists (2nd ed.). Vancouver, BC: Angus Wilde Publications Inc.

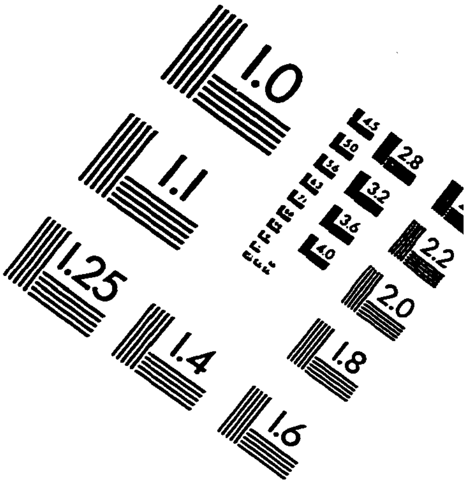
Stroustrup, B. (1991). The C++ Programming Language (2nd ed.). Don Mills ON: Addison-Wesely Publishing Company.

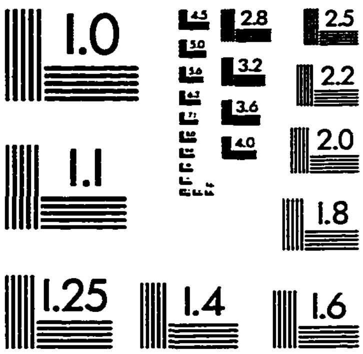
Weymouth, P. J. & Sztrimbely, M.W. (1990). Applications of Advisory Intelligence Capabilities To Process Plant Simulation & Scheduling. In B. Svrcek & J. McRae (Ed.), The Proceedings to the 1990 Summer Computer Simulation Conference (pp. 335 - 339). San Diego, CA: The Society for Computer Simulation.

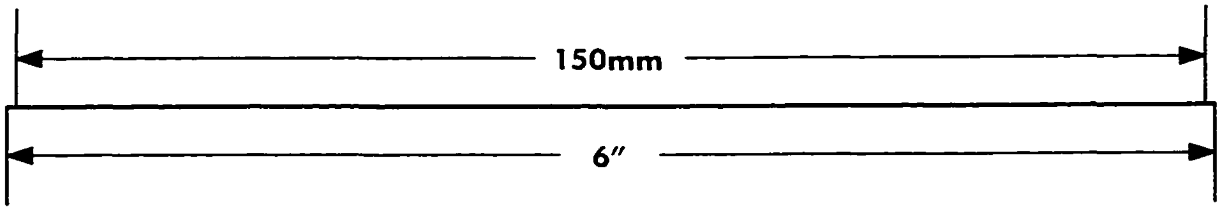
Yao, P. (1994). Borland C++ 4.0 Programming for Windows. Toronto, ON: Random House Electronic Publishing Inc.

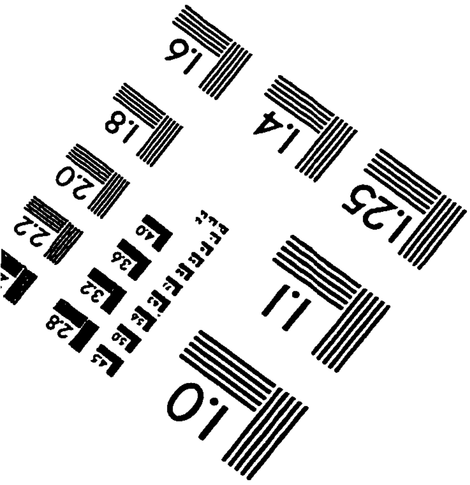
IMAGE EVALUATION TEST TARGET (QA-3)

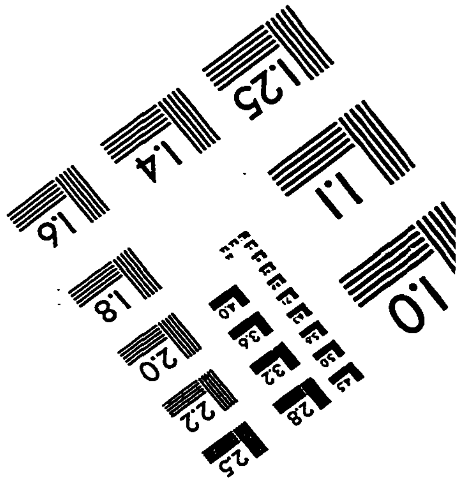












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989