A FRAMEWORK FOR A REAL-TIME KNOWLEDGE BASED SYSTEM

Ian Gebbie

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

Johannesburg, 1993

DECLARATION

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

(signature of candidate)

20 day of April 1993

ABSTRACT

A framework designed to contain and manage the use of knowledge in a real-time knowledge based system for high level control of an industrial process is presented.

A prototype of the framework is designed and implemented on a static object-orientated shell. Knowledge is stored in objects and in forward chaining rules. The knowledge has a well defined structure, making it easy to create and manage.

Rules are used to recognize conditions and propose control objectives. The framework uses the knowledge to determine variables that if altered will meet the objectives. Control actions are then found to implement changes to these variables

The use of explicit control objectives makes it possible to determine if an action worked as intended and if its use is suitable for the present conditions. This enables a learning mechanism to be applied in the expert system.

The prototype operated adequately, but the knowledge required to drive the system was found to be very detailed and awkward to create.

ACKNOWLEDGEMENTS

I would like to acknowledge the assistance of Gold Fields of South Africa for the opportunity to work on this project and for finances they provided.

In particular Mr P. Nienaber for the conception of the project and the support he provided during the project.

Thanks must be extended to Mr J. Groustch for the guidance and advice that he provided. Without his enthusiasm and support this project would not have been completed.

I would like to thank the people of Zincor, a subsidiary of Gold Fields of South Africa for their assistance in this work.

To my supervisor Prof I.M. Macleod, thank-you for your support and encouragement.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 Introduction

Traditionally control of industrial processes or plants has been carried out by human operators. The operators adjust various actuators that alter variables in the plant. Many of these actuators are part of a closed feedback loop which provides local control of a plant variable, for example, the flow rate from a tank. The overall control strategy is determined by the operator, in the form of set-points which he supplies for such feedback loops.

Although the operator cannot easily quantify the conditions in the plant, he is able to recognize changes in the process and take suitable control actions in response. The operator is able to adapt the control decisions taken to changes in the plant structure. For example, if a certain pump is being worked on, the operator can control the plant without using that pump.

Plant operators are not perfect, Hart[1] describes some of man's disadvantages as: the lack of availability; poor consistency and poor comprehensiveness.

Lack of availability describes the fact that operators are not just found, they must first be trained. During that training process mistakes are often made. These mistakes are necessary for the operator's training, but often fairly expensive.

Poor consistency describes that human operators don't always take the same control action under the same circumstances. Inconsistency arises when the shift changes, and a different operator takes over the control of the plant. The new operator has

different experiences to draw from in order to make a decision. The result is that under similar circumstances, the operators may behave differently. The same operator may also act differently, depending on mood or general awareness.

Such changes in the control strategy affect the quality of the final product, as they move the process towards or away from the optimum conditions. They also makes it very difficult to find the optimum conditions for the plant.

In a process there is often a conflict for example, between the design engineers who impose limitations on what the plant can do, and the production managers, who wish that the process takes place under specific conditions. The operator has to operate under the orders of different experts who sometimes have conflicting interests.

Ideally one would like to have one expert that takes both opinions into account. A system that is able to do this is said to be comprehensive.

Artificial inteiligence may provide a means to help limit the weaknesses of the human operators. The form of artificial intelligence that is to be used, is termed the expert system or knowledge based system. The expert system observes variables in the plant, recognizes conditions and suggests possible actions that should be taken in response to these conditions.

1.2 Expert systems

Expert systems are programs that exhibit high levels of intelligence. As to whether they are actually clever is a mater of debate, but they are able to appear to be intelligent[8].

Traditionally expert systems are used in the consultancy environment. Instead of communicating with an expert, one consults with a computer. For process control, real-time expert systems are required.

Real-time expert systems are different from consultancy or static expert systems as they are **event driven.** They must continuously watch the process that is being c ntrolled in order to determine events that should be reacted to[4]. The system must respond to the plant conditions in a timely manner. This differs from consultancy expert systems, which wait for the operator to initiate interaction.

All information about the state of the process being controlled must be gathered from instrumentation in the plant and fed into the real-time expert system.

Real-time expert systems need to be time based. The system must be able to make decisions at a speed that enables suitable response to the plant[4].

1.2.1 Advantages of expert systems

Expert systems can potentially provide the decision making required to implement an automated control system for an industrial process.

Of interest in this project, the expert system is more consistent than the human operator. Operators could miss conditions in the process being controlled. An operator's performance is affected by fatigue and the operator's personal condition. An expert system is unlikely be affected in the same way. The expert system will ensure that plant conditions are reacted to and that they are reacted to in the available time.

Additionally, to gather knowledge about the plant requires a better understanding of the process, thus the procedure of entering knowledge into the system will require process study. By obtaining a better understanding of the process to be controlled, improvements could be made to the process.

1.2.2 Limitations of expert systems

D.A Waterman[9] describes the following differences between the expert system and the human expert:

HUMAN	EXPERT SYSTEM
Creative	Uninspired
Adaptive	Needs to be told
Sensory experience	Symbolic input
Broad focus	Narrow focus
Common sense knowledge	Technical knowledge

The expert system is not creative[9]. It is not known what makes people creative, and this creativity has not been captured and placed onto a computer. This limits the performance of the expert system. A system will not be able to come up with dynamic new ways to do something, instead it can only do what it has been taught to do. For the normal control of a plant, this will not present a problem.

The human operator is able to take account of a very broad range of sensory data that the expert system cannot. The proposed expert system will be linked to a Distributed Control System (DCS) which will act as the expert system's eyes. But the expert system can only observe what the DCS is wired up to. Extensive instrumentation is

required before suitable performance can be achieved by an expert system.

The expert system's domain of operation is very narrow. This limits the adaptability of the system. The narrower the domain that the expert system is operating in, the less likely it is to be adaptive to abnormal situations. If information describing as many factors that can influence a domain is provided, the focus of the system would be a broader allowing it to be more adaptive.

1.? Background to expert systems

1.3.1 Introduction

Expert systems are systems that behave like an expert. They contain expertise and act intelligently in a narrow domain[5]. Over the years of development of artificial intelligent systems, many various schemes have been tried, but it was only in the 1970's that it was recognized that intelligence depends on the knowledge that a system contains. D.A Waterman[9] writes:

> **To make a program intelligent, provide it with lots of high-quality specific knowledge about some problem area.**

A type of expert system that is dealt with in this plant is the knowledge based system. A Knowledge based system is made up of a knowledge base that contains the domain knowledge, and an inference engine that uses the knowledge to make decisions. Through out this report, knowledge based system and expert system may be used interchangeably, but strictly speaking an expert system is a particular type of knowledge based system.

1.3.2 History of expert systems

Expert systems started to be developed in the early 1970's. The most famous of these systems are DENDRIL, MYCIN and Prospector[7]. MYCIN was a diagnostic system that was used to diagnose bacterial infections. Prospector was a system used to deduce the presence of various minerals.

1.3.3 Inference Engine

Knowledge base systems are divided into two parts, the inference engine and the knowledge base. The inference engine is some system that takes knowledge and uses it to deduce facts.

The inference engine is independent of the knowledge it contains. The inference engine will define the structure of the knowledge used in the expert system.

1.3.4 Knowledge base

The knowledge base contains the knowledge that makes the expert system intelligent.

For an expert system to control a process, knowledge is required describing:

- The process being controlled.
- The conditions in the plant the controller must watch out for.
- The control actions that can be taken to control the process.
- The behaviour of the process.

There are various ways to represent the knowledge in the knowledge base. In the

system that is proposed frames and slots, as well as rules will be used.

Frames and Slots

The knowledge is incorporated into a pre-defined framework, that contains knowledge areas called slots. The frame contains a description of the form of the information to be entered into the slots[8].

Rules

Rules are discussed in detailed in chapter 3. Basically rules are split into a condition or antecedent and a response to that condition, or consequent. Knowledge that is not stored in the frames, is stored in the rule structure[10].

1.4 Project objective

Little information has been gathered describing the implementation of real-time expert systems in the metallurgical process industry. To evaluate the effectiveness of real-time expert systems, an attempt will be made to implement a prototype system on a metallurgical process.

It is recognized that a framework should be developed on which such systems could be developed[6]. Such a framework would define the structure of the system. This dissertation describes a possible structure for such a framework.

A proposed expert system would have to deal the issues described below:

- Ease of knowledge management. An expert system is based on large volumes of information needed to determine the conditions that are to be reacted to, and the actions that should be taken in response to those conditions.

  The plant is modified occasionally. The expert system must be able to deal with any of these modifications without a major re-working of the system's knowledge base.

- Adaptability. As the plant under goes modifications, some control actions may not work as well as they did in the past. There could also be errors in the knowledge base. The system should be able to detect when knowledge fails to deduce a correct control action and revise the acceptability of the knowledge that lead to the incorrect solution.

- Consistency. The system should be as consistent as possible. The major advantage of using expert systems, is that they are consistent.

- Self reliance. The expert system is a real-time system and should not need any one to watch over it, or to prompt it.

  The expert system must always be on line. Any proposed system must not crash and must be self starting in the case of power failure.

The framework should provide:

- A defined structure for the knowledge to be entered. This makes knowledge creation and management easier. The knowledge will be grouped according to it's function.

- Management functions to control the use and implementation of the knowledge. The framework will take the knowledge and apply it as it is required.

- Conflict resolution. The framework will provide mechanisms to deal with conflicts between decisions made by the knowledge base.

- Management of decisions made by the system. The framework ensures that the decisions taken by the system are carried out, and indeed have the predicted effect.

The project describes the design and development of a framework that will attempt to meet the above-mentioned objectives. Knowledge will be added to this framework, and it will be used to control the process.

For the initial system, the plant operator will be kept in the control loop. The system will provide messages to the operator giving instructions describing a control action to be carried out. The operator is free to carry out the control action if it is felt to be suitable.

A description of the environment in which the expert system is to be tested in is given in appendix A.

1.5 Overview of dissertation

This dissertation describes a framework for an expert system that can be used to control an industrial process. The framework is designed for, built and tested on a metallurgical plant. It is created using an available expert system environment which

provides rule mechanisms and object-orientated design abilities.

Chapter 2 provides a description of the proposed framework in a brief form. Chapter 3 then describes how the knowledge and inference engine is stored in the object-orientated environment. The knowledge types used in the system are also described.

Chapter 4 describes the elements required for the system to be able to recognize conditions in the plant, and to propose changes to the plant. Chapter 5 gives a overview of the search routines and costing systems used to find a method to implement these changes. Chapter 6 discusses the storage of the control actions required to control the process. These control actions are the final result of the system.

Chapter 7 contains one of the main contributions of the research. It discusses the use of knowledge describing the interaction of the various plant variables. This knowledge is used to find a control action from the original condition.

Chapter 8 describes the process of implementing the discovered action. Chapter 9 describes the management of any decisions made by the expert system. Chapter 10 is an evaluation of the system, and chapter 11 draws final conclusions.

CHAPTER 2 PROPOSED KNOWLEDGE-BASED SYSTEM FRAMEWORK

2.1 Introduction

The objective of the expert system is to provide high level control of a process. The system will use knowledge describing the process plant to determine how to control it.

The expert system is in two parts. The inference engine or framework and the knowledge used by the framework. Together, the framework and the knowledge constitute the real-time expert system. The framework is rigid and independent of the process it is to control, but the knowledge built on the framework will alter according to the process it is being applied to.

The framework for the system will:

- Define the knowledge structures required to contain the information required to control a typical process.
- Define the interactions of the knowledge used.
- Provide management of the knowledge.
- Provide management of any decisions taken by the knowledge base.

The knowledge used by the system provides information describing:

- The equipment the plant contains.
- The variables that describe the plant.
- The interactions of these variables.

- The control actions there are available to control the plant.
- The conditions the controller must react to.

The framework requires the knowledge to be split into different groups. Each group of knowledge is used at a particular stage in the expert system, for example Control Rules are used to define the possible conditions that can occur in the plant while the Item group is used to define the equipment used in the plant. Each group of knowledge has a specific structure. The knowledge groups are described in chapter 3.

By grouping the knowledge and defining how it should be used, each piece should be easier to create and manage. Each piece of information is independent, and can be changed at any time without any need to alter much of the other knowledge. Easy management of the knowledge is important in a plant such as the one worked on, where the process is subject to modifications.

The inference engine uses the knowledge contained in the system to reason with, and from this knowledge and data retrieved from the metallurgical process, the inference engine will determine suitable control actions to be taken on the plant.

The procedure to control the plant is split into five major tasks:

- Recognition of conditions in the plant that requires a reaction.
- The definition of a response to that condition. The response is defined as a variable change in the process called an Objective.
- The search for a variable, that if changed will cause the required response.
- The search and implementation of a control action that will change the variable in question.

- Observation of the results of the action to determine if it was suitable.

The framework or inference engine not only manages the knowledge in the system, but stores and manages decisions made using that knowledge. The framework will take the knowledge groups and apply them at the appropriate times. The framework ensures that deductions made by the knowledge is correctly stored and grouped. This is necessary in order to manage the search for solutions, and the management of the resultant conclusions of the search.

The expert system was built using an object-orientated approach where independent objects are defined that have a particular behaviour. The behaviour of an object is defined by a class definition. In this system the knowledge is contained in instances of the class definitions while the classes contain the inference system or framework. Object -orientated programming is discussed in the next chapter. Knowledge that is not contained in the instances is defined in rules.

2.2 An overview of system behaviour

The control strategy is split up into a number of stages, each defined by the framework. These stages consist of:

- A read cycle where data is read in from the process.
- Condition recognition, when the data read in from the process is analyzed.
- The creation of objectives.
- The search for possible variables to alter in response to the objectives.
- The search for control actions to implement the proposed variable changes and hence meet the appropriate objectives.

- A management stage.

During the control cycle the framework reads data from the plant, and uses rules to determine any conditions that need to be reacted to. These rules define an objective that should be met in response to the determined condition. The framework then takes the objectives created by control rules and tries to find secondary objectives or actions in response to these initial objectives. The action or secondary objective, if implemented will cause the initial objective to be met.

During the search for necessary responses, inference paths are created that describe the changes the final action should cause to the process. These paths are used to define the conclusions of each relevant decision made by the system. Any number of these paths can exist at a time. These paths are then managed to ensure that the changes that they describe are implemented and that the paths are removed when they are no longer required.

Figure 2.1 shows a overview of the expert system.

2.2.1 Condition recognition

The first stage of the control procedure requires the recognition of conditions in the plant that need to be reacted to. In order to achieve this it is necessary to be able to observe the process.

Instrumentation is used to obtain information describing various variables in the process. The information is gathered by a distributed control system and transferred to the expert system running on a PC, via a serial link.

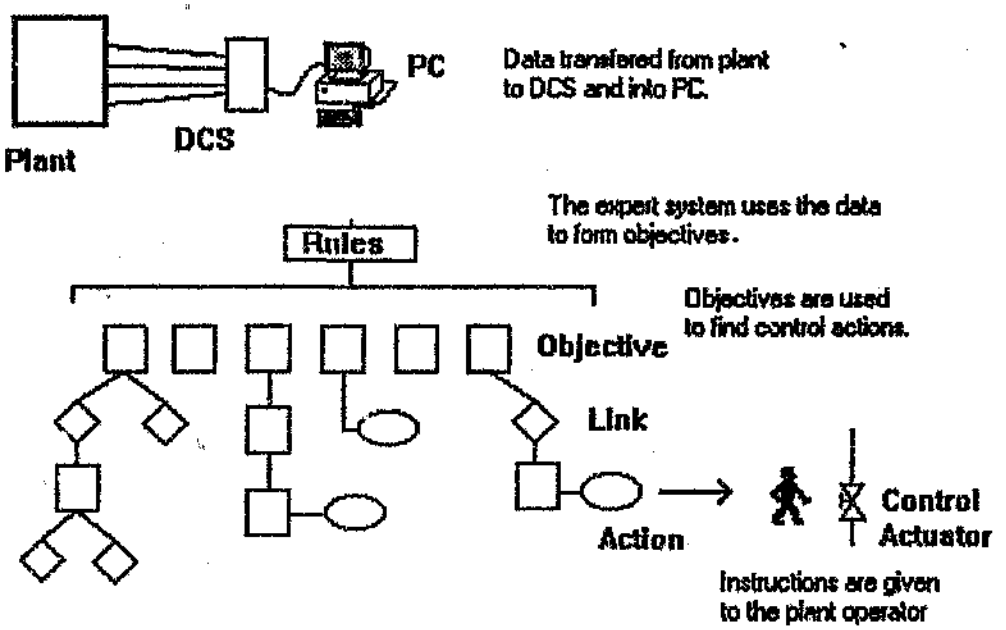
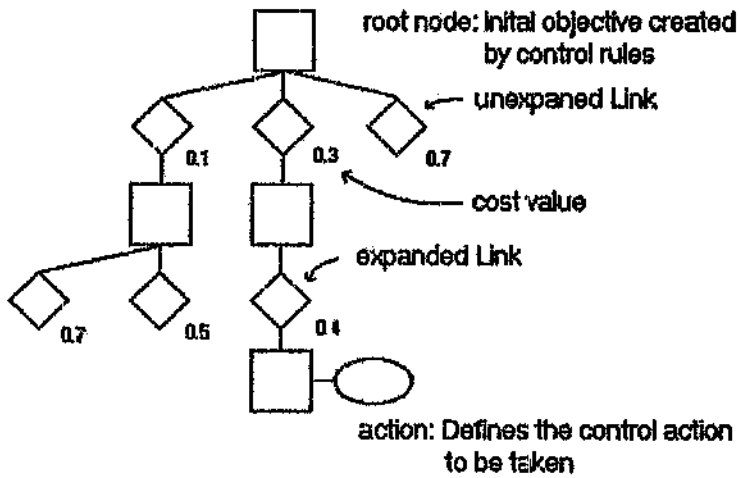


Figure 2.1 An overview of the expert system behaviour

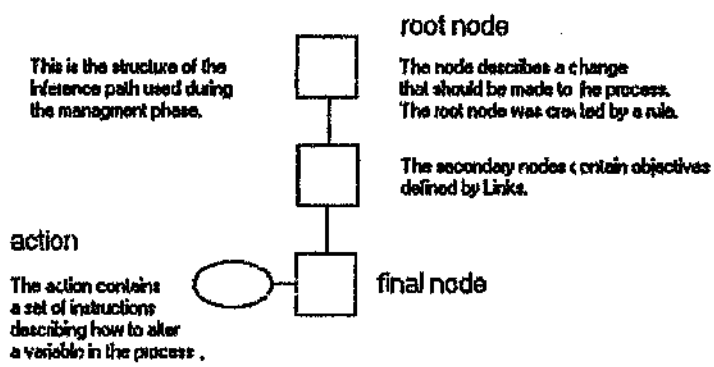
The variable values are read from the plant via the serial link and stored in the expert system. Rules are applied to the variables to check if any conditions exist in the process that must be reacted to. If the necessary variable pattern exists, and the rule is valid, the rule will define what should be done in response to that condition.

2.2.2 Defining objectives

The obvious thing to do at this stage is to define the action that should be taken in response to the condition. Instead the program requires the final objective of any control action. The objective describes what a final control action should achieve. For example if a tank is full, instead o. the objective being to shut a valve, the defined objective is to stop the tank level from increasing.



a) Typical inference path structure during a search



b) Inference path after activation

Figure 2.2 a & b Inference paths

By defining the ultimate objective of a control action:

- One is able to find the best control action for a situation.
- The knowledge required to do this contains less repetition as it is easier to generalize.
- The knowledge is more independent of the process.
- It is possible to evaluate the effectiveness of an action.
- In some cases the knowledge is easier to create.

Objectives

The objectives are variable changes that should be made in response to some condition. These objectives are created by the rules that are used to recognize conditions in the process. The objective will state:

- The variable to change.
- The amount it must be changed by.
- The time available to make the change.

The objectives are stored in a class called Nodes. The Nodes created in response to conditions in the process are called root Nodes. The root Nodes are the start of inference paths that are created during the search for control actions required to implement the necessary changes to the process.

2.2.3 Inference

Once a condition has been recognized and an objective set, it is necessary to find a variable that if changed will cause the objective to be realized, and a control action

to cause that variable to be altered. This assumes that all the objectives generated in response to conditions in the process will require some variable to change in the process.

In order to determine which final variable to change requires knowledge describing the interaction of the various variables in the process. This information is contained in rules called Link rules. Deductions from these rules are stored in a class called Links.

During the inference procedure a path is created from the initial objective created by the control rule, to some final control action. This is shown in figure 2.2 a. The

Links are used to create secondary objectives. These secondary objectives are connected together to form the inference path which connects the root objective to a control action. The information in the path is used during the management stage.

2.2.4 Control actions

The control actions define some change that needs to be carried out on the plant As stated in the introduction, the system is not designed to directly change the process being controlled. Instead a message is presented to the plant operator. The message contains an instruction that he is expected to carry out.

If the instruction is carried out, it is assumed that one variable in the process will directly change. This
variable could effect other variables. Each action is defined to directly affect this one variable, this makes the actions easier to create.

Actions are stored in a class called Actions. The action terminates an inference path.

2.2.5 General description of the inference path

The inference path describes the decisions that are made in order to determine the control action that should be taken in response to an objective. Each objective is contained in a Nodes and describes a variable that will change in the plant. The objective describes how much the variable will change, and when the change should occur.

The path starts with a root Node which contains a description of the initial objective set by rules. A search is made for actions that will cause the change defined by the objective. If a control action is found to directly affect the variable in the objective, the path will end here.

If no action is found, Links are used to determine other objectives that could be met, that would cause the initial objective to be realized. These other objectives are created when a Link is expanded. The new objectives are attached to the inference path. The new objectives are called secondary objectives.

Each secondary objective is also contained in a Node. The Node is attached to the Node containing the primary objective it effects thus building up the inference path. The inference path continues from the initial Node through the secondary Nodes down to a Node that finds an action to directly affect it.

The framework initiates a search for an Action to meet the new secondary objective. A secondary objective will either find a control action or other objectives that could solve it. The new secondary objectives will be added to the inference path. The

search for secondary objectives continues until an Action is found for one of them.

The framework can contain any number of these inference paths. It manages the creation and storage of the paths and ensures that no conflict exists between them.

2.2.6 Management

Once an action has been found for an inference path, and it's message has been presented to the operators, the system will watch to ensure that the variable changes specified by each objective in the path have indeed occurred.

In order to manage the Action found, it is necessary to store information describing the conclusions of various parts of the inference system. The information stored specifies the variables between the initial objective and the final action that should be seen to change, by how much they will change and the times the changes should be seen by.

The information required by the management stage is contained in the inference path for the particular root objective. Figure 2.2 b shows the path that resulted from an inference procedure. Note that because the rules can create any number of objectives, there can be any number of inference paths that need to be managed.

If the expected variable changes occur, the system presumes that the action was successful.

If the action did not cause the variables to change, the piece of information that is thought to have failed is marked and will not be used for a specified amount of time. By not using some piece of information, a different and possibly more suitable

response will be found to meet the objective the next time a search is carried out.

2.3 Costing system

During the inference procedure it is necessary to compare the possible variables that could be changed to meet an objective, and compare the actions that could be used to change these variables.

A number between -1 and 1 is used. This number is called a cost element as it represents how advantageous it is to make a certain decision. The decision that has detrimental consequences, or is not directly suited to the plant conditions is said to be expensive and have a cost of close to 1.

Costs are generated that take various factors that influence a decision into account and are combined using the technique used in MYCIN[5].

2.4 Learning

Because the actions are checked to ensure that they have met the objectives, it is possible to evaluate the actions. During this evaluation, the success rate of an action is derived.

A basis for comparison between Actions is provided using a costing system. Apart of the costing system takes this success rate of an action into account. This enables the system to learn that some actions are more successful than others, which will lead to consistency of the system, as the successful actions will be chosen in preference

to less successful actions.

2.5 Assumptions

Various simplifications and assumptions were made to enable the work to be tackled.

- Every action is assumed to directly change only one variable. This can be seen in a typical example; by changing a valve position one directly changes a flow rate which may in turn change a number of other variables. The action has however only changed one primary variable namely the flow rate.

  There are some circumstances, as when switching from one pump to another, that the action causes two states to directly change. Under such circumstances it is possible to define a variable that combines both states into a single value. The action will be causing a change to this one primary variable even though in reality two or more primary variables are changing.

- It is possible to link one variable to many others, thus it is possible to describe that to change a variable, for example variable A, the variables B,C,D or E could be changed. It is assumed that to meet an objective to change variable A, only one of the other variables need to be altered. Again, if a combination of variables must be altered to meet the desired objective, it is possible to create an abstract variable that describes the combination of these various variables.

- It is assumed that all the possible objectives can be met by a variable change. Because any control action is assumed to be directed at the plant this is should not be an issue. Any control action that is of any use will alter some of the plant

variables.

Problems could be experienced when the variable that is to be altered cannot be measured. The system had to be adapted to take this situation into account.

2.6 Conclusion

The proposed system is built in an object-orientated programming environment. It contains a separate knowledge and inference system. The knowledge is incorporated into instances of objects and into rules, depending on the knowledge type.

Various types of knowledge are used during different parts of the inference process. By providing defined knowledge groups that are independent of one another the system should be easier create and maintain.

The proposed system uses rules to detect conditions in the plant which propose objectives that the control actions need to meet. The rule mechanism allows generalities to be used, thus enabling one rule to cover numerous circumstances. Any number of objectives can be created by the rules.

The objective defines the variable that must be changed, the amount it is to be changed by and the time available for the change to be implemented.

A search is made for some control action that will meet the objective created by the control rule. This search is called the inference phase. By defining an objective, the solution to a specific problem is not limited. The control action that is found, could be any action that will meet the objective.

During the inference phase paths are created linking Nodes to Nodes or Nodes to Actions. The Nodes define objectives created by the control rules, or by Links. The Links define the interactions of the plant variables, and are used to determine if there are other variables that if altered, will cause the original variable to change. These variable changes are defined in the secondary objectives which are created by the Links. An inference path is created linking the root objective to these secondary objectives and the secondary objectives to subsequent objectives or to a control action.

A path will exist for each objective created by the control rules. These paths define the changes that are to be made to the process and are used during the inference phase and the management phase. The paths will end in a control action.

The control actions contain instructions that are presented to the plant operators describing a change to be made to the process. Obviously an attempt is made to use the best action found.

Control actions are grouped according to the primary variable that it alters. The action is easier to define if one considers the variable it will directly alter.

Rules are used to create the links between various variables. The rules again allow general pieces of knowledge to create specific deductions. These Links enable a connection to be made between an original objective and the actions primary variable.

A management phase will watch to ensure that the action has had the desired effect and update a learning mechanism. The learning mechanism will update the costing in the Actions and the Links which affects the use of the action in the future. Once the management phase has completed, the inference path is destroyed allowing the

objectives to be re-created if they are required.

CHAPTER 3 KNOWLEDGE REPRESENTATION

3.1 Introduction

Before explaining how conditions are detected and a how responses to the objectives created by the rules are found, it is necessary to describe how the information for the decision process is stored. This section will describe the knowledge representation used in the prototype.

The expert system was created on a object-orientated shell. This shell provided a platform for the rule base, and mechanisms for managing the rules. It also provided mechanisms for object creation and management.

3.2 Object-Orientated Programming

3.2.1 Introduction

Object-orientated programming is a programming philosophy in which software groups are created. These groups are given particular characteristics and are called objects. The characteristics of each object can be used to describe physical or abstract entities.

The object structure is in two parts. The first part is a class structure that defines the characteristics of various entities. The class structure does not contain any knowledge, but defines the information required to describe some entity. It also defines the reaction of the object to that information. The second part is made up of

instances.

Instances are data structures that result from the class description. Instances take on the characteristics of the class, the class defines the structure of the instances. The instances contain unique information describing a particular entity.

3.2.2 Classes and Instances

A general class is created that describes the standard characteristics of a particular object. This is similar to a type definition. Instances of the class are created to define actual items that are to take on the behaviour of the class, such as a feed slurry tank which is a type of tank i.e. the class tanks defines the structure of the specific object, in this case the feed slurry tank. A class can have many instances, each instance has the same behaviour but describes some different element of the same type.

For example if trying to describe birds, the class birds could define variables such as colour and wingspan. The instance of the class birds could describe a particular bird called an eagle and state a 2.5 m wingspan and a brown colour. The class defined the information required. The instance provided the information about a particular bird.

3.2.3 Slots

The information describing the object is contained in variables contained in the object called slots. In the example about birds, the slots would be the wingspan and colour.

If good object-orientated programming is used, slots can only be accessed by the

object itself i.e the slots cannot be directly altered by some external entity. For a slot to be altered a Method in the object should be called to make the change.

3.2.4 Methods

The functions that define how the objects behave are called Methods. Strictly speaking any communication to the object should be through its Methods. This forces a fixed interface to the object, should modifications to the object be required it should not be necessary to alter much of the external code using that object, as the Method will be the only section of code that will require modification. This assumes that information transfer to and from the Method will stay the same.

3.2.5 Inheritance

The properties of classes can be inherited. A class of storage tanks can be created that is decedent from the class tanks. The class storage_tanks which is a decedent of the class of tanks will have all the featuros of the class tanks. If the class tanks contained a variable called Level, so would the class storage_tanks. The class storage_tanks could contain some additional information such as a variable SG for example.

The "family tree" describing the relation between objects can be used to store information describing some instance, and is used in the expert system to enable general rules to be used. It is possible to say that for all tanks there is a variable called level, and that for all storage tanks the level should never be over 80%. The fact that a particular tank is a storage tank instead of a leach tank can be ascertained by looking at the family tree of the tank.

Figure 3.1 shows a typical hierarchical structure used in the expert system.

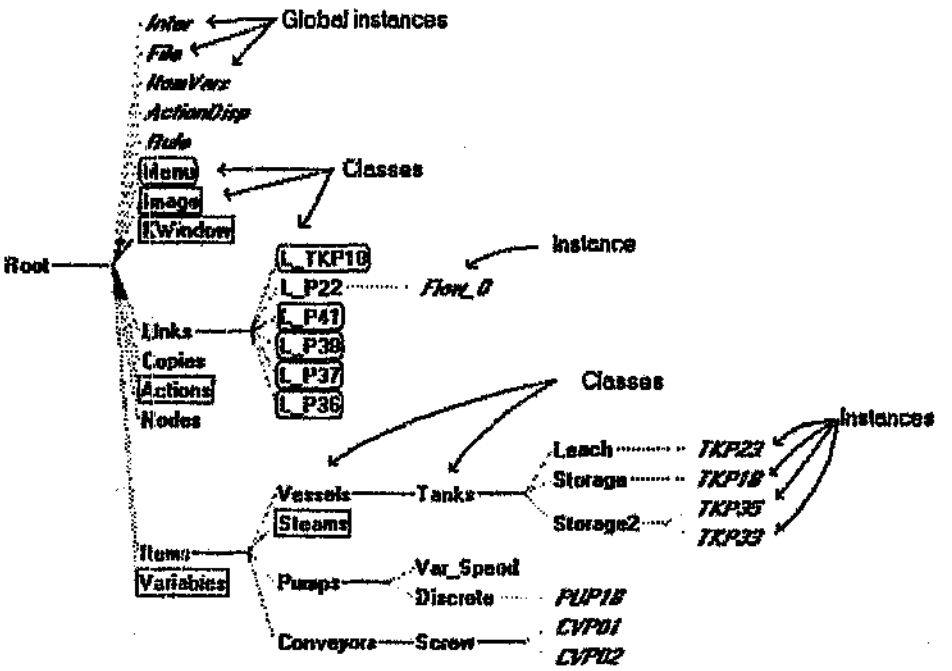


Figure 3.1 Hierarchical structure of objects in the expert system

3.3 Classes in the expert system

3.3.1 Introduction

Various classes define the different types of data storage that is required in the expert system. As in all expert systems, there is a clear distinction between the systems inference engine, and its knowledge base.

The inference engine for the system is incorporated in to the class definitions of the system, but the knowledge appears as instances of the classes. Different classes are created to contain and manipulate various parts of the knowledge and inference process.

The system is made of the following classes: Nodes, Links, Items, Actions, Variables as well as some auxiliary type classes for the user interfaces, and general system management. A brief description the major classes will be given below.

3.3.2 Plant information: Items

The instances and decedents of the Item class of knowledge is user defined and is the first part of the knowledge base created. It contains a description of all the physical objects in the plant, although obviously abstract objects can also be used.

The class will not be discussed again, but it is critical in the system as it defines the plant that is to be controlled and it defines the variables that describe the process. These variables are used to define the objectives which result from control rules, and the control actions used to control the plant. The interactions of these variables are described in the Links.

The class is used extensively by the control rules as it stores the variable values that describe the plant. When ever a variable is referred to, it is necessary to define the Item that the variable belongs to and the variable name e.g. TKP23:Level refers to the level in the leach tank TKP23.

This part of the knowledge base can be created directly from information on a process and instrumentation diagram. The user specifies what each Item is, by using

the class structure as shown in figure 3.1. The Items characteristics are defined by the variables that describe that it. The user also specifies the other Items that are connected at the input side and output side of a particular Item.

This raw information has it's most use when general rules are used to describe some knowledge. The more information that is provided at this level, the more general the rules can be.

The Item class contains the methods that read information into the variables from the distributed control system. The actual variable data is not contained in the Item class but access to these variables is via the class.

3.3.3 Variables

This class is the object used to store information describing each variable of an Item. Typical variables are pH, level, flow, temp and SG.

Some of the information required to describe a variable includes:

- The variable value.
- The tag name of the variable.
- Dimensions.
- Short text.
- Long text.

For binary variables:

- Text for logic 1.
- Text for logic zero.

When a variable is created for an Item, it is linked to the Item and created as a class of Variables. The user never sees this class and it is used as if it were apart of the Item class.

3.3.4 Storing conditions: Nodes

This class stores the objectives generated by the rules. It knows how to search for a solution to the objective it contains, and also contains the management functions required to manage itself in case of success, failure and inability to find a solution, or when it needs to find another objective to meet itself. The Node is able to decide what to do if it finds another Node that contains a similar or opposite objective to itself. A majority of the code for the system appears in the Methods for this class.

Because the number of objectives required depends on the rules which have been activated, which in turn depends on the plant conditions, the instances to the Nodes class are dynamic. They are created when required by a rule, and destroy themselves when their task is finished.

The creation of objectives and hence the creation of Nodes is discussed in chapter 4.

3.3.5 Storing final solutions: Actions

The Action class contains knowledge describing how to change variable values in the plant. The Actions class defines a control action that the operator should carry out. It contains:

- A message describing what the operator should do.
- The variable that will be directly changed if the operator carries out the

[illegible] instruction [illegible] in it [illegible]

[illegible] The amount the variable will be altered by.

[illegible] The [illegible] between the operator carrying out action, and the variable [illegible]

3.3.4 Storing [illegible]

[illegible] to alter one primary variable. The possibility that the [illegible] could alter many other variables is ignored. This makes the solution [illegible] much easier.

required [illegible]

or which [illegible] are created directly by the user and are associated according to Item the what to do if [illegible]

itself [illegible]

The Action contains information describing how it should be implemented. There are two possibilities, the Action can either alter some internal variable in the expert system, or it can give a message that should result in some external variable change in the plant.

[illegible]

In order to meet some objective it is necessary to find the resultant control action. The search procedures defined in chapter 5,7 describe how to find an Action that will cause an objective to be met. Actions are discussed in detail in chapter 6.

3.3.6 Linking of variables: Links

During the inference procedure it is necessary to find a variable that if altered will result in the original objective to be met.

From condition rules, an objective is created. This objective will define a variable to be altered. If no control action can be found to directly change this variable, a

instruction defined in it.

- The amount the variable will be altered by.
- The time delay between the operator carrying out action, and the variable changing.

Each Action is assumed to alter one primary variable. The possibility that the primary variable could alter many other variables is ignored. This makes the definition of Actions much easier.

The Actions are created directly by the user and are associated according to Item the Action effects.

The Action contains information describing how it should be implemented. There are two possibilities, the Action can either alter some internal variable in the expert system, or it can give a message that shou'd result in some external variable change in the plant.

In order to meet some objective it is necessary to find the resultant control action. The search procedures defined in chapter 5,7 describe how to find an Action that will cause an objective to be met. Actions are discussed in detail in chapter 6.

3.3.6 Linking of variables: Links

During the inference procedure it is necessary to find a variable that if altered will result in the original objective to be met.

From condition rules, an objective is created. This objective will define a variable to be altered. If no control action can be found to directly change this variable, a

secondary objective must be created defining another variable that if altered, will cause the objective to be met. It is possible that there is a control action that could alter this other variable.

A typical example would be trying to meet an original objective to alter the level of a tank. A secondary objective that would meet the primary objective, would be to alter the flow rate from the tank.

Note that some confusion can arise from the use of primary and secondary here. The primary variable is the variable that causes the secondary variable to change. In the case of a tank level, the flow rate out of the tank will be the primary variable, as changing this variable will cause the rate of change of the level, the secondary variable, to be altered. Figure 3.2 should clear up this matter.

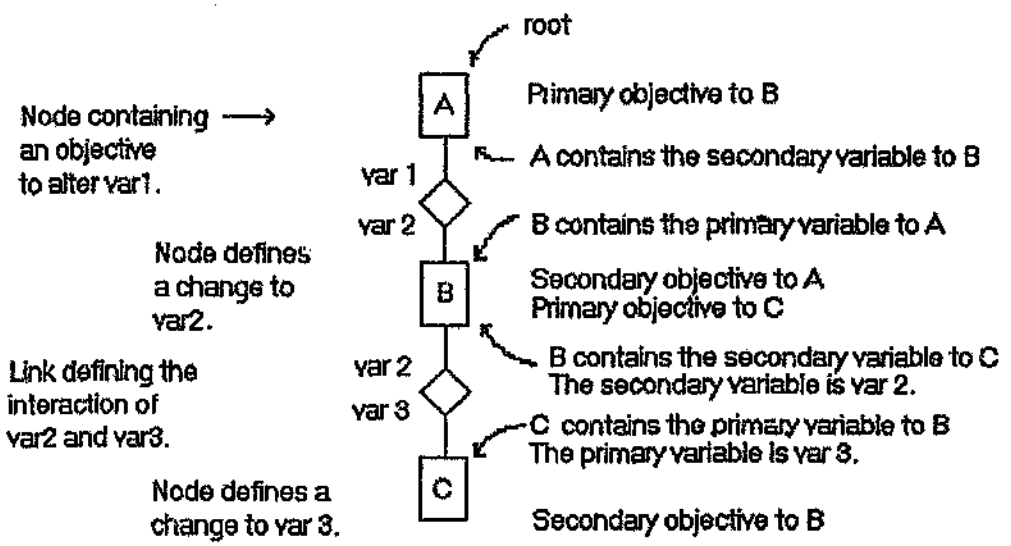


Figure 3.2 Diagram indicating primary and secondary nodes

However the secondary objective is to get the primary variable to be altered. The primary objective would be to alter the rate of change of the level. This can be met by meeting a secondary objective, namely to alter the flow rate out of the tank.

The class Links contains knowledge describing the interaction of the two variables it is linking together. It contains functions describing how to create itself, and now to manage itself. Rules called Link rules are used to create this class as they are required. They are never destroyed unless by the knowledge engineer.

The knowledge contained in the Links is deduced from the rules, but once it is created by a rule it is kept so that it can be used later. This makes it necessary for very careful knowledge management. If any of the rules that create the Links are changed the Links already created by that rule are not destroyed. The old Link may still exist but an updated Link from the new rule would also be created.

The relevance of the Links is altered as they are used, by altering a cost estimate of each Link, and by considering the percentage of times a successful action was found when the Link was used.

Because rules are used to create the Links it is possible use general rules that will create links between specific variables. To do this, information about the variables must be stored in a class called Items which has been discussed.

3.3.7 Conclusion

Objects are used to contain the knowledge base and inference engine of the expert system. Some of the objects are created by the user, such as the Items class, and Actions class. Some of the instances of classes are created by rules, this includes instances to the Links class and the Nodes class.

3.4 Rules

3.4.1 Introduction

Much of the system's knowledge is contained in IF THEN rules. The rules are used to determine if a specific condition has occurred, and what to do if it does. They are also used to direct the inference process.

IF THEN rules contain an **Antecedent** and a **Consequent**[5] and are of the form:

IF the Antecedent is TRUE THEN carry out the Consequent

A typical example is:

IF the LEVEL of Tank TKP01 is > 80%

THEN reduce the LEVEL of Tank TKP01 by 10%

Note that the Consequent part of the rule contains a reaction to the condition in the Antecedent. The Antecedent must evaluate to TRUE or FALSE. In this expert system, if the information in the antecedent is not available it will return FALSE.

There are various types of rules that differ in structure depending on their function.

Rule Sets

The rules are divided into rule sets according to their function. Each set will have a specific function and will be used at specific times in the inference process.

By defining the structure of these rules it will be attempted to constrain the type of knowledge being contained in them. This will make the knowledge easier to create and manage.

The sets are:

- **Control rules** which recognize conditions and create objectives in response to these conditions.
- **Variable rules** which are used to deduce further variable values using information from the plant, for example if a pump is off it can be deduced that a particular flow rate is zero.
- **Link rules** are rules that define the relationship between various variables and create Links.
- **Link Cost rules** are rules that are used to manipulate the desirability of using a particular link.
- **Action Cost rules** are used to alter the desirability of taking a particular action to meet an objective.
- **Default Rules** define the default action that should be taken if no solution can be found for an objective.

3.4.3 Control rules

This rule set is activated second in the cycle of the expert system. The rules in the set describe conditions that need to be reacted to.

The rules also specify what should be done if the condition occurs. The control rules specify the objectives that should be carried out when specific conditions are recognized.

To create an objective the rule requests the Item that contains the variable to be altered, to create a Node. The rule states:

- The variable name.
- The time in seconds available to alter the variable.
- The amount the variable is to be altered by.
- The priority of the request.
- A message to be used for explanation purposes.
- A name and number describing the rule.
- The Node is told if it should wait for a variable change, or wait a specified period of time before ending. The time period is stipulated as well.
- A term is passed telling the Node it is created by a rule.

Example

IF

IsAKindOf?(Item,Storage) And Item:Level < 20

THEN

Obj(Item, Level, 4, 20 - Item:Level, 600,1000, FALSE, "The tank "#Item#" is under 20% full",C_Levellw, 1);

This rule states that if the item of equipment being considered is a storage tank and if the level of the tank is below 20% then ar action must be found that will cause the level of the tank to increase over 20% in 600 seconds. The priority of the rule is 4. The Node must wait for the variable to change before 1000 seconds is up. If the change does not occur within the this time the Node fails. The rules name is C_LevelLw and its number is one.

3.4.4 Value rules

Value rules are used to deduce further information about the variables in the plant. They are the first rules to be activated in the expert system cycle. These rules take any piece of information in the system, and can be used to cause a variable change.

Example

IF

Item:Variable #= TKP18 And Item:Variable #= FTime;

THEN

SendMessage(TKP18, SetVar, FTime, (80 - TKP18:Level) /TKP18:DLevel);

This rule states that if a value is required for the variable FTime of Item TKP18, then the value to be used is (80 - TKP18:Level)/ TKP18:DLevel.

3.4.5 Link rules

Link rules describe the relationships between various variables. The antecedent describes when the rule can create a new link.

The consequent describes:

- The primary and secondary variables.
- The amount the primary variable is to be changed by in-order to alter the secondary variable by a specific amount.
- The amount this change will effect the secondary variable. In some cases the change made by the primary variable will not completely alter the secondary variable by the requested amount. This expression defines how much the secondary variable will actually change by. It has the most use when the primary variable is a discrete variable.
- The time lag between the primary variable being changed by the specified amount and the secondary variable changing by the specified amount.
- The range of secondary variation that the Link is valid for. The Link can only be used to cause a secondary variation within the range specified. This allows non- linear relations to be handled.

Example

IF

Node:Item #= TKP33 And Node:Variable #= SG;

THEN

List(Node:Item, P33, Node:Variable, Flow, crt*0.5, v, sx*t*v/(18*60), -2, 2, LSGP33Flow);

This rule states that if it is necessary to change the SG of the solution in tank TKP33 then the stream P33 should be changed by the amount:

TKP33:SG*t*v/(18*60)

where:

- TKP33:SG is the present value of the SG.
- t is a macro containing the available time for the final action to effect the Node created by the Link.
- v is a macro that contains the amount the SG is to be altered by i.e. the requested variation.

crt*0.5 defines the time it will take for the primary variable to alter the secondary variable. crt is the maximum time available to make the change and is apart of the objective definition.

The second v is the expression defining the resultant secondary variation and states

that if the flow of stream P33 can be altered by the specified amount, the SG can be expected to alter by the amount requested. This may seem to be a waste of information, but under some circumstances, a change in the secondary variable may not be able to cause the primary variable to alter exactly as desired.

In the example the Link is valid for any SG change from -2 to 2. The name of the rule is LSGP33Flow. The name is included in the rule for debugging purposes.

3.4.6 Action Cost rules

The Action Cost rule is used to specify which actions are more desirable under certain circumstances. The antecedent will check for various situations and whether the action that the consequent is to act on, is indeed being considered.

The consequent alters a cost value in that action. The cost value determines how suitable the action being considered is.

The rule is activated by the action being considered. This is done by the action placing a reference to the Item and Variable the action acts on. The rules antecedent must look at that Item and Variable slot before it can be used.

Example

```
IF
   If Action:Item #= P41 And Action #= Unchoke And TKP33:Level > 95

THEN
   SendMessage(Action,AltCost,-0.8);
```

The rule states that if the action being considered acts on stream P41, the action is called Unchoke and the level of tank TKP33 is over 95%, then this action is a good action to use. (Tank TKP33 will only reach 95% full if it's overflow pipe at 80% of the tank level is chocked).

3.4.7 Link Cost rules

Link Cost rules are very similar to Action Cost rules except that they direct the choice of Links used when searching for a secondary objective. Note that the name of the Link cannot be used in the rule as it is unlikely that the Link name is known. The Link could refer to the rule that created it.

Example

IF

Link:PItem #= CVP01 And Link:FVariable #= Frequency And Link:Variation > 0 And CVP01:ScrewCon #= OFF;

THEN

SendMessage(Link, AltCost, 0.5);

This rule states that if the link being considered is to change the frequency of the screw conveyor, and the screw conveyor is off, then the link should not be used.

3.4.8 Default rules

This set of rules are used in cases where no solution to an objective can be found. They would normally generate alarm messages.

Example

IF

Node:Item #= TKP18 And Node:Variable#= Level;

THEN

SendMessage(DAction_1, Activate, NULL);

If no action can be found to alter the tank level of tank TKP18, the default action DAction_1 must be used.

3.4.9 Conclusion

The rule system used in this expert system is fairly clumsy and could be improved in later prototypes. However the rules provide a mechanism of storing and using very general information. Rules are versatile as they can be used to convey general truths or very specific pieces of information.

Further improvements to the rule system would include activating the Action and Link Cost rules with the Control rules. The Actions and Links would know how desirable they are for use, before they are called. The advantages of this would be

time savings in the inference procedure and simplifications in the rules, as each rule would be aimed at a specific Action or Link making it unnecessary to place a direction expression in the antecedent.

Links are more suitable as instances, instead of being defined by rules. A better mechanism for Link creation should be considered.

3.5 Functions

Functions are used to control the interactions of the objects and form part of the inference system. As they are not part of the knowledge representation they will not be discussed in detail, but it is important to be aware that they are used in the expert system.

3.6 Conclusion

An expert system must provide a method to represent the knowledge used during the inference process. This system uses an object-orientated approach where the classes of the objects are used to define the inference procedure, but instances of these class form the knowledge base.

Various classes are used to separate various types of knowledge and inheritance is used to make the management of this knowledge easier. Further information can be deduced from the hierarchical structure used to define the inheritance.

The instances of the class structure Items and Actions were found to be easy to create

and manage. The knowledge was easy to obtain and describe. The shell provided suitable management functions for this knowledge.

Knowledge not suitable for storage in objects, is stored in the form of IF THEN rules. The antecedent describes when a piece of information can be deduced and the consequent states what can be deduced.

Control rules were found to be effective although improvements could be made to the consequent by defining much of the consequent in instances, and using the rules to trigger the instance. This would allow for easier debugging of the rules and a more general application of these rules. Originally this was not done, as it was felt that general control rules could be used enabling one rule to cover a number of situations. An instance could not provide this feature, but by using the rules to alter predefined slots in a standard instance, it is possible to over come this.

Link rules provided a general mechanism to create specific pieces of information, but were found to be clumsy. The Link Cost rules and Action Cost rules were found to be essential to direct the inference path.

The creation of knowledge is critical for an expert system. Although this dissertation concentrates of the structure used to deduce suitable control actions from the knowledge, the performance of the system is very dependent on the knowledge it requires to drive it. Some of this knowledge was found to be simple to obtain and create. Knowledge contained in the Link rules proved to be highly complex and difficult to obtain and manage.

CHAPTER 4 CONDITION RECOGNITION

4.1 Introduction

The subject of condition recognition has already been mentioned. In this chapter the issues involved with condition recognition will be discussed in detail. The control rules have already been discussed, although more detail must be provided to describe how the shell being used, handles them.

During condition recognition, objectives are formed. The chapter will discuss exactly what objects are, what happens when an objective is proposed, and the behaviour of the objectives in situations of conflict.

4.2 Conditions

A condition can be any pattern of process variable values. As conditions occur in the plant, variables describing the plant form particular patterns. By watching these variables it is possible to determine when a particular condition has occurred.

Variable values are measured using various pieces instrumentation. Values from the plant are gathered by a distributed control system (DCS). The data is transferred from the DCS to the PC via a serial link. The variable values are stored in the class Items.

The system looks at the plant variables as they are at the time of reading in the information, which is done cyclically when ever the inference from past conditions

has been completed.

The control rules use the variable values to determine the conditions that need to be reacted to and then create objectives in response to those conditions.

4.3 Control Rules

The control rule contains an antecedent that looks at various variables in the plant. The rule can be general and could look at a range of situations. Typical example is an antecedent that is true when any tank is over 80% full. A general rule of this nature would have to use specific data provided in the Items class in order to create a specific objective in response to the condition detected. An example of a control rules appears in chapter 3.

4.4 Application of control rules

4.4.1 Activation of rules

The mechanism provided by the shell being used, requires that the rules can only be used if the rule refers to an object:slot pair that has been placed on a rule stack.

During the read cycle of the expert system, all the variables are read in from the DCS, and each variable is placed on the stack e.g when the level of tank TKP33 is read from the DCS, the slot TKP33:Level will be placed on the stack.

If the control rule refers the variable, the system will try to apply the rule.

Should time ever become critical, only variables that change need to be placed on the stack, thus the system will only reason about new information. This mechanism has not been placed in the present system.

4.4.2 Objectives

An objective is some event that should take place in response to a condition. Objectives define a variable change in the plant. It is unlikely that a condition should occur that requires a reaction that does not result in a variable change to the process.

It is assumed that all the objective variable can be sensed directly, otherwise indirectly using value rules. If the objective variable cannot be sensed, a different management function must be defined.

An objective is defined as.

A variable of a particular plant item that is to be altered by a particular amount in a particular time.

Due to speed limitations of the system, objectives should not demand to be met under five minutes. This should be suitable for most situations in a process plant.

The information describing the objective is stored in the class Nodes, instances of which are created when the rule forming the objective fires. The Node created in response to a Control rule is called the root Node. The behaviour of root Node differs from Nodes that are created by Links.

4.4.3 Information required in rule consequent

The rule must provide the information required to create an objective. This information consists of:

- The variable to be altered, referred by the Item that contains it, and the actual variable name e.g. TKP18, Level.

- The amount the variable must be altered by.

- The available time to make the change. The time available to make the change is defined as the critical time, and is the time after the applicable action has been found, that the objective variable should be seen to change.

- A priority given to the objective.

- The objective contains a time duration that it should stay activated for. If the objective is waiting for a variable change it will presume to have failed if this time period is exceeded.

- The rule can also state whether the Node should watch for the variable change it requires or not. This later feature enables one to deal with cases where the variable change cannot be detected.

  If the variable change is observed, the Node will terminate after the change, and will mark the appropriate Link or Action as a success. If the Wait variable is TRUE, instead of the Node waiting for the variable change it will wait for the Hold time, and then end without incrementing any Success or Failure slots.

The Hold time is defined in the objective.

- For explanation purposes each rule is named, numbered and contains a message describing the reason for activation i.e a reference to its antecedent. It also contains a parameter used to indicate that the objective that it creates will be a root objective i.e. the first objective in the chain.

4.4.4 Activation of consequent

When an objective is to be created by a Control rule, a call is made to the function Obj transferring the information described above. Obj calls the Item that contains the variable to be changed. The call is to the Method Request which tries to create a Node that will contain the objective.

During the attempt to create the Node and define the objective, conflicts could occur. The Node and the Request method in Items handle the conflicts.

4.5 Conflict resolution

Various cases of conflicts can occur when the various rules propose objectives. A conflict occurs whenever an attempt is made to alter the same variable in more than one way.

Conflicts can occur between two objectives that are proposed during the same cycle, or conflicts can occur between a new objective and an objective that is waiting for an activated action to have the desired effect i.e the object is in the management phase.

The following table 4.1 describes the behaviour under each circumstance. The table states which Node will win, the old or new. The loosing Node is destroyed. If a path exists that contains a loosing node, it is also destroyed.

Table 4.1 Behaviour of root Nodes during conflict.

	Two conflicting objectives created during same cycle.	A new objective conflicting with an existing objective with a lower priority.	A new objective conflicts with an existing objective with a higher priority.	A new objective conflicts with an objective that could not be solved.
The two objectives change the variable in the opposite direction.	The objective with the highest priority wins.	The existing objective is terminated and the new objective is created.	The existing objective is left alone.	The existing objective is terminated and the new objective is created.
Both change the variable in the same direction.	The objective with the highest priority wins.	The existing objective is left alone.	The existing objective is left alone.	The existing objective is left alone.

Note that the table only applies to objectives created from rules i.e root Nodes. Objectives that are created during a search for an action behave slightly differently during conflict.

4.6 Conclusion

This chapter describes the recognition of conditions that have occurred in the process. Conditions are recognized by particular variable patterns and may indicate if a change needs to be made to the process.

The first step in the recognition procedure is to obtain variable values from the process. This information is gathered by a DCS and transferred to the PC. Rules then use these variables and look for conditions in the plant that must be reacted to. If a condition is recognized an objective is formed.

The objective describes a change that needs to be made to one of the variables in the plant. It states how much the variable is to change by, and the time available to make the change. This is necessary in order to determine a control action.

Once an objective has been formed, and that objective has not conflicted with any previous objectives that are still waiting for a certain response, it will be listed so that it can be processed. This processing forms part of the response search. This search is discussed in the next chapter.

The listed objectives are ordered according to their priority using a Shell sort. The objectives will then be processed starting with the highest priority. The most important Node is processed first ensuring that the most suitable action for it is found. It also ensures that searches carried out for less important Nodes do not need to be unnecessarily terminated when a conflict occurs.

In the prototype the rules used to determine conditions were very simple. Because of the structure of the shell being used, complex condition recognition can be used, as

long as it results in an objective. Defining the rules for the condition recognition proved to be fairly obvious provided the control strategy had been decided on.

Creating the objectives for each rule was instructive as it insured that the desired response to a control action had been thought of. If a particular control action is to be taken in response to some condition it is because that control action can achieve a particular goal. By using this goal to find a control action ensures that the goal of the control action is defined. Any control action that meets this objective can be used. Debugging the system is easier as it is possible to check the intention of using some control action in response to a condition as the result of the control action is well defined.

5.1 Introduction

Once the Control rules have proposed a list of objectives, a search is carried out to find an action for each one. When the operators carry out the suggested action the objective variable should vary by the desired amount. Figure 5.1 shows the state of the system thus far.

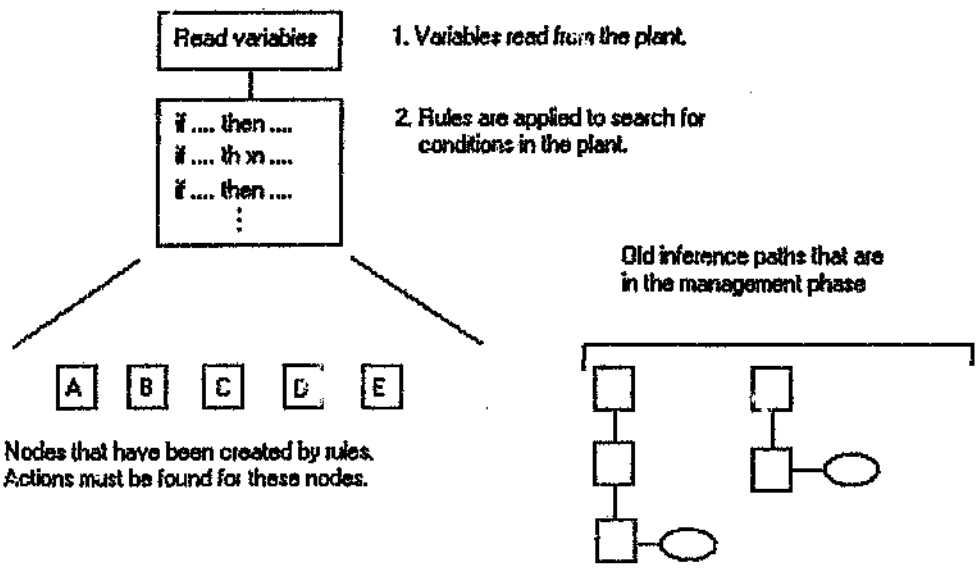


Figure 5.1 Overview of cycle thus far

The search for a control action starts by looking for an action that directly acts on the objective variable. If no such actions are available other variables are searched for, that when changed will cause the objective to be met. The information describing these other variables is contained in the Links.

Ultimately some action is found that will cause a variable in the plant to change, with the result that the variable in the initial objective will change by the amount required. The search for an action continues until a solution is found or all the options have been explored.

It is necessary to compare the various Actions or Links to determine which one should be implemented. The chapter will discuss the costing system that is used to form a basis of comparison between the various options.

It will first describe the basic procedure for finding a suitable action to meet an objective.

Chapters 6 and 7 will describe the Actions and Links in more detail.

5.2 Search Overview

5.2.1 Introduction

Up to this point the search for conditions in the plant has been described. Rules were used to recognize conditions in the plant, and objectives defined in response to those conditions.

It is now necessary to determine the control action that should be taken in response to the condition i.e. to find a control action that will cause the objective to be met.

5.2.2 Actions

Finding the correct action to meet an objective is the purpose of the search. The correct action is the action that will cause the objective variable to change by the amount specified within the required time.

Actions are described in chapter 6 but it is necessary to briefly describe their purpose and the information they contain.

The action stipulates how to change a particular variable. It defines:

- The variable to change. This is done by labelling a Item - Variable pair as discussed in chapter 3.
- The amount the action will change the variable by or to, depending on its function.
- A message presented to the plant operator indicating what to do.
- The time delay after the message has been presented to the operator, that the change should be seen by.

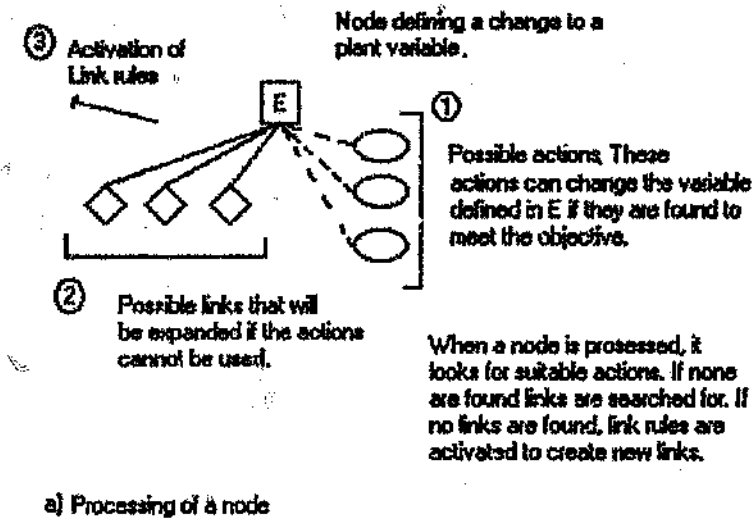
Other information required will be discussed in chapter 6.

5.2.3 Search for an action

The search for an action starts at the root Node which is created during the condition recognition stage. When an objective is created a Node is created describing that objective.

The Node is able to manage the search for an action that will meet the objective it

defines. It will first look at all the suitable actions. A cost value is determined for



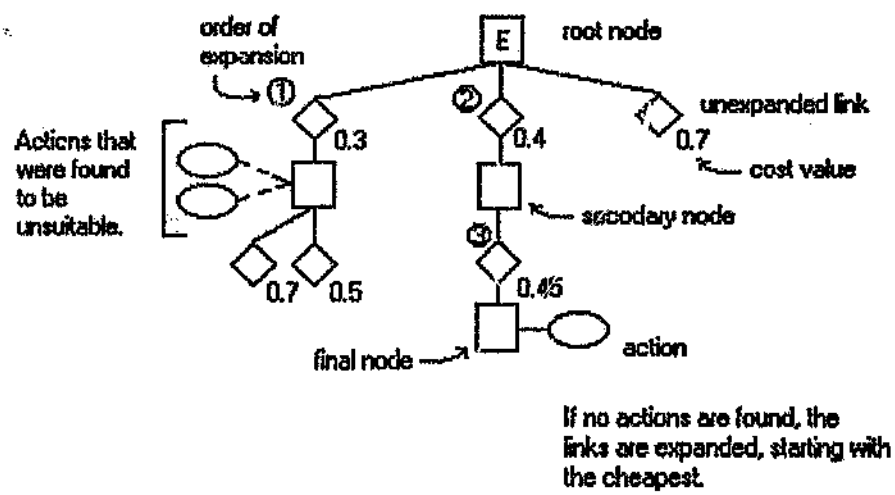


Figure 5.2 a & b A typical inference path

each action and the cheapest is chosen.

If no Action is found that can be used the Node will look for a Link it has found in previous inference attempts. The Link contains knowledge describing a primary variable that could be altered to cause the objective variable to change i.e. if no Action is found that can directly alter the objective variable a search is made for variables that if altered will affect the variable defined in the objective. A search is then carried out for an action that will alter the newly found variable. This is shown in figure 5.2.

During the search for Links a list of suitable Links is created and cost values are determined for each. The cheapest Link will be expanded.

During Link expansion the Link creates a Node with the objective of changing the primary variable by a calculated amount. Each Node that is created during Link expansion will look for a direct action, and if one is found the inference on the path stops, but if no action is found the Node will then search for Links that act on it. Each of these newly found Links are added to a global list. After the new Node has completed its search it will have either found a solution to the path or added new Links to the global list.

The global list is then sorted according to the predicted costs of each Link and the cheapest Link is expanded. This process continues until an Action is found or no more Links are available in the list. The search is known as a best first search[7]. It is not an exhaustive search but still tries to find the best possible solution to the problem. Figure 7.1 demonstrates this search procedure. The process will be described in more detail in chapter 7.

Once a path has found an action, it is activated. Activation is described in chapter 8 and is the process in which the inferer th is prepared for management. Actions are implemented at this point.

5.3 The costing system

5.3.1 Introduction

There are numerous paths that can be taken to meet a certain root objective, but normally there are only a few Actions or Links that are suitable for use under a particular set of circumstances. In order to compare these various options, a costing system is used.

The costing system determines a number between -1 and 1 and indicates how "costly" it is to implement an Action.

An Action that reacts very slowly could cause further damage to the plant and is therefore more costly than a more timely action. An Action that does not cause the variable to alter by the correct amount, is also considered to be more costly.

A cost value of -1 indicates that the action is very "cheap" i.e it causes the objective to be met without affecting many other variables and it causes the change within the required time. A cost of 1 indicates a control action that should not be used.

The costing system must be able to take independent pieces of information, and combine them in such a way that the total is not dependent on the amount of information available. The values returned must provide a common basis for

comparison.

There are various costing strategies that could be used, these include using Bayesian estimation, Dempster-Shafer theory of evidence and a method used by MYCIN[5].

The method used by MYCIN was felt to be the best system as it allows new information to be added without changing the cost values in the rest of the knowledge base. The MYCIN system is based on deriving a certainty factor for a particular hypothesis. In MYCIN, the system uses a Measure of Belief (MB) and a Measure of Disbelief (MD) to derive the certainty factor[5]. The part that is of particular interest to the project is how the certainty factors are combined.

5.3.2 Combination formula

Two certainty factors, X and Y, are combined to form a third certainty value using Eq 5.1.

$$
\begin{aligned}
CF(X,Y) &= X + Y - XY && \text{If } X\&Y > 0 \\
&= \frac{X + Y}{1-\text{Min}(X,Y)} && \text{If } X/Y < 0 \\
&= -CF(-X,-Y) && \text{If } X\&Y < 0
\end{aligned}
\qquad \text{Eq 5.1}
$$

The certainty factors range between -1 and 1.

In this project the certainty factors are called cost elements as they represent slightly different information. A cost of 1 means expensive i.e. the Action or Link should not be used, -1 means a large profit is made i.e. the Action or Link should be used, and zero is the break even point.

The formula always results in a number between -1 and 1 and should not be dependent on the number of pieces of information.

Impartial or uncertain information (cost of close to zero) has little effect on the overall conclusion, and conflicting costs (a factor with an opposite sign to the current total) results in an uncertain value i.e. a value close to zero.

5.3.3 Cost elements

Cost elements are calculated for each of the various factors that influence a decision. These factors are then combined with the above formula, to form a single value which forms a basis for comparison between the Links or Actions.

The final cost number will represent how suitable a particular Action or Link is, under the current circumstances.

Both the Links and Actions derive a cost element to take into account:

- Required **variation** of the specified variable.
- Number of **failures** of the Action or Link.
- User specified information in the form of **rules**
- The **time delay** between implementing the Link or Action, and observing the defined variable change.

Links also contain an:

- An estimate of what the rest of the path will cost
- A cost element passed down from the Node that called it defining the cost of

the path thus far.

Each of the cost elements are described .

Variation

The Action or Link chosen must produce the correct change in the objective variable. This cost element describes how closely the Action or Link will come to meeting the variation stipulated in the objective. The graph used to calculate the cost is shown in figure 5.3. Negative values are not used as the value -1 returned when the variation is exactly as requested, was found to saturate the costing system. The system is impartial to an Action that produces the correct change, but is certain not to use an action that does not cause the correct change.

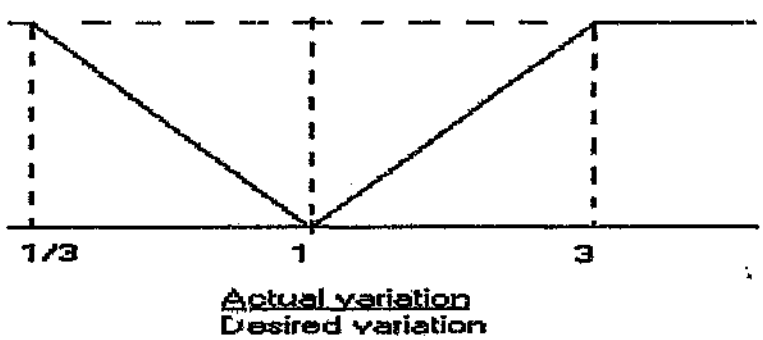


Figure 5.3 Graph used to calculate variational cost element

Time Delay

There is no point in using an Action that will not solve the problem in the allocated time. The cost element that takes this delay into account is calculated using the graph

in figure 5.4.

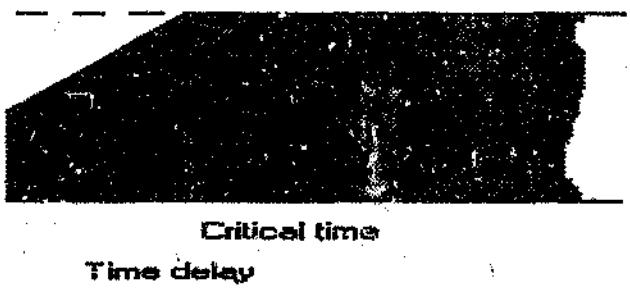


Figure 5.4 Graph used to calculate delay cost element

The cost is never below zero as there can never be a negative delay. Delay time is normally detrimental to a control loop and hence the positive cost values.

Failures

The success rate of a particular Action is taken into account in determining a cost value for an action. If a particular Action is used that never seems to work, it will be less likely to be used in the future (depending on what other options are available).

The failure cost value is calculated using equation 5.2

$$Cost = \left(\frac{No.Failures}{No.Failures + No.Successes} - 0.5\right) \cdot 1.6$$

Failures cost will help enforce the consistency of a decision. If a decision is taken that works often, it will be more likely to be used again. The failures cost forms a simple learning mechanism that is updated whenever a node succeeds or fails.

Heuristics

Heuristic rules take the form of Action Cost rules and Link Cost rules, which are used to direct the inference process. The rules have already been discussed in chapter 3. To revise, Link Cost rules allow the knowledge engineer to manipulate the search for secondary variables so that the most suitable variable is used for the existing circumstances.

Link Cost rules also allow the effect of changing a variable to be taken into account e.g. if a possible solution to a problem is to increase the flow into a tank that is already full, a Link Cost rule can be used to ensure that proposal is not used.

Action Cost rules allow the user to define which action is more suitable to act on a variable when taking conditions in the plant into account.

The costs of the rules are combined to the total cost last. This is done incase the rule contradicts the conclusion from the rest of the cost system. The rule will have the power to over ride the rest of the cost value.

Node cost

The Node cost element takes into account, the cost of the path to get to the Node being considered. Normally the cost of each Node in a path will become higher as

one moves down the path. This is more realistic as there are more variables being altered as the path becomes longer, thus there are more secondary effects to the control action. The more secondary effects an action has the more expensive it is to use as it is more likely to have detrimental consequences.

The Node cost element combines the Delay cost of each Node, and the cost value resulting from the Link Cost rules applied to the path. Originally a cost element describing the attractiveness of changing a variable was built in. This cost element would be high if a variable was to be moved off a set-point for example. This element was removed as it over complicated the system.

5.3.4 Lack of associativity of the costing system

The MYCIN system is not associative. The table below shows the effect of combining three arbitrary cost values in different orders. The final column is the total of the three middle numbers using equation 5.1 to combine them.

Table 5.1 Lack of associativity in MYCIN system

1	0.5	0.7	-0.8	0.027
2	0.5	-0.8	0.7	0.416
3	-0.8	0.5	0.7	0.416
4	-0.8	0.7	0.5	0.416
5	0.7	-0.8	0.5	0.416
6	0.7	0.5	-0.8	0.027

The result is that one must be careful of the order the cost elements are combined. Note that problems are only experienced if the two parts to be combined are of opposite sign and thus contradict each other. This situation indicates uncertainty in the final conclu

In the system they are combined as follows:

Actions: CF(CF(CF(Variation, Failures), Rules) , Delay)
Links: CF(CF(CF(Estimate, Failures), Node) , CF(CF(Delay, Variation),Rule))

The combination used could be improved but this combination was found to be adequate. In the sequence for actions, the Delay cost is dominant. It would be better to swap this cost element with Variation, making the variation dominant. For the Links, the system used was the easiest to implement, but it would be better to use a system such as:

Link: CF(CF(CF(CF(CF(Estimate,Node),Failures),Delay),Variation),Rule)

By combing the rule cost element towards the end, if it has an opposite sign to the total, it will have the most effect. This allows one to over ride decisions.

5.4 Conclusion

Once an objective has been created from the control rules, a search for an Action to meet that objective is initiated. The Actions that can directly affect the variable being

worked on are considered first.

If no suitable Actions can be found a best first search is carried out for new objectives. If the new objectives are met they could cause the primary objective to be realized. The new objectives are attached to the old objectives to create an inference path which describes the propagation of variable changes through the process.

The search continues until an Action can be found for any objective in the inference path. The action will cause a variable change in the process which will propagate up the variables described in the inference path.

A costing system based on the system used by MYCIN is used to generate values to compare various Actions and Links. The system uses cost values derived from a description of the Action or Link it is comparing. These cost values take variation, time delay, success rate and directed cost (rules) elements into account. These cost values are combined to create a value between 1 and -1. A cost of -1 is more suitable than a cost of 1. The costing system was found to work well and formed an adequate basis for comparison between the various possibilities.

The next two chapters will discuss the search for a suitable Links and Actions in more detail.

CHAPTER 6 ACTIONS AND THE ACTION SEARCH

6.1 Introduction

This chapter will discuss the structure of the Action class and it's behaviour. It describes the information contained in the action and how the action uses the information.

The Actions class defines the possible control actions that can be taken on a plant. The actions are created by the knowledge engineer.

When the action is used it will present a message to the plant operator giving instructions to be carried out. A typical instruction would be to alter a valve position by half a turn clockwise. The action could also cause some variable to be directly altered in the expert system.

6.2 Actions purpose

The action is the final result of the inference process. It defines the control action that is taken to cause some change to the process being controlled. This control action is used to meet some objective which resulted from a condition found in the process.

The action contains specific instructions to the plant operator. These instructions indicate exactly what control action needs to be taken. The instructions are displayed to the operator from the PC. The operator is expected to acknowledge the action

mes age. This is necessary to ensure that the learning mechanism is not foole. by the operator not executing the specific actions.

The action does not need to give the operator a message. Instead it could alter a variable in the expert system.

6.3 Action data

The action must contain particular pieces of information, to describe what it does and how it can be expected to behave. This information is provided by the knowledge engineer. The knowle is contained in slots with in the action. These slots are listed below:

- **Item:** The piece of equipment that the action should act on e.g. CVP01 which is a screw conveyor.
- **Variable:** The variable within the equipment that is changed by carrying out the action e.g. Frequency.
- **Variation:** Either the amount the action will cause the variable to change to, or the amount the variable will change by, depending on the action mode. E.g. 50 Hz. After the operator has carried out the instructions in the message, the variable should change by/to the amount stipulated here.
- **Mode:** FIXED or DELTA. If the mode is FIXED the action will calculate the difference between the present value and the fixed value that the variable should be set to. This calculated value will be used as the variation of the variable. If the mode is DELTA the action should alter the variable by the specified amount.
- **Rule number:** A rule number is written to the DCS to enable a station

removed from the PC to determine which rule was activated. It is possible to use this information to create user messages from within the DCS.

- **Tag:** The tag name that the rule number is to be sent to.
- **Message:** The message to be displayed when the action activates. E.g Increase the screw conveyor speed to 50 Hz.
- **Time:** The time delay after the action has been activated, that the variable change will be seen. E.g. 300 sec.
- **Initial cost:** An initial cost value.
- **Action type:** MANUAL or AUTO. When an action is activated it can either display a specific message to the user or alter a specified variable in the expert system. This later feature can be used over value rules, when a complicated inference procedure is required.

This information is provided by the user to specify the control actions that can be taken.

The control actions are fixed i.e. the expert system is not able to generate control actions from general templates. This was originally done so that the control action could be displayed from the DCS, which is less flexible than the PC.

6.4 Action search

The search for an action is carried out from the Nodes. When a Node is searching for an appropriate action, it forms a list of the actions that will affect the objective variable.

The Node calls the method Precondition in each Action which returns a cost number

back to the Node. If the cost is lower than the maximum allowable cost for an action of that priority then the delay cost component is added by the Node. If the resultant cost is the cheapest thus far, the cost and the action name is stored. The cheapest Action is the name stored at the end of the search.

Once the Node has found an Action, it returns a message FOUND to the control function. The message causes the main control loop to stop searching, to clear the Links used, and to activate the path. Activation is described in chapter 8.

6.5 Cost criteria

The cost value returned by the action incorporates:

- Variational cost
- Failures cost
- Rule cost
- An initial cost.

Variational costs and Failures costs were discussed in chapter 5.

6.5.1 Initial costs

The initial cost is used to indicate whether an action is always the most suitable action available. A set of actions could act on the same variable, causing similar effects. The user can use the initial cost values to indicate a preference.

It can also be used with actions that should only be used under specific

circumstances. The action can be given a high initial cost and Action Cost rules will be used to reduce this cost if the correct situation exists.

6.5.2 Rule costs

Action Cost rules are activated when the rule refers to the Item slot in the Action. The Action's method Precondition places the Item on the stack, and activates the chaining process. When the antecedent is satisfied, the consequent causes a cost value to be combined with the total cost of the Action. The Action Cost rules are used to indicate when a particular action should or should not be used and is very useful when dealing with exceptional circumstances.

A typical example of an exception is when an Action requests a particular stream line to be un-choked. The Action is created with a very high initial cost, typically 1. Normally the Action will be ignored because of it's high cost. If particular condition exists such as the feed tank to the chocked line is getting fuller, an attempt will be made to increase the flow through the choked line. The Action to unchoke the line will be considered. Normally because of the Actions high cost it is ignored but because the Action Cost rule can be used to reduce the overall cost of the Action below the threshold value when symptoms of a choked line appear, and the Action will be found and used.

6.6 Activation

Once a rule has been found, it must be implemented. This phase is called activation. When a Node activates an Action it calls the method Activate. This method causes the Action to list itself in an Action list and marks itself as Activated. The Node then

notes the amount the action should cause the variable to change by, and when the change should occur.

The actions are removed from the list at periodic intervals, and the action message is presented to the operators.

When the action is presented to the operator a message describing the initial condition, and the action to be carried out in response to that condition, is displayed. The operator is expected to acknowledge that the action has been carried out.

Actions that have not been acknowledged are assumed not to have been implemented and therefore do not fail.

Should an action be found and an attempt made to activate it fails, the Nodes created in the inference path must be destroyed and the inference started over. This is done as it is difficult to return to the inference path.

6.7 Action failure

An Action is considered to have failed if it did not cause the specified variable to alter in the specified time. The Action only increments the Failure slot if the operator acknowledged executing the action. When an Action fails, it is not available for a period of time. By marking an Action as failed it will be unavailable for a period of time enabling a different Action to be found later.

6.8 Action success

Actions that are successful increment a Success slot. Actions that succeed are also not available for a specified time, as there is often no point in carrying out an Action that has just been implemented. If a situation occurs that requires the Action that has been implemented, to be applied it will find some other action, as the action taken has obviously failed to affect the condition being considered. Situations can occur where the propagation of the effect of the action is still to be seen. This case was ignored.

6.9 Conclusion

The Actions class describes the individual control actions that can be taken to control the process. Each Action will only affect one variable. When the Action is called it will return a cost value indicating how suitable it is. This value is used by the Nodes to pick the best Action. The Node will then activate the Action. During activation the Action is listed and will be displayed to the operator.

Actions were found to be very easy to create and manage. The reason is that they are assumed to affect only on primary variable. The creation of Actions tended to be a bit tedious as every possible control action needs to be found. The Actions were grouped according to the item of equipment they act on, making the management easier.

Deciding on delay times and initial cost values required that the system had to be "tuned".

The Action mechanism could easily be adapted so that the message could be created

depending of the Actions requirements. This would make the Actions easier to create, and less repetition would be required.

Actions are searched for before Links are considered. If a Node finds an Action first it will be used before looking for suitable Links. Normally this works well, but occasionally a better Action could be found if the Links were used first. To work around this problem requires the use of Action Cost rules. In future systems the search for an Action should be simultaneous with the Link search. This issue will be discussed in more detail in chapter 7.

CHAPTER 7 LINKS AND THE SEARCH FOR A SECONDARY OBJECTIVE

7.1 Introduction

After a Node has searched for an action that could cause it's objective variable to change, and none is found, the Node will search for a secondary objective that if met would result in the objective the Node defines being realized. Each of these secondary objectives will in turn become a primary objective, and search for a suitable Action or another Link and hence another secondary objective. During this process inference paths are created describing changes to the process.

As an example of a secondary objective consider two solutions that are mixed in a tank to a constant pH. The primary objective is to change the pH. A secondary objective would be to find an action that will effect the flow rate of either solution. Thus the secondary objectives would change variables that are found to affect the primary variable. In order to find the secondary objectives it is necessary to know the relationships between the pH variables.

This knowledge is contained in instances of the class Links. Links are created from knowledge in Link rules. The knowledge in the Link rules can be general or specific, but the Links themselves are always specific. The Link rules are created by the user.

Before discussing the search for secondary objectives the Links must be described.

7.2 Links

Links are critical parts of the search for a secondary objective. They contain knowledge that describes how variables interact. This knowledge is created by Link rules or directly by the user. An example of a Link rule appears in chapter 3.

Links contain a description of:

- **The primary variable.** This is the variable that if changed, will cause the secondary variable to change. The objective created by the Link will alter this variable e.g. The primary variable for a tank level is the flow out of the tank.
- **The secondary variable.** This is the variable that is described in the primary Node and is the variable that needs to be changed by a specific amount.
- An expression used to calculate the **desired variation** of the primary variable. This expression stipulates the amount the primary variable must be altered by. Normally this expression is a function of the required variation of the secondary variable.
- An expression to calculate the **resulting variation** of the secondary variable if the primary variable alters by the correct amount. The expression returns a number predicting the secondary variable's variation. This value will be compared to the amount the secondary value should change by, and a variational cost element will be derived using these two values.
- The time after the primary variable has changed that the secondary variable will change by the amount specified.

7.3 Link Rules and Link Creation

Links are specific, they link one variable to another with a specific relationship. However the knowledge used to create the Link could be more general.

A Link could state that in order to reduce the rate of change of the level of any tank the flow out of the tank must be altered by -dl/dt*(Cross-sectional Area). This is could be true for all tanks. However an unique Link is required to specify the relationship between each specific tank. This is necessary as the use of each Link depends on the specific circumstances surrounding the item it is to affect, also information such as the Estimate cost and Failures cost will be specific to each Link.

The Link rules provide a mechanism to allow general knowledge to be used to link variables together. When the Link rule is fired, it takes specific conditions and tests to determine if these conditions can be applied to a general rule. If so, a specific Link is created.

The antecedent of the Link rule refers to a Item - Variable slot pair that is contained in the Node for which the Link is to be used. The Node:Item refers to some item e.g. TKP18, and the Node:Variable is some variable e.g. Level, that the Node is trying to alter.

The Item - Variable slot pair could be specific in which case the antecedent of a Link rule could refer to:

Node:Item# = TKP01 And Node:Variable#= Level;

In this case the Link will only be created for a Node trying to change the level of

Or the antecedent could be general e.g.

IsAKindOf?(Node:Item)#= Storage And Node:Variable #= Level;

In which case the Link rule could be activated by any Node that describes a change to the level of a storage tank.

When the antecedent of a Link rule is true the consequent will call a function called List. The following information is passed during the call:

- The Primary variable.
- The Secondary variable.
- An expression to calculate time delay.
- An expression to calculate the variation required of the Primary variable.
- An expression to calculate the resultant variation of the Secondary variable.
- The maximum variation of the secondary variable that the link can be used for.
    - The minimum variation of the secondary variable, that the link can be used for.
- The name of the Link rule.

List() takes this information and creates a new Link. The information passed to List() contains specific information, although the rule that created that information could be general.

Note that the Link is defined to be valid for a particular range of required variations. This allows one to handle non-linear relationships between the primary and secondary

variables. Approximate linear relations can be used over a small range of the relation. Numerous Links are used to describe the complete variation range.

The Links are grouped according to the item of equipment that the secondary variable belongs to. The name of the Link is derived from the secondary variable and a suffix of the next free number of Links e.g. Flow_11.

7.4 The inference path

The connection between a initial objective and a final control action is defined by an inference path. This path is created during the inference process. This process will be discussed in this section.

The inference path describes the interactions of variables required in order for the root objective to be met. It is built using Nodes, these Nodes are inter-connected using Links.

As already described the Links define how variables interact. The Nodes define changes that need to be made to a process. Knowing how a variable needs to be changed, and using the Links, it is possible to define secondary objectives that if met will cause the necessary change to the primary objective.

7.4.1 Creation of an inference path

The creation of the inference path is based on a best first search technique[6]. Objective of the search would be to find and create the best path to a control action.

To find the best solution to some situation requires an exhaustive search which could take some time. The best first search is a compromise that uses the path that looks the best at the time. To ensure that the best solution is found during most circumstances, an estimate of the total cost of taking some path is made. A description of this cost value is given later.

The search technique is best explained by means of an example.

Example

Let us consider an attempt to change a tank level in a tank called TKP01. This tank was full and a response to this condition requires that the tank level is reduced by 20% in one hour. This objective forms the root objective and is shown in figure 7.1a. The Node is called NTKP01_Level.

The Node has looked for an action that will directly affect the level, and found none. The Node will find Links that list the Nodes variable as a secondary variable. In this example three Links are found. L1.1 indicates that to decrease the tank level, it is necessary increase the flow out of the tank. The other, L1.2 indicates that to alter the level the flow into the tank should be altered.

The third link contains a heuristic connection; it stipulates that to alter the tank level, an unrelated flow rate called P12 must be altered some where in the plant. From experience, the operators have found that under particular conditions, this helps.

A call is made to these Links, passing the required variation and the available time. This results in a cost value indicating how suitable each Link is. The cost values are discussed later. The Links are listed in a list called the Unexpanded list.

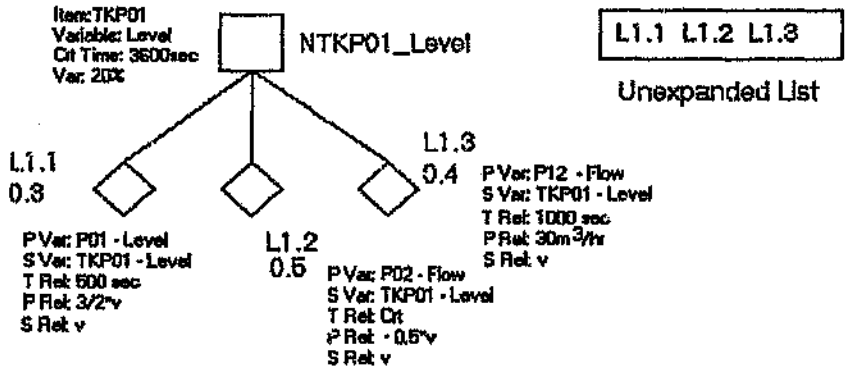


Figure 7.1 a The Root Objective

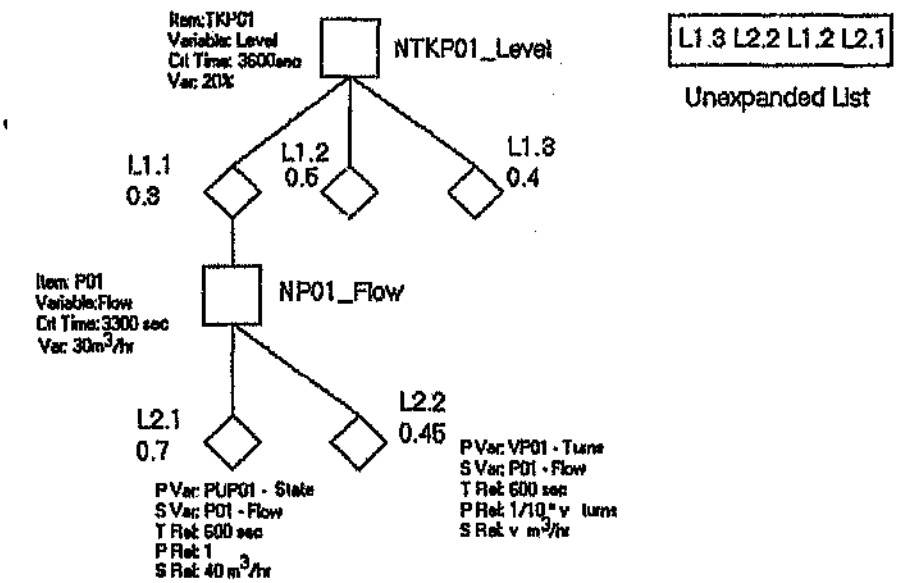


Figure 7.1 b Expansion of the link L1.1

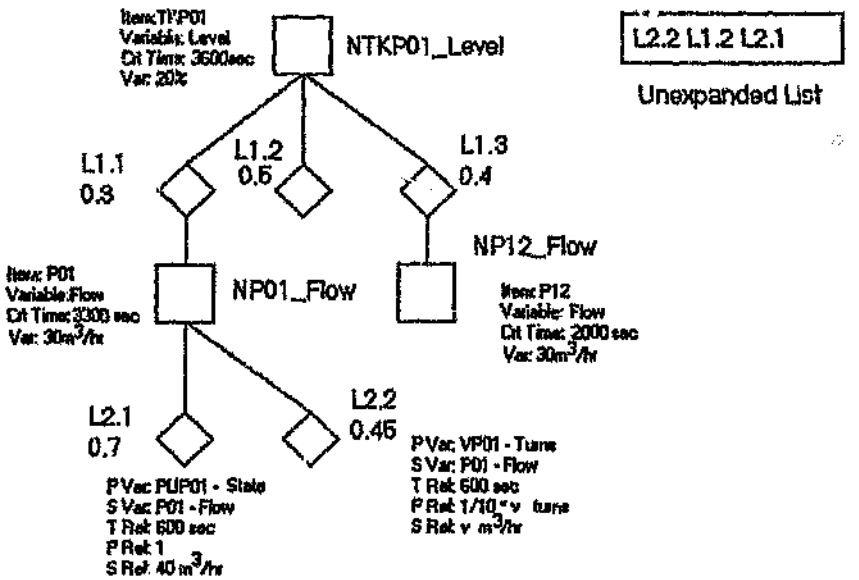


Figure 7.1 c Expansion of the cheapest link

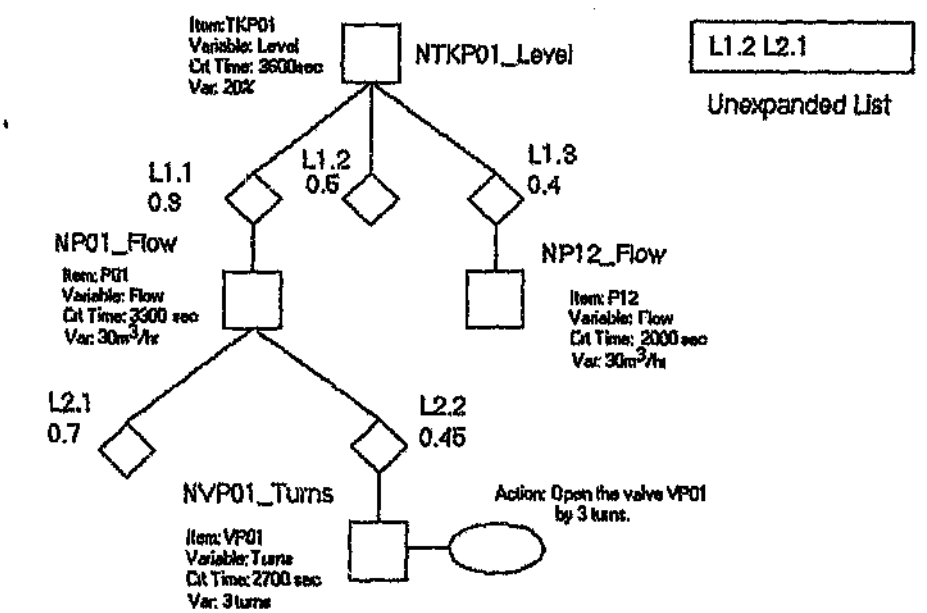


Figure 7.1 d Finding a control action

Now move onto figure 7.1b. The cheapest Link in the Unexpanded list is found. This Link is L1.1 which links the level to the output flow rate P01. If the estimate cost in this path is accurate it should lead directly to the best action. This Link is expanded. Expansion is described in detail later, but briefly the Link now creates a new objective defining how to change the flow rate of P01. This Node is called NP01_Flow.

No direct action can be found to alter this flow rate, but the Node NP01_Flow is able to find two other Links that will effect the flow rate. These are links L2.1 and L2.2. L2.1 describes the relation between the flow rate and a pump. L2.2 finds the relation between the flow rate and a valve. The costs of these Links are calculated. It turns out that this path may not be that cheap after all.
The Link L1.3 is predicted to provide a cheaper action than the other Links, it has a cost value of 0.4 as apposed to 0.7 and 0.45. Thus the Link L1.3 will be expanded next, as shown in figure 7.1 c. Switching to the cheapest Node is a part of the Best First search mechanism. The Node NP12_Flow which is generated as a result of this Link is unable to find any Actions or Links.

The search has not yet ended so the next Link is expanded, this is Link L2.2 which defines a change to a valve. The new Node NVP01_Turns is able to find an action which requires that a valve be opened by three revolutions. Remember that the action defines how much it will effect the primary variable. The primary variable for the action would be to alter the valve position. Because an action has been found, the search will end.

Discussion

This example has described the process of finding an action that will ultimately effect the primary Node. By changing the valve position, the flow rate of stream P01 will be altered, and hence the rate of change of the tank level. If the rate of change of the tank level has been altered by the correct amount, the tank level should change within the specified time.

The best first search uses a cost estimate that is derived from past experience, to determine how suitable a Link is. This estimate would reflect the cost of the action to be found, and the time delay when implementing the action. The cost values determined for each link is described in section 7.5.

The inference path predicts how each variable should change after the action is executed as one moves back up the path. This information is used in the management stage which is described in chapter 9, thus the inference path must be cleaned up and prepared. This will occur during activation which is described in chapter 8.

7.5 Cost Elements

Cost elements are calculated in order to compare the various Links which define the form of the inference path.

The cost of a Link to be expanded depends on:

- The delay between the secondary variable changing, and the primary variable being changed.

- The amount the secondary variable will be changed by, if the link is used.
- The success rate of the link.
- An estimation of what the total cost of using this path. This estimation will depend on the cost of the final action found, and the cost of the Nodes between the present Node and the Node that finds the action.
- The cost of the path thus far.
- User cost provided using Link Cost rules. The cost can be used to alter the total link cost and hence the inference path.

7.5.1 Estimate costs

The estimate cost is a critical part of the search technique. It is an estimate of what it will cost to continue down the present path, thus it tries to predict what is up ahead.

At each Link, the total cost to reached the current point is determined. The true cost of the link should also indicate how many other variables need to be altered in order for this variable to change, and how expensive the action that is ultimately found is.

This information is not definite but can be estimated from past experience. Once a path has been used, and has resulted in an action being found, cost values are returned up the inference path during activation. The cost of the path is determined and the estimate is updated.

The updated cost value is calculated by taking a cost value from the secondary node. This value includes the final action cost and cost element of each Link used in the path. The cost of the Link combines the values of the Link Cost rules, delay and variation.

7.5.2 The use of link cost rules

Link Cost rules allow factors that are not directly considered, to influence the use of various Links. Under some situations certain Links are significantly better than others although their standard cost values do not indicate this. A typical example, there are two flows feeding into a tank but only one flow can be altered at a time. The Link Cost rules will cause the flow being used at present to be altered.

The Cost rules allow external situations to be incorporated into the decision. These rules can become very important when trying to describe some situations.

7.5.3 Node Cost

The Node cost is a cost value passed in to the Link when each Link is found. The Node calculates the Node cost by using the cost passed down during expansion, and the delay between it and the previous Node.

A description of the other cost elements have been discussed in chapter 5.

7.6 Link Expansion

When a Link is called using the method Call, it lists itself in the Unexpanded list. If the Link is the cheapest Link in the list, it will be expanded. During expansion the Link creates a Node that contains the objective to alter the primary variable in the Link by the variation calculated. The Link also calculates the critical time for the Node i.e. the time available for the decision to have an effect.

If the Node tries to alter a variable that is being altered by some existing objective,

a conflict will occur. The Node acts differently depending on the circumstances of the conflict. If the Node is not able to expand the search continues as if that Link were not there.

7.6.1 Conflict resolution

During the expansion of a Node several conflicts could occur. These are best represented diagrammatical and are shown in figure 7.2.

These are:

- Looping back. The Link finds a Node in its own path. If this Node is used no solution will ever be found.
- Conflict with an activated Node with opposite variation. If the new Node has a higher priority a copy will be made of the existing node. The result is two Nodes that act on the same variable at the same time. This will be discussed later.
- Conflict with an activated Node with similar variation. In this case it is possible for two paths to share a Node.

These conflicts are detected by the Node during creation. When the new Node has a lower or equal priority compared to the conflicting Node, it will not be created and if it is created it will not activate. Because all the Nodes in a path have the same priority, no looping back will occur.

Normally if a conflict occurs and the new Node has a higher priority, a copy of the Node will be made. If the new Node is activated, the lower priority Nodes will be terminated. The old Node is only terminated when the new Node is definitely needed.

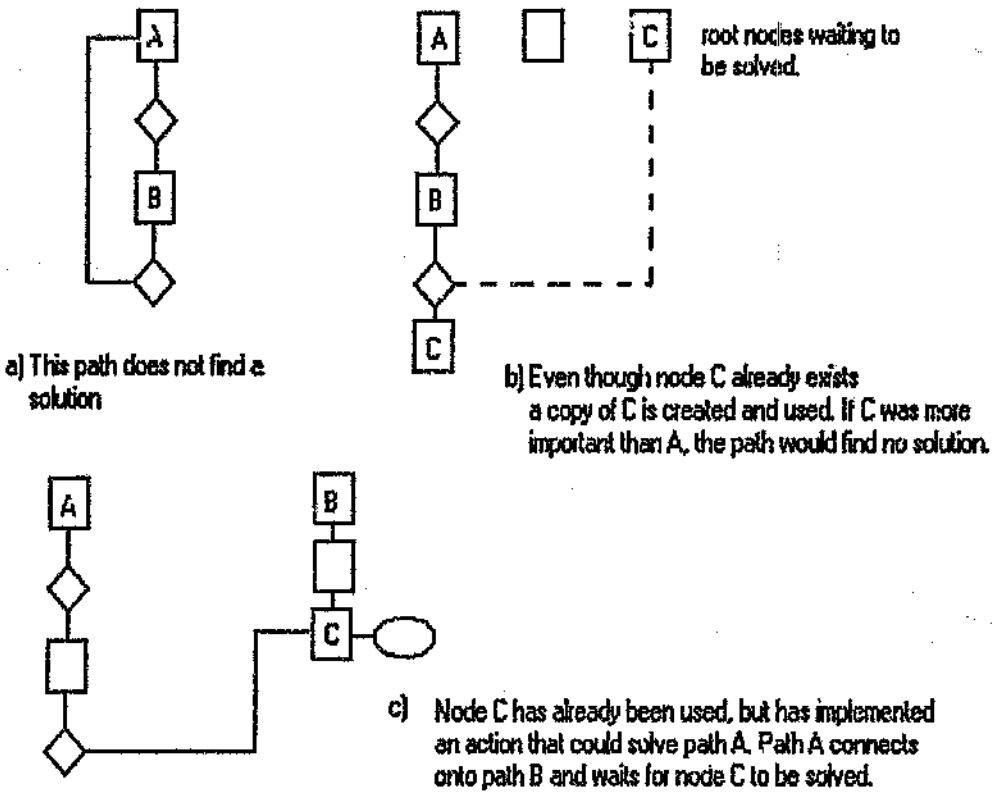


Figure 7.2 Conflicts during link expansion

7.6.2 Copies

In order to meet some primary objective, it may be necessary to consider an secondary objective that has already been used. Because of the system architecture, in order to consider changing a particular variable it is necessary to create a new Node.

Normally only one Node can be created for one variable as it is only possible to

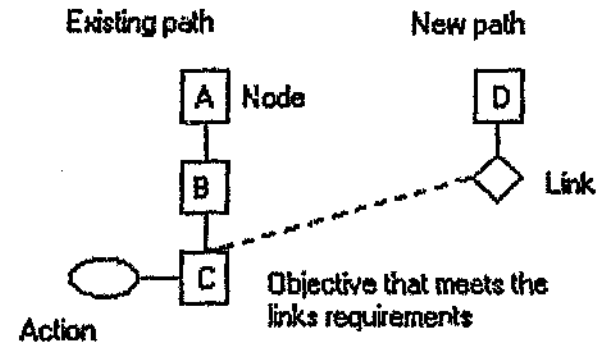
make one change to a variable at a time. However, if a high priority condition is searching for a secondary objective that has already been used, it may be necessary to terminate that existing objective, and replace it with a objective more suitable for the high priority condition. It is not possible to terminate an existing objective if the option of using that condition is merely being considered as that objective may not be used. For this reason when ever a conflict occurs under these circumstances, a copy of the objective is created.

A class called Copies contains a list of all the objectives that will alter the same variable i.e they conflict. However only one activated objective can exist that will alter a variable. Table 4.1 is applied during activation and the loser commits suicide. Note that two copies act on the same variable, but may alter the variables by totally different amounts.

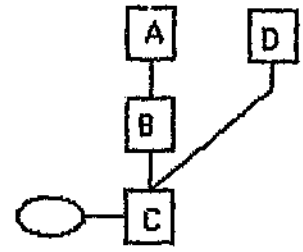
7.6.3 Using existing Nodes

In some situations paths may exist that contain Nodes that require similar changes to a secondary variable. If this situation occurs, the new objective will register itself with the existing objective that causes a similar variation. When the existing secondary objective ends, it will send the appropriate message to all the primary objectives that use it.

This was done to ensure that the new Node does not find a sub-optimal action when the most suitable action has already been used. It will also not reactivate an action that has already been implemented. Figure 7.3 shows a path where the Node is using a Node from another path.



a) The inference path during the search.



b) The inference path after activation.

Figure 7.3 a & b Inference paths sharing Nodes

7.7 Clearing

Once a path that the Links were used in, has found an Action, the Links are cleared so that they can be used for other objectives. This process is called Clearing, and is initiated by the control function. Once cleared, the Link will have no record of how it was used.

7.8 Percentage Failures

Like Actions, Links also contain a cost value indicating how successful they are. Every time a Link is expanded, it percentage failures decreases. Every time the Node that the Link created failed to change the primary Node, the Link's percentage failures increases.

This feature helps to ensure consistency in the decision process.

7.9 Conclusion

Links are critical for the search for relevant actions. The Node and the Links specify the few Actions that could provide a suitable control action for the plant. The use of Links has several weaknesses which could be over come by improving the method of Link creation and search.

7.9.1 Evaluation of Best first search

A best first search technique was used to find the best path to an action. Practically

the best action for a particular situation was not always the cheapest. To solve this problem Action Cost rules were introduced. These would be used to direct the choice of an action.

Because of the best first search technique, in order to find this particular action, the path to the action has to be the cheapest as well. Often this means the cost of a path must be manipulated so that it always appears the cheapest. This can be done using Link Cost rules but the knowledge base becomes more difficult to manage.

For other situations the best first search worked well. Normally in these situations, the particular path chosen was the fastest or provided the closest variation.

The best first search is good in situations where there are many paths to actions that can be taken in response to some condition. It does not preform well when it is necessary to keep the inference on a particular path as extensive Link Cost rules must be used. This situation occurs when for a particular situation only one solution can be used out of a number of choices.

Because of the % failure, once a decision has been made under particular circumstances the same decision is normally taken again.

7.9.2 Action Link separation

The present system searches for an Action that will affect a variable. If no Action is found it searches for a secondary variable to cause a variable change i.e. it searches for a suitable Link. This was done because it was assumed that an action that directly affects a variable will always be better than an action that could cause many other variables to be affected before altering the objective.

This is not always the case. An example, a particular flow rate is altered to keep a pH constant, but under exceptional cases lime must be added. When implementing such a system a Link is created between the pH and the flow rate and an action is created describing how to alter that flow rate. An action describing the addition of lime is associated directly with the pH. During the inference procedure under normal conditions, the system finds a objective to alter the pH which is linked to the flow and the action to alter the flow will be found. Unfortunately the system will find the action to directly alter the pH i.e add lime before finding the better action to alter the flow rate. To solve this problems Action Cost rules are required to ensure the action to add lime remains undesirable unless needed.

This problem could be overcome if the Links and Actions were compared at the same time. In this case the Link to the flow rate would appear to be the best option for the situation.

To implement this, both Actions and Links should be placed on the Unexpanded list and the search should continue until an Action was the first in the list i.e cheaper than any of the available Links.

By comparing the Actions and Links at the same time, the best solution to a problem can be found with out the need for Action Cost rules to eliminate options. It also allows further versatility to the system.

7.9.3 Link creation

The Links were created from Link rules. The reason is that the rule mechanism is well suited for generalities.

For example: A Link rule could state that if the level of any tank must be changed, reduce the flow rate of the input stream. The input stream is defined in the Item class. Thus this general rule can be applied to any tank.

If a Node is searching for a Link that changes a particular tank level, and none is found, this rule will be activated, and from the general information in the rule and in the specific description of the tank, a new Link can be created specifically for the tank.

Two problems occur with this approach:

- When the rule changes, the information deduced by the rules becomes invalid. It is necessary to revoke all the Links created by the rule.
- Knowledge used in the rules is difficult to create. It is often easier to create direct Links, instead of general Link rules as large amounts of information is required to create the general Link rules. In the example mentioned above, the input streams for each tank would have to be defined and the dimensions of the tank would be required for the variation expressions.

7.9.4 Inadequacy of link search mechanism

If a rule has created a Link, which may only be done under specific conditions, the Link is always available. This Link will be looked at before new Links are created. The Link will always be found if the secondary variable must be changed, even if the Link is inadequate for the particular circumstance. To rectify the condition Link Cost rules are used. Often information in the Link rule is repeated in the Link Cost rule.

The Link cost rules have been found to be easier to create if they are very specific.

7.9.5 Gathering information for links

The information in the Links describes how the variables interact. This information is available from the process. From an understanding of the process being controlled it should be possible to obtain the necessary information to describe the variable interactions.

In the plant that the system was applied to, the information describing the process was not available. A detailed process study would have to be done.

In order for the correct action to be found it is not necessary provide the correct variable changes so long as the knowledge creation is consistent. Problems occur when the system expects the process to physically change by the amount specified. This will be discussed further in the chapter discussing management.

7.9.6 Adaptive mechanisms

It was initially felt that because the plant structure that is being controlled changes, the knowledge used should be adaptable. Actions used at present may not work as well in the future. Knowledge claimed to be suitable by the knowledge engineer may also present problems in practice.

A method to counter-act this is to measure how successful each Action is. If an Action is found to fail often, it should not be used. In order to accomplish this, the number of times the Link results in a Node that succeeds or fails, is recorded and this information forms part of the total cost.

The effectiveness of this adaptive mechanism was never fully tested and it could be neglected in an initial prototype. It will only be useful when dealing with situations where two possibilities could be used, that are very similar. The system will tend to use one instead of the other, enforcing the consistency of it's decision.

7.9.7 Recommended developments

Search

The search technique should be improved. A better alternative is to place Actions onto the global:Unexpanded list as well as the Links. The search continues until the cheapest Action is found. If Links are cheaper, they will be expanded first. If an Action is cheaper than the Links, it will be used and the search stopped. Actions that act on a particular Node, could be excluded using general Action Cost rules. Link Cost rules could be used to exclude groups of Actions.

Link creation

Links, even general Links, should be created in objects. Links are easier to create, edit and manage while they are stored in objects.

Links have been found to be easier to create if they are specific. General Links should also be contained in the objects. These general Links could be used to create more specific links. The general Links could be defined as classes.

7.9.8 Conclusion

The comments on the Links have been pessimistic at best but Links make the creation

of actions and objectives much simpler as only the Actions primary variable need be defined. By defining how the variables interact, Actions that are not suitable are automatically excluded making the it unnecessary for extensive rules to be used to discover particular actions.

The principle of using Links is important as it makes it possible to apply more general knowledge to describe a situation. General knowledge is more likely to be portable to similar situations. Work must still be done to improve the method of Link creation and management, and methods used in the search for the Links. The ideas on this topic have already been discussed.

The search for Actions and Links can be done simultaneously. Enabling a better chance for the best Action to be found.

CHAPTER 8 ACTIVATION

8.1 Introduction

Once an action has been found that will affect the root Node it is necessary to check that the path used by the action can be implemented. Delay times, costs and names must be transferred to the relevant points in the path and the action must then be implemented. This is done during the activation stage.

After a search has been completed a structure similar to the one shown in figure 8.1 could exist. After activation a structure shown in figure 8.3 should exist. Each Node will find it's parent and descendent Nodes and will have found the delay time between the lower Node changing and itself changing. Each Node will find what change to expect after the delay.

8.2 Behaviour if unsolved

If no action could be found for a Node, it is unsolved. Under such circumstances default rules are activated, and the Node is marked as unsolved and will not be used for a period of time. This will ensure that process time is not wasted on a Node that has no solution, and gives time for conditions to change in the plant. A solution may be possible under the new conditions.

8.3 Normal Activation

When a Node finds an action it returns a message FOUND to the control routine. The control routine will then call the method Activate of the root Node. The activated Node will check that it can activate itself. This entails checking if any copies of that Node exist.

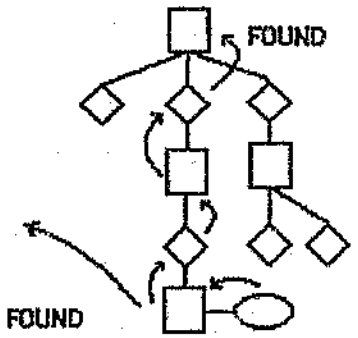
A copy of a Node is a Node that will act on the same variable. If other copies exist the chances are that the Node being activated is the most important one, otherwise it would not have been created. However each of the copies are checked and if they are less critical than the present Node they are prematurely terminated. If one of the other Nodes are more important then the Node cannot be activated and the Node fails. The behaviour under such conditions is described in section 8.5.

If the Node can be activated it will retrieve the name of the decedent Node from the Link that created that descendent. The Nodes that are activated are in the path that contains the action. The other decedent Nodes are destroyed.

The Node then retrieves the time delay between the decedent Node being solved and the current Node changing. The decedent Link contains this information. The Node will then activate the decedent Node which will in turn go through the same process.

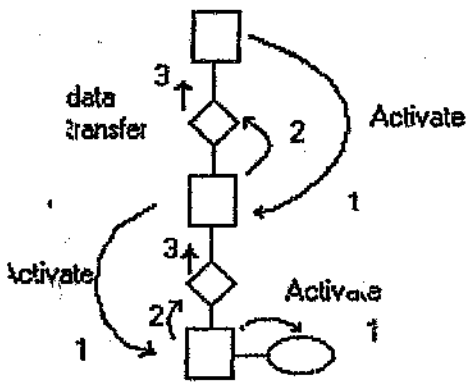
Each Node in the path will call the Activate method of the next Node in the path, until the Action is reached and activated. The last Node in the path labelled as ACTIVATED, all the others are labelled as HOLDING as they are waiting for the decedent Node to succeed. This will be described in the next chapter.

Figure 8.2 shows the activation diagrammatically.



The message FOUND is sent up the inference path and to the control function.

Figure 8.1 Inference path before activation.



1. Activation path
2. Reply to calling node
3. Data transfer from called link

Figure 8.2 Inference path during activation.

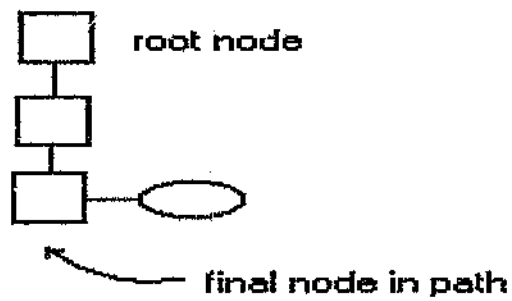


Figure 8.3 Inference path after activation.

When the action is activated it marks itself as activated, and places itself on an action stack. The actions are removed from the stack at a user specified rate and the method Action is called.

The method Action will cause a message to be sent to the operator, and rule number, action number and variable numbers to be transferred to the DCS. This has been described in chapter 6.

8.4 Information transfer

During activation information required during the management phase is gathered. This information includes the delay between the Node and it's decedent, and the expected variation. This informarion is necessary in order to determine when a Node has succeeded or failed.

Costs are gathered in order to calculate the estimated cost for the Link that created the Node. The cost of the lower part of the path is retrieved and the cost of the Node

is added. This cost is transferred to the Link that created the Node and is averaged with the existing estimate cost for the Link.

8.5 Conflict resolution

During the search for actions for the various objectives, Nodes that act on the same variable can be created. These Nodes are copies of each other and listed together in the list Copies.

As it is only possible to alter a variable in one particular way at a time i.e. only one of the copies of the Nodes acting on a particular variable can be used at a time. The most important Node, or the first Node is always activated. Any other Nodes are either terminated or they are not activated.

A Node is terminated if it has already been activated. When a Node is terminated a termination message is passed down to all the lower Nodes until the action is reached. A termination method is called in the action which will allow the action to be terminated properly. Each Node then destroys itself.

This behaviour ensures that the most important action under all circumstances is used. A typical situation is when an action has been taken to fill a tank. Should that tank become overfull, it will be necessary to reduce that tank level. The path waiting for the level to be increased will be terminated as the path to decrease the level is activated.

8.6 Conclusion

Activation is a procedure where the Action found to cause a change to a Node is implemented. It forms the transition between the search and management stages.

Message passing necessary for the implementation of this section proved to be confusing but was found to work well. Activation is a necessary stage in the control cycle.

CHAPTER 9 MANAGEMENT

9.1 Introduction

Once an action has been implemented, the inference path used to find that action is not destroyed. Instead a management phase is entered. The nanagement phase determines when to destroy the path.

It is important to ensure that on successive cycles of the expert system, conditions that have already been responded to, do not reactivate the search procedure. This is done by not erasing the inference path that was created from the condition.

When a condition is detected an objective is created. If a control rule tries to create an objective and the objective already exists which describes a similar situation, the rule assumes that it has already been attended to and does nothing.

If the original condition exists and the Node is destroyed, the Node will be recreated and another search for a response implemented. Depending on the action, it may be found again or another action could be found in response to the condition.

Because the inference path exists, it is necessary to determine when to erase it.

The inference path is made up of individual Nodes. Each Node governs itself but can alter the behaviour of the complete path. The behaviour depends whether the Node has succeed or failed to have its objective met.

9.2 Node behaviour

Each Node in the path is cleared individually. To determine when to erase each Node there are two possibilities:

- Check when the objectives described in each Node are physically realized. This requires watching the variables described in the objective. If they changed by the amount specified, within the available time, the initial condition is assumed to have been attended to, and the Node succeeds.

- It was found that it is not always possible to determine if the objective variable has changed. This is due to a lack of instrumentation or noise in the plant.

  Also, by solving the objective does not necessary mean the original condition not longer exists. Feed forward control is a typical example.

  In such circumstances the Node remains active for a particular amount of time. It is assumed that after this period the condition should have been cleared and if it has not, it must be reacted to again.

9.3 Conditions for success

The Nodes are cleared after they succeed or fail. The behaviour of the rest of the inference path depends on the Node.

As already mentioned there are two possible management systems for each Node.

- **Waiting for a variable change:** On each cycle of the expert system, the variable change described in the activated Node is compared to what is actually happening in the plant. If the variable change is within more than 75% of the predicted value, the Action or Link is assumed to have worked. 75% is used as it is recognized that the variable change may not occur as predicted.

- **Waiting for a specified time:** This management scheme is only applied if the Node is the final Node in a path. If a Node is the final Node, it will wait for a specified time, and then succeed. Thus if a condition exists that uses the Node as a response, it will be blocked for that time period.

9.4 Conditions for failure

Node failure depends on two conditions

- The Node has **existed longer than the maximum time** available. This does not apply if the Node is the final Node and managed by the second management scheme.

- The Node is expecting a variable change i.e ACTIVATED but the **delay time** between either the implemented action, or the secondary objective, and the Node has been **exceeded.**

The description below describes the process of clearing a particular inference path depending on the success or failure of the Nodes.

9.5 Conditions for clearing the path

The inference path is collapsed slowly as long as each Node in the path succeeds. If any of the Nodes in the path fail, it is assumed that the objective required by the initial condition will not be met. In this case the complete path must be destroyed, enabling a new action to be found.

Figure 9.1 indicates the various parts of the inference path and is used extensively for the description of the path management.

Let us consider the Node at the end of the path. This Node is the Node that found the implemented action, or its secondary Node has already been met. In figure 9.1 a it is Node D. In figure 9.1 b it is Node C.

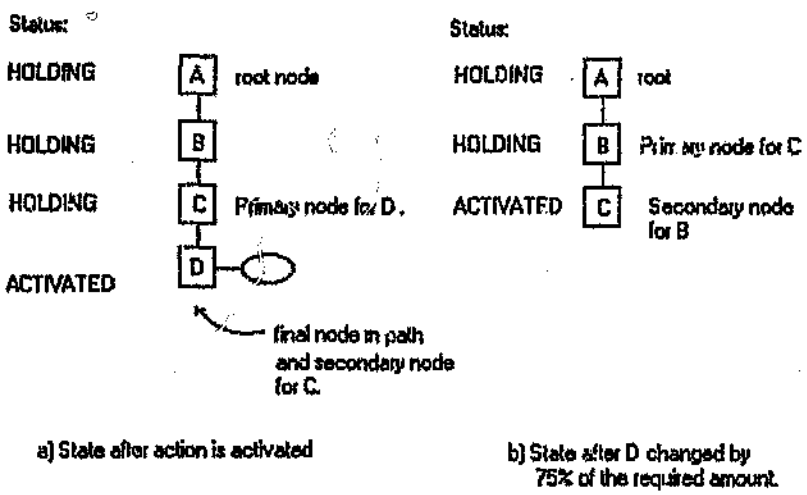


Figure 9.1 Management of an inference path.

Consider the Node that found the action. The action has been activated and the operator has carried out the instructions in the action message. The Node knows how soon after the operator has read the message that it should expect a change in the plant.

Depending on the Nodes management, this Node will watch the variable it defines for the time period defined by this delay. If it has not changed with in the specified time the Node fails. If the variable changes by 75% of the objective it is considered to have succeeded. Or the Node could wait a predetermined time before it succeeds without any regard for the process variables.

When the Node succeeds it will inform its primary Node, Node C in figure 9.1a. Node C will then become the final Node as shown in figure 9.1b.

If the final Node, Node D fails, the complete path is considered to have failed. When a path fails it is destroyed, making it possible to find another response to the initial condition if it still exists.

Consider Node C in figure 9.2 a after Node D succeeds. Node C knows that the action has worked and has thus far altered the variables below it in the path. The Node C knows from information obtained in the Links that after the secondary objective, Node D, is met there is a time delay before the variable in Node C will be seen to change.

Depending on the management of the Node C, it will either:

- Wait for the delay time period. If the variable has changed within the this time

period, it will succeed. If the Node does not change within this time period it will fail.

- Wait a predetermined time and succeed regardless of what the plant does.

If Node C succeeds it will end and inform its primary Node of its success, in this case Node B. If the Node fails the complete path will be erased.

After Node C succeeds, it will be destroyed causing Node B to be the final Node. This will continue until the root Node succeeds or fails. If the initial condition still exists after the root has been destroyed the inference cycle will start again. Depending on the nature of the actions, another action could be found.

The other Nodes in the path are also watched. Should one of the other Nodes succeed, such as Node B is figure 9.1a, the Nodes below it will be erased and Node B will succeed causing Node A to be the final Node. Note that a middle Node can only succeed if it detects a variable change it is waiting for.

All the Nodes contain a time value that they should ultimately have changed by. The action should cause the change before this time value, but if this ultimate time value is exceeded the complete path will terminate.

9.6 Learning

Because variables are checked to ensure they do change, and because of the presence of a costing system, it is easy to implement a learning mechanism.

Whenever a Node succeeds or fails, it will update the Action it called or the Link it used. During the update, the percentage successes and percentage failures of each action or link will be altered.

9.7 Conclusion

The management phase of the expert system ensures that the system does not repeatedly find a solution that has already been attended to. It determines when a control action has worked and when a condition should be reacted to again.

Each Node in the path is checked to ensure that it has indeed changed as it predicted. This is done so that the system is able to determine if an action was actually implemented by the operator, and if it failed, to enable a better action to be searched for.

A problem that could be experienced with this stage occurs when an action is found to solve a particular inference path. The system does not check that the final Action will alter the root variable by the required amount, it only checks to ensure that it will meet the final Nodes objective, which should in turn meet the objective in the root Node. However each Link and the final Action in the inference path may not change the necessary variables exactly as required. This could eventually lead to large errors in the required change to the final variable . This problem could be solved using the secondary relation in the Links and using this either during Activation or during the Management phase to propagating an expected variation value back up the path. It can also be circumvented using Action Cost rules.

It was found that it is not adequate for the management phase to only watch the

variables in each Node to check that they change as specified. Conditions occur where:

- The variable being considered is not measured.
- The variable value cannot be deduced using Value rules.
- The variable change is obscured by noise.
- Feed forward control is used where the initial condition could still occur after a response has been implemented.
- The variable does not change as defined by the Node i.e. The inference path used to find the correct action is not accurate.

In these conditions a second management scheme was used, where the Node stayed active for a specified period of time.

Using these two management schemes ensured searches carried out as a result of a condition are only repeated when necessary. The management also provides a mechanism to enable the system to learn.

CHAPTER 10 SYSTEM EVALUATION

10.1 Introduction

In order to test the system, various control tests were carried out. The majority of the tests were done by simulating conditions that were known to occur in the plant. Instead of reading instrument value from the process, the values are fed directly into the system.

In order to test the complete system, the expert system was applied to a section of plant. The scenario that the system was applied to will be described. The procedure for entering knowledge will be presented and results of the testing will be then be discussed.

10.2 Scenario

In the process being controlled, a tank feeds a solution into a mixing tank TKF. A calcine slurry, called stream P41, is added to this solution. The ratio is altered to keep the pH constant, at 3.5. The amount of calcine added is indirectly altered by altering the speed of two screws. A variable speed drive is connected to the two screws, but only one screw can be on at a time. The pitch of the two screws are different. Figure 10.1 shows the plant process diagram.

The pH of the control system is altered by regulating the flow out of storage tank. This is done by a dedicated control loop. The controller controlling this loop is fast.

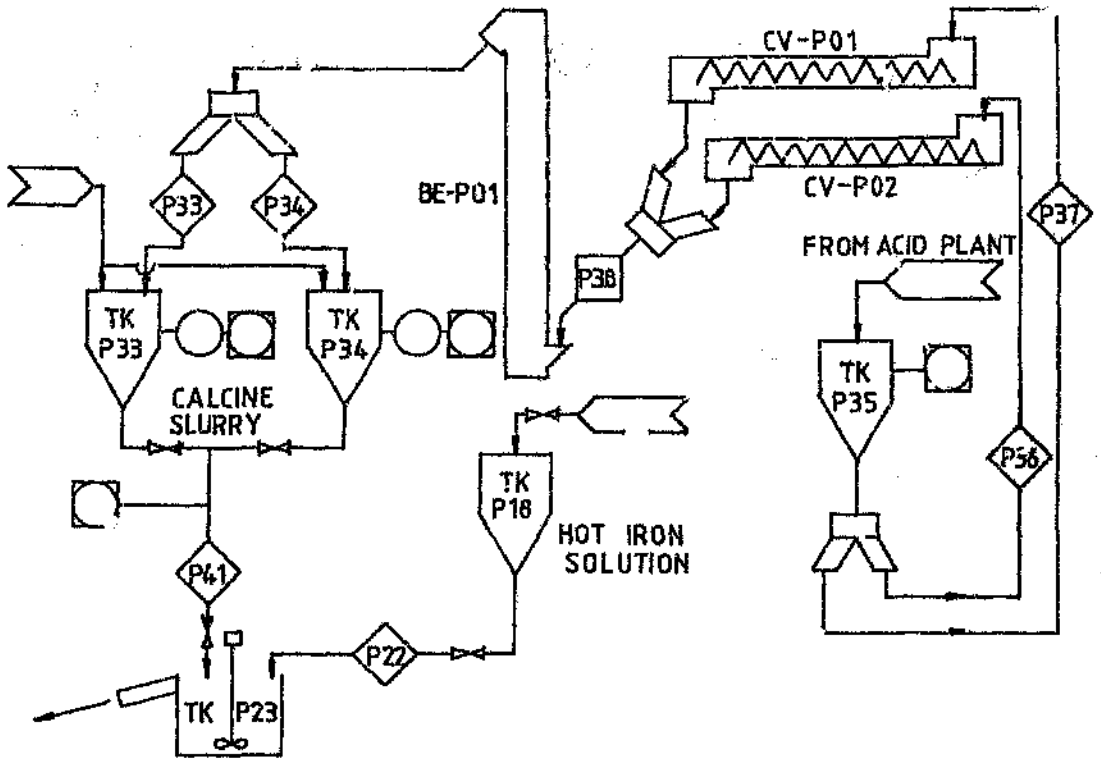


Figure 10.1 P&ID of process

The objective of the control was to keep the rate of change of the tank level of TKP18 below 2.5%/hr while the tank level is between 40 - 60%. The rate of change will be limited to -2.5%/hr - 0%/hr if over 60% full, and 0%/hr - 2.5%/hr if below 40%.

The control will be accomplished by altering the calcine screw speed as this will increase the ... in the tank. To compensate, the flow from the storage tank is increased, thus altering the tank level.

The control action must alter the screw speed until the limit is reached, and then switch over to the next screw.

10.3 Knowledge implementation

From figure 10.1 the following Items are created: TK P18, P22, P41, P38, P36, P37, CVP01, CVP02. The appropriate variables a a d for each. Note that the items are referred to by their tag names that are used n process and instrumentation diagrams.

To revise, the items describe the physical entities i the plant and provide access to the variable values. Figure 10.2 shows the hierarch cal structure used to describe the plant.

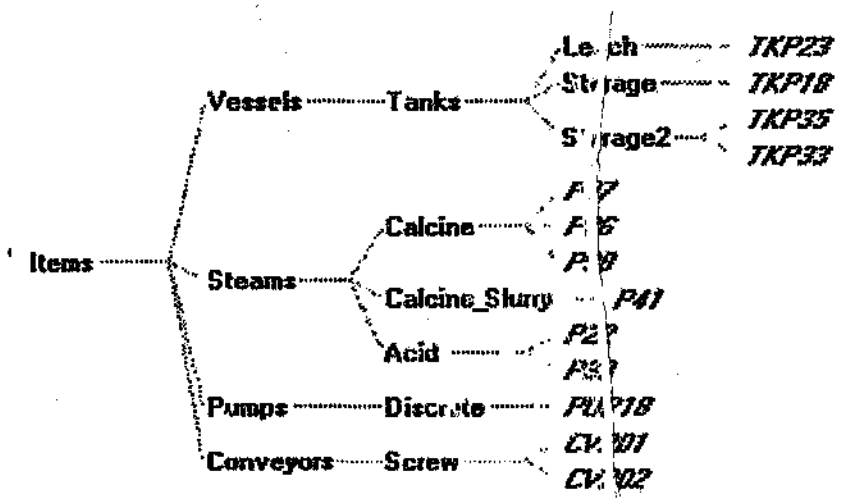


Figure 10.2 Structure used to describe process to be controlled

Conditions that the plant must respond to are defined. These are in Control rules. The rules are shown in Appendix C. The control rules all start with the prefix C_.

Actions were defined. The actions indicate the possible control actions that can be taken. Some of the control action messages are shown below:

Actions on CVP01:

- Increase the drive frequency of CVP01 by 5Hz.
- Decrease the drive frequency of CVP01 by 5Hz.
- Turn off CVP01.
- Turn on CVP01.
- Switch from CVP02 to CVP01.

Actions on CVP02:

- Increase the drive frequency of CVP02 by 5Hz.
- Decrease the drive frequency of CVP02 by 5Hz.
- Turn off CVP02.
- Turn on CVP02.
- Switch from CVP01 to CVP02.

Link rules are then created. The Link rules will link the necessary variables. Because it is not possible to measure the flow rates of the streams P36, P37 and P38 Value Rules are used to generate the values from the calcine screw rates. Appendix C shows some Link rules and Value rules.

Because only one screw is on at a time, it is necessary to direct the inference path to the crew that is on. Link Cost rules were used for this purpose. Further rules were to handle the situation where crossover was required.

Action Cost rules were used to ensure the correct action was used under the necessary circumstances. A typical example is to use the action to switch from one screw to the other. These rules are also shown in Appendix C.

10.4 Evaluation of the test

The system was implemented and executed. Initial simulations were run in order to debug the knowledge. From these simulations it was found that the cost values used in the Actions, Action Cost rules and Link Cost rules had to be altered to ensure the system worked correctly.

The system worked. It would determine the correct action when required. The screw frequency would be adjusted in 5Hz increments or more, depending on the amount the tank level had to be altered by. The screw was switched from one to the other as required.

The inference path used seven Nodes in the path. The Link Cost rules would direct the choice of path to be taken.

Figure 10.3 shows a typical inference path generated during the execution of the example.

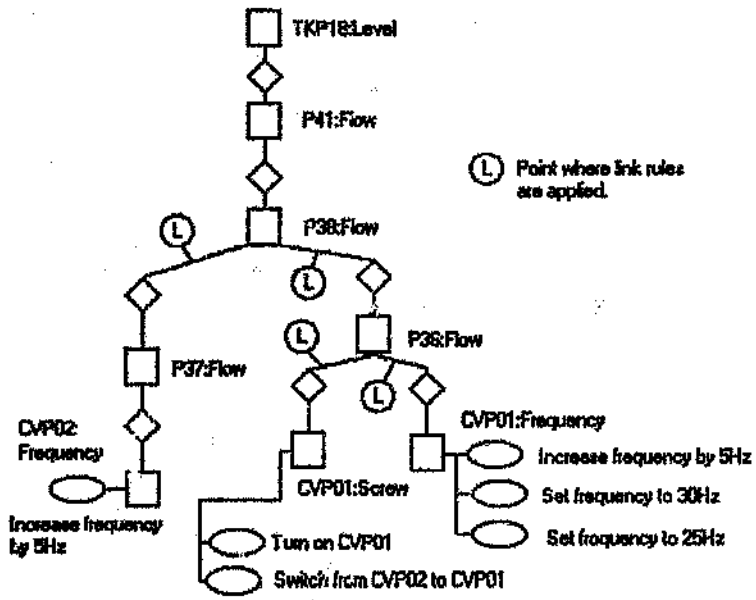


Figure 10.3 Typical inference path in the test example.

10.4.1 Expected plant response

From the simulations, it was ascertained that the system would behave as expected. To ensure that the behaviour was adequate for process control, the system was connected to the process. The requests made by the program were carried out and the system would wait for the expected response from the process variables. Problems were experienced at this point.

The problems related more to the control action than the behaviour of the expert system. The plant did not respond to the action carried out on it. If the change occurred, it was not reflected in the variables being monitored. This could be due to high noise levels.

Simulations of the variables being changed showed that the program did indeed work. To implement the control system, the management method used for the Nodes was altered. Instead of the Node waiting for a specific variable change, the Nodes would exist for a predetermined time and then succeed regardless of the plants condition. This constituted "open loop" control.

10.5 Conclusion

Even though the system responded correctly under simulation, and when placed on the process plant it responded with the correct control action for the existing condition, it was not possible to observe the variable change in the process.

The system recognizes conditions in the process. It creates objectives to alter the objective. In this case the objective was to alter a tank level. The inference process would be carried out correctly. In this scenario, to alter the tank level, flow P38 would have to be altered. The system could then try to alter the flow of streams P36 or P37. Link Cost rules would direct the process to the correct Link. To alter the flow rate the calcine screw could either be started or the frequency changed. Again Link Cost rules were applied to direct the decision process. Once ascertaining the variable to change, the most suitable control action would be found.

Once the system had found the correct action it would then watch the variables to ensure that the change occurred. If a simulation was used, the change on the plant was mimicked, the system would succeed, and the appropriate learning take place. On the plant, the change was not always seen and the process would fail. The search would be carried out again.

The example showed that the knowledge structures are adequate to represent the required knowledge to describe the control of a process.

The system is able to recognize conditions in the process. These conditions propose objectives, that are used to determine a suitable control action. During the search for a control action, actions that do not case a suitable change to the process are ignored.

The management of the decisions was shown to be adequate under simulated conditions. But issues such as variable noise, and poor process knowledge made it difficult to complete the control loop. The system was easily adapted to an "open loop" system where the Node did not wait for suitable variable changes by using the alternative management strategy.

The test indicated the complex nature of the knowledge to be entered and the need for adequate process study to gather the necessary information for the control knowledge.

The test also indicated the need to be able to deal with inaccurate and noisy measurement of variables in the control loop, but it is felt that these issues are outside the scope of this project to be investigated further.

CHAPTER 11 CONCLUSION

11.1 General

A real-time knowledge base for controlling an industrial process based on a object orientated techniques has been designed. The inference engine is defined in terms of functions and classes of the objects. Knowledge is stored in the instances of the classes. This interaction keeps the inference engine separate from the knowledge but allows a simple inter-connection of the two. Rules are used to contain knowledge that is not suitable for the objects.

Rules are used to recognize conditions in the process. Conditions are variable patterns that occur in the process. These conditions are used to determine if any change needs to be made to the process. If a change is required the rule creates an objective describing that change.

Once objectives have been defined, a search is made for a control action to cause the required change to the plant. To find the control action, knowledge describing the interactions of the various plant variables is required. The knowledge describing these variable interactions is contained in rules. These rules create a class called Links, which contain the specific deductions from the rules.

The knowledge in the Links describes how to change a primary variable in order to obtain a desired response in the secondary variable. The knowledge must also define the expected time lag between changing the primary variable and seeing the secondary variable change.

During the process of finding a control action, an inference path is created. The path contains information describing variable changes that should occur in the process when the action works. This information is used to evaluate the action.

The plant is watched to ensure that the action does change the necessary variables as specified. This is done to determine if an action worked. If the action failed, a more suitable action will be sought. The success rate of an action can be used to influence the use of the action in the future.

11.2 Assessment of the prototype

11.2.1 Conditions and objectives

The use of objectives instead of directly defining control actions, allowed the control rules to remain simple. Fewer rules are required to find a suitable action, as it is possible to determine if an action can meet the desired objective from the information defined by the objective. The creation of objectives makes it easier to adapt the system to plant changes. By altering an objective, the control actions found could be totally different but the change to the knowledge base would be minimal.

The condition recognition of the system was found to be adequate although the control rules could be made easier to create by improving the use of the consequent. This could be achieved by defining conditions in an object class instead of containing all the information in the consequent is as is done at present.

Conflicts occur when rules create different objectives that act on the same variables. Conflict resolution determines what to do in these situations. The conflict resolution

works well. A variable can only be altered in one way, thus only one objective on a variable can be activated.

It is often necessary to consider altering a particular variable in more than one way. To allow this copies of objectives are made. A copy of an objective is an objective that acts on the same variable, but can specify a different variable change. The use of copies is important in the search procedure. The copy mechanism was found to be necessary and in future work should be expanded to allow for copies describing different possible changes to a variable within the same inference path.

11.2.2 Actions

Creation of actions proved to be simple although somewhat tedious. The actions are well defined, and are easier to create when only the primary variable is considered.

The search for actions is carried out before the search for Links. In some cases this lead to sub-optimal actions being found. The search for Actions and Links should be combined in future work.

11.2.3 Items

Items were found to be successful technique for are storing information describing the physical layout of the process being controlled. The information required to create the items is directly available on process and instrumentation diagrams.

11.2.4 Links

The information required is important, as it simplifies the description of the control

actions.

The knowledge required for the Links is not easy to define. Detailed process study is often required. Links do not have to be precise but there are repercussions in the management phase if this information contains inaccuracies.

The information describing time lags is equally important, as it describes if an action can meet the required objectives with in the available time period.

It was found that the mechanism for creating and describing the Links used in the prototype system is not adequate, and a more versatile method should be devised. In this system expressions were used to describe the interactions of variables. It would be better to use rules, fixed values as well as expressions for this purpose. Rules acting on the Link object class would be particularly powerful as they would allow for the Links to be adapted to the current plant situation.

Methods for excluding Links, and for manipulating the choice of Links should also be improved. This includes the use of initial costs, and further use of Link Cost rules.

11.2.5 Rules

Various rule groups have been used. These are:

- Control rules
- Link rules
- Value rules
- Default rules
- Action Cost rules

- Link Cost rules

These rule groups each define a different part of the knowledge base and are each applied at specific points in the inference procedure.

In the present implementation, the Link and Action Cost rules are called from the Link or Action that they could influence and must refer to values in the Action or Link. It would be better if rules were called when variables in the plant change, or when conditions are activated. Values could then be written directly to the appropriate Action or Link. Thus these objects will have an indication of how suitable they are, before they are used.

11.2.6 Management

The management stage determines when to terminate the inference path. Two management schemes were evaluated as discussed in chapter 9:

During the inference path creation the required variational values are passed down the path during Link expansion. Because variation values are not propagated back up the path after an action has been found, errors in the variation expected by each Node could accumulate. Minor problems were experienced in this regard.

11.2.7 Knowledge requirements

The knowledge the framework requires has been well defined. The objective is to make the knowledge easier to create and manage.

The amount of detail of knowledge required to implement the system was found to

be surprisingly large. The knowledge in the actions is acceptable, as is the knowledge in the Control rules. However the knowledge in the Links is too detailed and difficult to obtain in a typical metallurgical industry for practical implementation of the present system to be feasible.

11.3 Future work

Several shortcomings were discovered during the development of and the experimentation with the system. This section describes suggestions for future development. Some possible improvements stem from practical limitations of the development platform. Others from awkwardness of the proposed system.

The design of a new system should:

1) Be timer based. This will solve some practical problems with the shell. It will also ensure that decisions can be made timeously. No problems were ever experienced with slow decisions, but they could occur.

2) Provide a class for condition definitions. Instead of Control rules containing a condition message and objective, provide this in a class of conditions. The advantages are :

   - Easier management. Objectives can be destroyed when they are met. Conditions will ensure rules are not fired again.
   - The Conditions can be used as an alarm system. Objectives need not be proposed with every condition. The system could be used as an intelligent alarm system. As the need arises, conditions can be expanded

into objectives.

- The control rule's Consequent would be simpler. Various rules can be used to defir an objective and the control rules would be easier to debug.
- Complex decisions processes are possible with the use of a Condition class.

3) Link Cost rules and Action Cost rules applied with Control rules. At present these rules are called when they are required. The rules would be easier to write if they refer directly to the plant variables as the Control rules do and are applied when the Control rules are applied.

4) Value cost rules should be introduced. Link cost rules provide two functions. They indicate which Link should be used, i.e., how the variable should be altered, and also watch the proposed variable change and indicate if the change is suitable for the present plant conditions. Value rules that are activated during the Node search should take over this second function.

5) Link rules used to create Links, should not be used as they are difficult to create and manage even though they provide a mechanism to use general knowledge. Specific Links would have to be built to contain the information in the Link rules. It has been found that in many cases the knowledge required by the Links had to be fairly specific as each situation in a plant has its own peculiarities. The Link rules tended to be specific and are not the best way to represent the knowledge.

A knowledge system such as predicate logic could be much more suitable for creating Links. If this is the case the search for new Links should occur before

existing ones are used unlike the present system where Links are created if no existing Links can be found. This is necessary to enable new knowledge to be used in the system.

6) Objectives are associated with Items at present. Instead the management of the objectives should be directed through the Variables class. Objectives would request the variable that it is to alter for the change, instead of making the request through the Item class. This would make the management of conflicts and copies easier.

7) Search for Links and Actions simultaneously. This was discussed in chapter 7.

REFERENCES

1. Hart A., **Knowledge acquisition for expert systems**, McGraw-Hill book company, 1986, pp.20-22.
2. Groustch J., Plant Supervisor, Zincor, South Africa.
3. Kang A.M., **A Real-time expert system shell for process control**, M.Sc. Thesis, University of the Witwatersrand, Johannesburg, 1990.
4. Lalka C.J., Weber R. **Real-time verse static shells for real-time expert systems**, Exxon Chemical Company.
5. Lun V., Macleod I.M., Hanarahan H E. **An introduction to expert systems**, Course Notes, Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1988.
6. Lun V., Macleod I.M., **Towards distributed real-time intelligence**, Department of Electrical Engineering, Unversity of the Witwatersrand, Johannesburg.
7. Rich E., **Artificial Intelligence**, McGraw-Hill book company, 1983, pp.1-30,pp.229-233.
8. Schildt H., **Artifical Intelligence using C**, McGraw-Hill book company, 1987, pp.1-117, pp.267-315.
9. Waterman D.A., **A Guide to Expert Systems**, Addison-Wesely Publishing Company, 1986, chapters 1-4, 7,9.
10. **Kappa User Manual**, Intellicorp, 1992.
11. **Kappa Reference Manual**, Intellicorp, 1992.

GLOSSARY

Action: A class that describes the control action that is to be executed on the plant

Activation: The process where an inference path is prepared for the management phase.

Critical Time: A time value in a Node indicating time that is available for the action to effect the Node.

Expert System: A form of artifical intelligence that mimics the behaviour of a plant operator.

Forward Chaining: A chaining technique where rules are checked to ensure that the antecedent is true. If it is the consequent is applied.

Link: A class of knowledge that defines the interactions between variables in the plant.

Link rule: A rule used to create Links.

Hold Time: The time that a Node should stay active for, after the descendent Node has succeeded.

Inference engine: A mechanism that takes knowledge in a particular format, and uses this knowledge to make deductions.

Inference path: A path made up of Links and Nodes, that describes a sequence of variable changes.

Item: A class of knowledge that describe the items of equipment (real or abstract) that make up a process plant.

Knowledge base: A collection of knowledge. This knowledge is usually in a well defined format.

Knowledge base system: A system that contains knowledge. Because of this knowledge the system is able to appear to be intelligent.

Node: A class of knowledge that defines the objectives of some control action.

Objective: A description of a variable change to the process. It defines the

amount the variable is to be altered by and the time available for the change.

Primary node: The node that is searching for a link.

Primary variable: The variable in the secondary node.

Priority: A number between 0 and 10 indicating the relative importance of various pieces of knowledge.

Ro The start of an inference path.

Root node: The node created by the control rule.

Secondary node: The node that if solved, will eventually change the primary node.

Secondary variable: The variable in the primary node.

Shell: A basis on which a expert system can be created.

Variable: A value describing a property of the process.

Variation: The amount a variable is to be altered by.

APPENDIX A BACKGROUND TO PROJECT ENVIRONMENT

A.1 Introduction

The Knowledge based system is to be implemented in a section of plant at the Zinc Corporation of S.A. This plant produces zinc from a zinc oxide ore. The section of interest forms part of the leaching process where zinc is leached from a slurry under high temperature and acid levels. The system is being developed as a prototype to show that expert systems can work in process control, and is restricted to the Residue Treatment Section.

The project has been restricted in order to reduce complexity and size of the prototype and because of the availability of process information. Unlike the rest of the plant, a Distributed Control System (DCS) is being used to monitor the Residue Treatment Section and thus information required by the knowledge base can be directly retrieved from this DCS.

An important aspect to any reasoning system is its reaction to various inputs. A systems intelligence is measured by its reaction to its environment or its input. Before any intelligent system can react in a suitable manner to its environment, it must be able to determine what state that environment is in. A highly "Intelligent" system may be used to control a particular environment but if it is not supplied with enough knowledge about the environment that it is controlling its reactions will appear to be unsuitable.

A.2 Interface to a DCS

The expert system will use the data obtained from the distributed control system

to drive itself. The DCS will be linked by a serial link to the expert system which will be running on an IBM compatible PC.

The operators will be using the DCS to monitor, and eventually control the plant. The only interface for the expert system is via the DCS. The DCS provides information describing the plant, and messages to the operators are displayed by the DCS.

As the DCS does not provide a dynamic interface, all the possible messages must be pre-configured. The expert system must find the suitable message for the condition in the plant.

A.3 Plant experts

During the development of any expert system it is critical to be able to identify the sources of expert knowledge. In real-time expert systems this information is available from plant operators, plant foreman and metallurgists.

A.4 Hardware and software environments

A.4.1 Shells

The dissertation describes a proposed structure for a real-time expert system. A critical part of the project is to find a platform on which to build this structure. A static expert shell was found for this purpose. The shell provided an object-orientated programming environment and forward and backward chaining mechanisms. The shell is more suitable for interactive expert system building but it was felt that a real-time expert system could be created on it.

The proposed structure of the expert system would attempt to provide a constrained environment in which knowledge about a process could be placed that would be easy to create and manage, but still provide suitable control of the process. The framework would also control the decision process and manage decisions taken by the system.

The dissertation describes a framework that is created using the tools provided by the shell. The shell itself is not suitable for real-time process control.

A.4.2 Hardware Platform

The hardware platform is defined by the shell being used, as well as speed requirements of the system. Cost is a major factor.

The structure being proposed would form a prototype to investigate some of the issues involved in creating an expert system. The speed performance did not need to be very fast, especially as the process being controlled was fairly slow. The fastest decision would have to be made in five minutes.

For these reasons a PC based system was adequate.

APPENDIX B PRINT OUT OF A TYPICAL EXPLANATION

This explanation was generated during test work described in chapter 10. The conditions during the test:

- Rate of change of TKP18:Level = 4.72
- CVP01:Screwcon = OFF
- CVP02:Screwcon = ON
- CVP01 & CVP02:Frequency = 45Hz

The explanation describes the process of finding an action to switch over from a slow screw conveyor to a faster screw conveyor.

12:55:57 AM 01/23/93

Initializing HAL Expert

12:56:38 AM 01/23/93
Rule C_DLevelHi has initiated a change to Flow of P41
by 2.38. The original value is 51.50 units.

---------- Working on NP41_Flow ----------
Action not available.
- Considering action Unchoke to change the Flow of P41.
The delay cost is 0.200 cost of the action is 1.000.
- The Flow of P41 may be changed by changing the Flow of
P33 by 2.38 . Original value: 29.00. Link: Flow_1 .
Node 0.000, Estimated 0.000 , Var 0.000, Delay 0.007,
Fail 0.000, Rules 0.000, Total Cost 0.007
- The Flow of P41 may be changed by changing the Flow of
P38 by 2.38 . Original value: 22.50. Link: Flow_2 .
Node 0.000, Estimated -0.653 , Var 0.000, Delay 0.007,
Fail -0.635, Rules 0.000, Total Cost -0.861
- The Flow of P38 may be changed by changing the Flow of
P36 by 2.38 . Original value: 0.00. Link: Flow_3 .
Node 0.007, Estimated -0.305 , Var 0.000, Delay 0.007,

Fail -0.004, Rules -1.000, Total Cost -0.991
- The Flow of P38 may be changed by changing the Flow of P37 by 2.38 . Original value: 22.50. Link: Flow_4 . Node 0.007, Estimated 0.155 , Var 0.000, Delay 0.041, Fail -0.004, Rules 0.111, Total Cost 0.280
- The Flow of P38 may be changed by changing the Flow of P37 by 12.62 . Original value: 22.50. Link: Flow_8 . Node 0.007, Estimated 0.423 , Var 0.000, Delay 0.041, Fail 0.000, Rules 0.111, Total Cost 0.512
- The Flow of P36 may be changed by changing the Frequency of CVP01 by 2.38 . Original value: 45.00. Link: Flow_5 . Node -0.973, Estimated 0.373 , Var 0.000, Delay 0.041, Fail 0.540, Rules 0.000, Total Cost -0.108
- The Flow of P36 may be changed by changing the ScrewCon of CVP01 by 1.00 . Original value: 0.00. Link: Flow_6 . Node -0.973, Estimated 0.347 , Var 0.000, Delay 0.041, Fail 0.004, Rules -1.000, Total Cost -0.958
Rule: 0.000, Var: 0.000, Fail: 0.000, Total Cost 0.000
Requested Variation 1.00
- Considering action CVP01On to change the ScrewCon of CVP01.
The delay cost is 0.432 cost of the action is 0.432.
Action not available.
- Considering action CVP01Off to change the ScrewCon of CVP01.
The delay cost is 0.432 cost of the action is 1.000.
Rule: -0.200, Var: 0.000, Fail: 0.000, Total Cost -0.200
Requested Variation 1.00
- Considering action CVP01Switch to change the ScrewCon of CVP01.
The delay cost is 0.432 cost of the action is 0.193.
12:57:41 AM 01/23/93
- The action CVP01Switch was found, which will cause the ScrewCon of CVP01 to change by 1.00. The original value is 0.00 units.

The critical time is 1390.00
Node NCVP01_ScrewCon to be solved after 600.00 sec
Critical time for NP36_Flow is 1460.00
Critical time for NP38_Flow is 1480.00
Critical time for NP41_Flow is 1500.00

12:57:55 AM 01/23/93 ACTION: CVP01Switch
The rate of change of TKP18 is over 0%/hr
Please switch over to CVP01. Set the frequency to 15Hz slower than it is now.

APPENDIX C RULES USED IN FINAL EXAMPLE

Appendix C shows a list of the rules that are used in the example described in chapter 10.

The rules are created with the function MakeRule. The syntax for MakeRule is:

MakeRule(Rule Name, General variables, if expression, then expression);

LINK RULES

```
    /*************************************
     **** RULE: L_P37Screw1On
     *************************************/
MakeRule( L_P37Screw1On, [Node|Nodes],
   Node:Item #= P37 And Node:Variable #= Flow,
   List( Node:Item, CVP02, Node:Variable, ScrewCon, 60, v, 1,
      0, 20, FALSE, LP36Screw1On ) );

    /*************************************
     **** RULE: L_P37Screw1Off
     *************************************/
MakeRule( L_P37Screw1Off, [Node|Nodes],
   Node:Item #= P37 And Node:Variable #= Flow,
   List( Node:Item, CVP02, Node:Variable, ScrewCon, 60,
      -1 * P37:Flow, 0, -100, 0, FALSE, LP36Screw1On ) );

    /*************************************
     **** RULE: L_P36Screw1Off
     *************************************/
MakeRule( L_P36Screw1Off, [Node|Nodes],
   Node:Item #= P36 And Node:Variable #= Flow,
   List( Node:Item, CVP01, Node:Variable, ScrewCon, 60, " -1 * sx",
      0, -100, 0, FALSE, LP36Screw1On ) );

    /*************************************
     **** RULE: L_LevelDLevel_1
     *************************************/
MakeRule( L_LevelDLevel_1, [Node|Nodes],
   Node:Item #= TKP18 And Node:Variable #= Level,
   List( Node:Item, TKP18, Node:Variable, DLevel, "Max( 60, crt * 0.25- 1800 )",
      v, "-1 * TKP18:DLevel + v /( t+1800)*60", -100, 100, FALSE,
      LLevelDlevel1 ) );

    /*************************************
     **** RULE: L_DlevelFlow1
     *************************************/
```

```
MakeRule( L_DlevelFlow1, [Node|Nodes],
   Node:Item #= TKP18 And Node:Variable #= DLevel,
   {
   EnumList( Node:Item:Inputs, PI, List( Node:Item, PI, Node:Variable,
                                 Flow, 2400, v, "v ", -100,
                                 100, FASLE, LDlevelFlow1 ) );
   EnumList( Node:Item:Outputs, PI, List( Node:Item, PI, Node:Variable,
                                 Flow, 2400, v, "-1* v",
                                 -100, 100, FALSE, LDLevelFlow1 ) );
   } );

      /*************************************
       **** RULE: L_pHP221
       *************************************/
MakeRule( L_pHP221, [Node|Nodes],
   Node:Item #= TKP23 And Node:Variable #= pH,
   List( Node:Item, P22, Node:Variable, Flow, crt, v, "t/60*v*(TKP18:Acid -
TKP23:Acid)/(TKP23:Acid*((P22:Flow+P41:Flow)+58-TKP18:Acid*P22:Flow)",
      -100, 100, FALSE, LpHP221 ) );

      /*************************************
       **** RULE: L_pHP411
       *************************************/
MakeRule( L_pHP411, [Node|Nodes],
   Node:Item #= TKP23 And Node:Variable #= pH,
   List( Node:Item, P41, Node:Variable, Flow, crt, v, "t/60*v/(P41:Flow)",
      -100, 100, FALSE, LpHP241 ) );

      /*************************************
       **** RULE: L_P41P33Flow
       *************************************/
MakeRule( L_P41P33Flow, [Node|Nodes],
   Node:Item #= P41 And Node:Variable #= Flow,
   List( Node:Item, P33, Node:Variable, Flow, 10, v, v, -100,
      100, FALSE, LLP41P33Flow ) );

      /*************************************
       **** RULE: L_P41P38Flow
       *************************************/
MakeRule( L_P41P38Flow, [Node|Nodes],
   Node:Item #= P41 And Node:Variable #= Flow,
   List( Node:Item, P38, Node:Variable, Flow, 10, v, v, -100,
      100, FALSE, LP41P38Flow ) );

      /*************************************
       **** RULE: L_P38P36Flow
       *************************************/
MakeRule( L_P38P36Flow, [Node|Nodes],
   Node:Item #= P38 And Node:Variable #= Flow,
```

```
    List( Node:Item, P36, Flow, Flow, 10, v, v, -100, 100, FALSE,
       LP38P36Flow ) );

      /*************************************
       **** RULE: L_P38P37Flow
       *************************************/
MakeRule( L_P38P37Flow, [Node|Nodes],
    Node:Item #= P38 And Node:Variable #= Flow,
    {
    List( Node:Item, P37, Flow, Flow, 60, v, v, -20, 20, FALSE,
       LP38P37Flow1 );
    List( Node:Item, P37, Flow, Flow, 60, v, "15 - v", -20, 20, FALSE,
       LP38P37Flow );
    } );

      /*************************************
       **** RULE: L_P36Screw1
       ************************************ /********/
MakeRule( L_P36Screw1, [Node|Nodes],
    Node:Item #= P36 And Node:Variable #= Flow,
    List( Node:Item, CVP01, Node:Variable, Frequency, 60, v, v,
       -100, 100, FALSE, LP36Screw1 ) );

      /*************************************
       **** RULE: L_P37Screw2
       *************************************/
MakeRule( L_P37Screw2, [Node|Nodes],
    Node:Item #= P37 And Node:Variable #= Flow,
    List( Node:Item, CVP02, Node:Variable, Frequency, 60, v, "2 * v",
       -100, 100, FALSE, LP37Screw2 ) );

    /*************************************
       **** RULE: L_SGP33Flow
       *************************************/
MakeRule( L_SGP33Flow, [Node|Nodes],
    Node:Item #= TKP33 And Node:Variable #= SG,
    List( Node:Item, P33, Node:Variable, Flow, crt * 0.5, v, sx*t*v/18/60,
       -100, 100, FALSE, LSGP33Flow ) );

      /*************************************
       **** RULE: L_PUP18P22
       *************************************/
MakeRule( L_PUP18P22, [Node|Nodes],
    Node:Item #= P22 And Node:Variable #= PFlow,
    {
    List( Node:Item, PUP18, Node:Variable, PumpAOrB, 60, 40, 1,
       0, 100, FALSE, LPUP18P22 );
    List( Node:Item, PUP18, Node:Variable, PumpAOrB, 60, "-1*sx",
       0, -100, 0, FALSE, LPUP18P22 );
```

```
} );

    /*************************************
     **** RULE: L_P22P41
     *************************************/
MakeRule( L_P22P41, [Node|Nodes],
   Node:Item #= P22 And Node:Variable #= Flow,
   List( Node:Item, P41, Node:Variable, Flow, 2900, v, "v*5/8",
      -100, 100, FALSE, LP22P41 ) );

    /*************************************
     **** RULE: L_P36Screw1On
     *************************************/
MakeRule( L_P36Screw1On, [Node|Nodes],
   Node:Item #= P36 And Node:Variable #= Flow,
   List( Node:Item, CVP01, Node:Variable, ScrewCon, 60, v, 1,
      0, 20, FALSE, LP36Screw1On ) );
```

CONTROL RULES

```
    /*************************************
     **** RULE: C_Level
     *************************************/
MakeRule( C_Level, [Item|Items],
   IsAKindOf?( Item, Storage ) And Item:Level > 80,
   Obj( Item, Level, 3, 70 - Item:Level, 6000, 6000, FALSE, " The tank "#Item#"  is over 80%
full",
      CLevel, 3 ) );

    /*************************************
     **** RULE: C_LevelLw
     *************************************/
MakeRule( C_LevelLw, [Item|Items],
   IsAKindOf?( Item, Storage ) And Item:Level < 20 And TKP18:DLevel
      < 0,
   Obj( Item, Level, 3, 20 - Item:Level, 6000, 6000, FALSE, "The tank "#Item#"  is under 20%
full",
      CLevelLw, 1 ) );

    /*************************************
     **** RULE: C_DLevelHi
     *************************************/
MakeRule( C_DLevelHi, [Item|Items],
   Item #= TKP18 And Item:DLevel > Item:DLevMax And TKP18:Level
      > 40,
   Obj( P41, Flow, 5,
      {
      0.5 * TKP18:DLevel;
```

```
    }, 1500, 2000, TRUE, "The rate of change of " # Item # " is over "
                    # Item:DLevMax # "%/hr", C_DLevelHi,
    2 ) );

    /*************************************
     **** RULE: C_LevelHi2
     *************************************/
MakeRule( C_LevelHi2, [Item|Items],
  Item #= TKP33 And Item:Level > 95,
    { Item, Level, 3, 85 - TKP33:Level, 600, 600, FALSE, " The calcine storage tank is over
95 full",
    CLevelHi2, 4 ) );

    /*************************************
     **** RULE: C_DLevelLw
     *************************************/
MakeRule( C_DLevelLw, [Item|Items],
  Item #= TKP18 And Item:DLevel < Item:DLevMin And TKP18:Level
    < 80,
  Obj( P41, Flow, 5, 1 * Item:DLevel, 6000, 6000, TRUE,
    "The rate of change of tank " # Item # " is less than -" #
      Item:DLevMax # "%/hr", CDLevelLw, 5 ) );

    /*************************************
     **** RULE: C_TKP33Level
     *************************************/
MakeRule( C_TKP33Level, [Item|Items],
  TKP33:Level > 95,
  Obj( TKP33, Request, Level, 5, 90 - TKP33:Level, 1800, 1800,
    FALSE, " The calcine slurry tank is too full", CTKP33Level,
    10 ) );

    /*************************************
     **** RULE: C_DLevMax1
     ****
     *************************************/
MakeRule( C_DLevMax1, [Item|Items],
  TKP18:Level > 65 And TKP18:DLevMax > 0,
  Obj( TKP18, DLevMax, 5, -1 * TKP18:DLevMax, 120, 120, FALSE,
    "Changing the maximum allowable rate of level change", CDLevMax1,
    7 ) );
SetRuleComment( C_DLevMax1, " " );

    /*************************************
     **** RULE: C_DLevMax2
     *************************************/
MakeRule( C_DLevMax2, [Item|Items],
  TKP18:Level < 60 And TKP18:DLevMax < 2.5,
  Obj( TKP18, DLevMax, 5, 2.5 - TKP18:DLevMax, 120, 120, FALSE,
```

```
      "Changing the maximum allowable rate of level change", CDLevMax2,
      8 ) );

    /*************************************
     **** RULE: C_DLevMax3
     *************************************/
MakeRule( C_DLevMax3, [Item|Items],
   TKP18:Level < 60 And TKP18:DLevMax > 2.5,
   Obj( TKP18, DLevMax, 5, 2.5 - TKP18:DLevMax, 120, 120, FALSE,
      " Changing the maximum allowable rate of level change", CDLevMax3,
      9 ) );

    /*************************************
     **** RULE: C_DLevMin1
     *************************************/
MakeRule( C_DLevMin1, [Item|Items],
   TKP18:Level < 35 And TKP18:DLevMin < 0,
   Obj( TKP18, DLevMin, 5, -1 * TKP18:DLevMin, 120, 120, FALSE,
      "Changing the maximum allowable rate of level change", CDLevMin1,
      11 ) );

    /*************************************
     **** RULE: C_DLevMin2
     *************************************/
MakeRule( C_DLevMin2, [Item|Items],
   TKP18:Level > 35 And TKP18:DLevMin > -2.5,
   Obj( TKP18, DLevMin, 5, -2.5 - TKP18:DLevMin, 120, 120, FALSE,
      " Changing the maximum allowable rate of level change", CDLevMin2,
      12 ) );

    /*************************************
     **** RULE: C_DLevMin3
     *************************************/
MakeRule( C_DLevMin3, [Item|Items],
   TKP18:Level > 35 And TKP18:DLevMin < -2.5,
   SendMessage( TKP18, Request, DLevMin, 5, -2.5 - TKP18:DLevMin,
      120, 120, FALSE, "Changing the maximum allowable rate of level change",
      CDLevMin3, 13 ) );
```

VALUE RULES

```
    /*************************************
     **** RULE: V_FTime
     *************************************/
MakeRule( V_FTime, [Item|Items],
   Item:Variable #= TKP18 And Item:Variable #= FTime,
   SendMessage( TKP18, SetVar, FTime, ( 80 - TKP18:Level ) /
                                      TKP18:DLevel ) );
```

```
    /*************************************
     **** RULE: V_PAOrPB
     *************************************/
MakeRule( V_PAOrPB, [Item|Items],
   ( PUP18:PumpA #= On ) Or ( PUP18:PumpB #= ON ),
   SendMessage( Item, SetVar, PumpAOrB, 1 ) );

    /*************************************
     **** RULE: V_P36
     *************************************/
MakeRule( V_P36, [Item|Items],
   CVP01:ScrewCon #= ON,
   SendMessage( P36, SetVar, Flow, CVP01:Frequency ) );

    /*************************************
     **** RULE: V_P36_1
     *************************************/
MakeRule( V_P36_1, [Item|Items],
   CVP01:ScrewCon #= OFF,
   SendMessage( P36, SetVar, Flow, 0 ) );

    /*************************************
     **** RULE: V_P37_1
     *************************************/
MakeRule( V_P37_1, [Item|Items],
   CVP02:ScrewCon #= ON,
   SendMessage( P37, SetVar, Flow, 0.5 * CVP02:Frequency ) );

    /*************************************
     **** RULE: V_P37_2
     *************************************/
MakeRule( V_P37_2, [Item|Items],
   CVP02:ScrewCon #= OFF,
   SendMessage( P37, SetVar, Flow, 0 ) );

    /*************************************
     **** RULE: V_P38
     *************************************/
MakeRule( V_P38, [Item|Items],
   Item:Flow > -1 And Item #= P36,
   SendMessage( P38, SetVar, Flow, P36:Flow + P37:Flow ) );

    /*************************************
     **** RULE: V_P41
     *************************************/
MakeRule( V_P41, [Item|Items],
   P38:Flow > -1,
   SendMessage( P41, SetVar, Flow, P38:Flow + P33:Flow ) );
```

```
/*************************************
 **** RULE: V_CVP02Freq
 *************************************/
MakeRule( V_CVP02Freq, [Item|Items],
   CVP01:Frequency < 55,
   SendMessage( CVP02, SetVar, Frequency, CVP01:Frequency ) );
```

ACTION COST RULES

```
/*************************************
 **** RULE: AC_P22Stop
 *************************************/
MakeRule( AC_P22Stop, [Action|Actions],
   Action:Item #= P22 And Action:Variable #= Flow And Action:Variation
      < 0 And TKP18:Level < 10,
   SendMessage( Action, AltCost, -1 ) );

/*************************************
 **** RULE: AC_P22FlowStrt
 *************************************/
MakeRule( AC_P22FlowStrt, [Action|Actions],
   Action:Item #= P22 And Action:Variable #= Flow And Action:Variation
      > 0 And P22:Flow < 5,
   SendMessage( Action, AltCost, -1 ) );

/*************************************
 **** RULE: AC_P41Unchoke
 *************************************/
MakeRule( AC_P41Unchoke, [Action|Actions],
   Action:Item #= P41 And Action #= Unchoke And TKP33:Level >
   95,
   SendMessage( Action, AltCost, -0.8 ) );

/*************************************
 **** RULE: AC_CVP01Swch
 *************************************/
MakeRule( AC_CVP01Swch, [Action|Actions],
   Action:Item #= CVP01 And Action #= CVP01Switch And CVP02:ScrewCon
      #= ON,
   SendMessage( Action, AltCost, -0.8 ) );

/*************************************
 **** RULE: AC_CVP02Swch
 *************************************/
MakeRule( AC_CVP02Swch, [Action|Actions],
   Action:Item #= CVP02 And Action #= CVP02Switch And CVP01:ScrewCon
      #= ON,
   SendMessage( Action, AltCost, -0.8 ) );
```

```
    /*************************************
     **** RULE: AC_ScrewFreq
     *************************************/
MakeRule( AC_ScrewFreq, [Action|Actions],
   ( Action:Item #= CVP01 Or Action:Item #= CVP02 )
      And ( Action:Variable #= Frequency ) And ( Action:Variation
                                              < 0 )
      And ( CVP01:Frequency < 26 ),
   SendMessage( Action, AltCost, 1 ) );

    /*************************************
     **** RULE: AC_ScrewFreq2
     *************************************/
MakeRule( AC_ScrewFreq2, [Action|Actions],
   ( ( Action:Item #= CVP01 And CVP01:Frequency > 35 )
       Or ( Action:Item #= CVP02 And CVP02;Frequency > 45 ) )
      And Action:Variable #= Frequency And Action:Variation
      > 0,
   SendMessage( Action, AltCost, 1 ) );
```

LINK COST RULES

```
    /*************************************
     **** RULE: LC_P38P37_1
     *************************************/
MakeRule( LC_P38P37_1, [Link|Links],
   Link:PItem #= P37 And Link:PVariable #= Flow And Link:SItem
      #= P38 And Link:SVariable #= Flow And Link:SVariation
      < 0 And Link:Variation > 0 And CVP01:ScrewCon #= ON And
      CVP01:Frequency < 30,
   SendMessage( Link, AltCost, -1 ) );

    /*************************************
     **** RULE: LC_P38P36_1
     *************************************/
MakeRule( LC_P38P36_1, [Link|Links],
   Link:PItem #= P36 And Link:PVariable #= Flow And Link:SItem
      #= P38 And Link:SVariable #= Flow And Link:SVariation
      > 0 And CVP02:ScrewCon #= ON And CVP02:Frequency > 44,
   SendMessage( Link, AltCost, -1 ) );

    /*************************************
     **** RULE: LC_CVP01
     *************************************/
MakeRule( LC_CVP01, [Link|Links],
   Link:PItem #= CVP01 And Link:PVariable #= ScrewCon And CVP02:ScrewCon
      #= ON,
   SendMessage( Link, AltCost, -1 ) );
```

```
    /*************************************
     **** RULE: LC_CVP02
     *************************************/
MakeRule( LC_CVP02, [Link|Links],
   Link:PItem #= CVP02 And Link:PVariable #= ScrewCon And CVP01:ScrewCon
      #= ON,
   SendMessage( Link, AltCost, -1 ) );


    /*************************************
     **** RULE: LC_P38P37_3
     *************************************/
MakeRule( LC_P38P37_3, [Link|Links],
   Link:PItem #= P37 And Link:PVariable #= Flow And Link:SItem
      #= P38 And Link:SVariable #= Flow And Link:SVariation
      < 0 And Link:Variation > 0 And Not( CVP01:ScrewCon #=
                                    ON And CVP01:Frequency
                                    < 30 ),
   SendMessage( Link, AltCost, 1 ) );

    /*************************************
     **** RULE: LC_P38P37_2
     *************************************/
MakeRule( LC_P38P37_2, [Link|Links],
   Link:PItem #= P37 And Link:PVariable #= Flow And CVP02:ScrewCon
      #= ON,
   SendMessage( Link, AltCost, -0.8 ) );

    /*************************************
     **** RULE: LC_P38P36_2
     *************************************/
MakeRule( LC_P38P36_2, [Link|Links],
   Link:PItem #= P36 And Link:PVariable #= Flow And CVP01:ScrewCon
      #= ON,
   SendMessage( Link, AltCost, -0.8 ) );

    /*************************************
     **** RULE: LC_P38P37_4
     *************************************/
MakeRule( LC_P38P37_4, [Link|Links],
   Link:PItem #= P37 And Link:PVariable #= Flow And Link:Variation
      > 0 And CVP02:Frequency > 44,
   SendMessage( Link, AltCost, 1 ) );

    /*************************************
     **** RULE: LC_P38P37_5
     *************************************/
MakeRule( LC_P38P37_5, [Link|Links],
   Link:PItem #= P37 And Link.Variation > 0 And Link:SVariation
```

```
    > 0 And CVP01:ScrewCon #= ON,
SendMessage( Link, AltCost, 1 ) );
```

APPENDIX D PARTIAL CODE LISTING

This code is written in KAL, an interperator language for the shell being used. The shell provided an object-orientated platform on which the work described in the thesis was built.

Omitted

Page 150 not included in the pagination.

```
/*************************************************************/
/**        ALL FUNCTIONS ARE SAVED BELOW                    **/
/*************************************************************/

      /*****************************************
       **** FUNCTION: Start
       **** This function is called to start the knowledge base.
       *****************************************/
MakeFunction( Start, [],
   {
   If ( Null?( Global:RunState ) Or Not( Global:RunState #= Running ) )
    Then {
        NULL;
        Init( );
        Control( );
        };
   } );
SetFunctionComment( Start, "This function is called to start the knowledge base." );

      /*****************************************
       **** FUNCTION: Init
       **** This function initilizes the system.
       *****************************************/
MakeFunction( Init, [],
   {
   Global:State = INITIAL;
   HideWindow( KTOOLS );
   HideWindow( KAL );
   IconifyWindow( KAPPA );
   SetMenuBar( SESSION, HalEdit );
   ShowWindow( SESSION );
   MaximizeWindow( SESSION );
   SetMenuBar( BROWSER, BrowserMenu );
   Explanation:Explain = TRUE;
   Explanation:Error = TRUE;
   SendMessage( Explanation, Clear );
   SendMessage( Explanation, Add, FormatValue( "\n  Initilizing HAL Expert       \n" ) );
   ResetClock( );
   Text1:Title = Date( );
   Text2:Title = Time( );
   Global:TempAction = NULL;
   SetForwardChainMode( DEPTHFIRST );
   Global:StartTime = 0;
   Global:StartTime = GetTime( );
   SetTimer( 1, 30, 0 );
   SetTimer( 2, 1, 0 );
   SetTimer( 3, Global:BackUpRate, 0 );
   SetTimer( 4, 0, Global:ActionSpacing );
   SetTimer( 5, Global:SimulationSpacing, 0 );
```

```
If Not( Instance?( Rule ) )
  Then {
       MakeInstance( Rule, Root );
       MakeSlot( Rule, ControlRules );
       SetSlotOption( Rule:ControlRules, MULTIPLE );
       MakeSlot( Rule, DefaultRules );
       SetSlotOption( Rule:DefaultRules, MULTIPLE );
       MakeSlot( Rule, CostRules );
       SetSlotOption( Rule:CostRules, MULTIPLE );
       MakeSlot( Rule, LinkRules );
       SetSlotOption( Rule:LinkRules, MULTIPLE );
       MakeSlot( Rule, ActionRules );
       SetSlotOption( Rule:ActionRules, MULTIPLE );
       MakeSlot( Rule, ValueRules );
       SetSlotOption( Rule:ValueRules, MULTIPLE );
       };
ClearList( Global:NewSetNodes );
ClearList( Global:SetNodes );
ClearList( Global:NewLinks );
ClearList( Global:Item_List );
ClearList( Global:Init_List );
ClearList( Global:Temp_List );
ClearList( Global:Activated_Nodes );
ClearList( Global:ActionList );
GetInstanceList( Nodes, Global:Init_List );
EnumList( Global:Init_List, x, DeleteInstance( x ) );
ClearList( Global:Init_List );
GetInstanceList( Copies, Global:Init_List );
EnumList( Global:Init_List, x, DeleteInstance( x ) );
ClearList( Global:Init_List );
EnumSubClasses( Actions, XAction,
   {
   GetInstanceList( XAction, Global:Init_List );
   EnumList( Global:Init_List, X,
      {
      SendMessage( X, Init );
      } );
   } );
ClearList( Global:Init_List );
EnumSubClasses( Links, XLink,
   {
   GetInstanceList( XLink, Global:Init_List );
   EnumList( Global:Init_List, X,
      {
      SendMessage( X, Init, CLEAR );
      } );
   } );
SendMessage( ActionDisp, Init );
Global:ItemNo = CountAllInstances( Items ) + 1;
Global:VarNo = CountAllSubClasses( Variables ) + 1;
CatchError( HBOpen( Global:StationID ), Error( 20, NULL, NULL ) );
ShowWindow( Session3 );
```

```
    MaximizeWindow( Session3 );
    Global:Continue = Yes;
    Global:RunState = NULL;
    ) );
SetFunctionComment( Init, "This function initilizes the system." );

    /*************************************
     **** FUNCTION: Control
     **** This  function forms the main control loop in which the program cycles through.
     *************************************/
MakeFunction( Control, [],
    {
    Global:RunState = Running;
    If Not( Member?( HalEdit:Disabled, Run ) )
      Then AppendToList( HalEdit:Disabled, Run );
    SetMenuBar( SESSION, HalEdit );
    While  (( Global:Continue #= Yes ))
        {
        Global:State = WAITING;
        While  (WaitForInput( ))
            {
            Global:Busy = NULL;
            };
        Global:Busy = BUSY;
        MapEscapeKey( Menu );
        Global:State = READING;
        Get_Data( );
        Global:State = CHAIN;
        ForwardChain( NULL, Rule:ValueRules, Rule:ControlRules );
        While  (WaitForInput( ))
            {};
        Global:State = RESET;
        If ( LengthList( Global:Activated_Nodes )
              > 0 )
          Then {
            EnumList( Global:Activated_Nodes, Node,
              If Instance?( Node )
                Then SendMessage( Node, Reset ) );
            };
        While  (( Not( LengthList( Global:NewSetNodes )
                    == 0 ) And ( Global:Continue #= Yes ) ))
            {
            While  (WaitForInput( ))
                {};
            Global:State = SORTING;
            SendMessage( Quicksort, Shell, Global:NewSetNodes,
              Priority );
            ClearList( Global:SetNodes );
            AppendToList( Global:SetNodes, Quicksort:Item );
            If ( LengthList( Global:SetNodes ) > 0 )
              Then EnumList( Global:SetNodes, Ex,
                    {
```

```
                    If Member?( Global:NewSetNodes, Ex )
                     Then {
                         Global:AVTimeS = GetTime( );
                         While (WaitForInput( ))
                             {};
                         Global:State = SEARCHING;
                         Global:Active = TRUE;
                         ClearList( Global:UnexpandedList );
                         ClearList( Global:ExpandedList );
                         SendMessage( Explanation, Display,
                            "\n ---------- Working on "
                              # Ex # " ----------" );
                         Global:Temp = SendMessage( Ex,
                                  Seek, 0 );
                         While (( Global:Active And
                                  ( Global:Continue
                                      #= Yes ) ))
                             {
                             While (( ( Not( Global:Temp
                                          #=
                                          FOUND )
                                      And ( LengthList( Global:UnexpandedList )
                                              >
                                              0 ) )
                                     And ( Global:Continue
                                             #=
                                             Yes ) ))
                                 {
                                 While (WaitForInput( ))
                                     {};
                                 SendMessage( Quicksort,
                                    Shell, Global:UnexpandedList,
                                    ECost );
                                 ClearList( Global:UnexpandedList );
                                 AppendToList( Global:UnexpandedList,
                                    Quicksort:Item );
                                 Global:Temp = SendMessage( GetNthElem(
Global:UnexpandedList,
                                                          1 ),
                                                  Expand );
                                 };
                             Global:State = ACTIVATING;
                             If ( Global:Temp #= FOUND )
                              Then {
                                  If SendMessage( Ex,
                                         Activate,
                                         Ex )
                                   Then {
                                      Global:State
                                         = CLEARING;
                                      EnumList( Global:UnexpandedList,
                                         UNEX,
```

```
                                        {
                                        SendMessage( UNEX,
                                            Init,
                                            NORMAL );
                                        } );
                                    EnumList( Global:ExpandedList,
                                        UNEX,
                                        {
                                        SendMessage( UNEX,
                                            Init,
                                            NORMAL );
                                        } );
                                    Global:Active
                                        = FALSE;
                                    }
                                  Else Global:Temp
                                        = TRUE;
                                }
                              Else {
                                  If Instance?( Ex )
                                   Then {
                                      If Null?( Ex:Copies )
                                        Then SendMessage( Ex,
                                                NoSolution )
                                        Else SendMessage( Ex,
                                                End,
                                                Ex,
                                                NORMAL );
                                      };
                                  Global:Active =
                                     FALSE;
                                  };
                            };
                        Global:AVTimeF = GetTime( );
                        SendMessage( Explanation, Display,
                            "\n
----------");
                        };
                    } );
            };
        };
    Global:State = READY";
    Terminate( );
    } );
SetFunctionComment( Control, "This  function forms the main control loop in which the program cycles
through." );

    /*************************************
     **** FUNCTION: Get_Data
     **** This function calls each item getting them to get the necessar, data.
     *************************************/
MakeFunction( Get_Data, [],
```

```
    {
    ClearList( Global:Item_List );
    EnumSubClasses( Items, Item,
       {
       GetInstanceList( Item, Global:Item_List );
       EnumList( Global:Item_List, An_Item,
          {
          SendMessage( An_Item, Get_Value );
          } );
       } );
    } );
SetFunctionComment( Get_Data, "This function calls each item getting them to get the necessary data." );

      /*************************************
       **** FUNCTION: Read_Data
       **** This function provides the interface between the KB and the H&B.
       *************************************/
MakeFunction( Read_Data, [Tag],
    {
    While  (WaitForInput( ))
         {};
    Global:ReadState = READ;
    If Global:Simulation
     Then {
          If Slot?( Simulation, Tag )
           Then {
              Global:Value = Simulation:Tag;
              }
           Else {
              Global:Value = NULL;
              };
          }
     Else {
          Global:Value = CatchError( {
                                HBRead( {
                                      Tag;
                                      }, 1 );
                                },
                           {
                           Global:ReadState = NULL;
                           Error( 20, Read_Data, Tag );
                           Global:ReadState = CLOSE;
                           CatchError( HBClose( ),
                              Error( 20 ) );
                           Global:ReadState = OPEN;
                           CatchError( HBOpen( Global:StationID ),
                              Error( 20, NULL, NULL ) );
                           NULL;
                           } );
          };
    Global:ReadState = NULL;
    If ( Not( Null?( Global:Value ) ) And Number?( Global:Value ) )
```

```
    Then Global:Value
    Else NULL;
  } );
SetFunctionComment( Read_Data, "This function provides the interface between the KB and the H&B."
);

    /*************************************
     **** FUNCTION: Clear_KB
     **** This function will clear any links created during initilization.
     *************************************/
MakeFunction( Clear_KB, [],
  {
  ClearList( Global:Init_List );
  EnumSubClasses( Actions, Inst,
    {
    GetInstanceList( Inst, Global:Init_List );
    EnumList( Global:Init_List, Element,
      {
      If Instance?( Element )
        Then DeleteInstance( Element );
      } );
    If ( Not( Inst #= Actions ) And CountSubClasses( Inst )
          == 0 And Not( Inst #= Default ) )
      Then DeleteClass( Inst );
    } );
  ClearList( Global:Init_List );
  EnumSubClasses( Links, Inst,
    {
    GetInstanceList( Inst, Global:Init_List );
    EnumList( Global:Init_List, Element,
      {
      If Instance?( Element )
        Then DeleteInstance( Element );
      } );
    If ( Not( Inst #= Links ) And CountSubClasses( Inst )
          == 0 )
      Then DeleteClass( Inst );
    } );
  ClearList( Global:Init_List );
  GetInstanceList( Nodes, Global:Init_List );
  EnumList( Global:Init_List, Instance,
    {
    DeleteInstance( Instance );
    } );
  ClearList( Global:Init_List );
  EnumSubClasses( Items, Inst,
    {
    GetInstanceList( Inst, Global:Init_List );
    EnumList( Global:Init_List, Element,
      {
      If Instance?( Element )
        Then DeleteInstance( Element );
```

```
    } );
  If ( Not( Inst #= Items ) And CountSubClasses( Inst )
       == 0 )
    Then DeleteClass( Inst );
  } );
While (Not( CountSubClasses( Items ) == 0 ))
  {
  EnumSubClasses( Items, Item,
    {
    If ( Not( Item #= Items ) And CountSubClasses( Item )
         == 0 )
      Then DeleteClass( Item );
    } );
  };
ClearList( Global:Init_List );
EnumSubClasses( Variables, Inst,
  {
  GetInstanceList( Inst, Global:Init_List );
  EnumList( Global:Init_List, Element,
    {
    If Instance?( Element )
      Then DeleteInstance( Element );
    } );
  If Not( Inst #= Variables )
    Then DeleteClass( Inst );
  } );
If Instance?( Rule )
  Then {
    If Slot?( Rule:ControlRules )
      Then {
        EnumList( Rule:ControlRules, Rule,
          If Rule?( Rule )
            Then DeleteRule( Rule ) );
        ClearList( Rule:ControlRules );
        };
    If Slot?( Rule:LinkRules )
      Then {
        EnumList( Rule:LinkRules, Rule, If Rule?( Rule )
                                    Then DeleteRule( Rule ) );
        ClearList( Rule:LinkRules );
        };
    If Slot?( Rule:DefaultRules )
      Then {
        EnumList( Rule:DefaultRules, Rule,
          If Rule?( Rule )
            Then DeleteRule( Rule ) );
        ClearList( Rule:DefaultRules );
        };
    If Slot?( Rule:CostRules )
      Then {
        EnumList( Rule:CostRules, Rule, If Rule?( Rule )
                                    Then DeleteRule( Rule ) );
```

```
            ClearList( Rule:CostRules );
            };
        If Slot?( Rule:ActionRules )
          Then {
            EnumList( Rule:ActionRules, Rule,
              If Rule?( Rule )
                Then DeleteRule( Rule ) );
            ClearList( Rule:ActionRules );
            };
        If Slot?( Rule:ValueRules )
          Then {
            EnumList( Rule:ValueRules, Rule,
              If Rule?( Rule )
                Then DeleteRule( Rule ) );
            ClearList( Rule:ValueRules );
            };
        DeleteInstance( Rule );
        };
  } );
SetFunctionComment( Clear_KB, "This function will clear any links created during initilization." );

    /*************************************
     **** FUNCTION: Opposite?
     *************************************/
MakeFunction( Opposite?, [Value1 Value2],
  Abs( Value1 + Value2 ) < ( Abs( Value1 ) + Abs( Value2 ) ) );

    /*************************************
     **** FUNCTION: CostAdd
     *************************************/
MakeFunction( CostAdd, [X Y],
  {
  If ( ( X > 0 ) And ( Y > 0 ) )
    Then {
      X + Y - X * Y;
      }
    Else If ( ( X < 0 ) And ( Y < 0 ) )
      Then {
        Negative( Negative( X ) - Y - ( X * Y ) );
        }
      Else {
        ( X + Y ) / ( 1 + Min( Abs( X ), Abs( Y ) ) );
        };
  } );

    /*************************************
     **** FUNCTION: CostVar
     *************************************/
MakeFunction( CostVar, [AV DV],
  {
  If Not( Null?( AV ) Or Null?( DV ) )
    Then {
```

```
        If ( Number?( AV ) And Number?( DV ) )
          Then {
            If ( AV == 0 )
              Then ( Global:Cost = 1 )
              Else {
                If ( DV / AV > 1 )
                  Then {
                    Global:Cost = 0.5 * DV / AV -
                      0.5;
                    }
                  Else {
                    Global:Cost = -3 / 2 * DV / AV
                      + 3 / 2;
                    };
                If ( Global:Cost > 1 )
                  Then Global:Cost = 1;
                If ( Global:Cost < -1 )
                  Then Global:Cost = -1;
                };
            }
          Else {
            Error( 8 );
            Global:Cost = 1;
            };
        }
      Else {
        Global:Cost = 1;
        };
    Global:Cost;
    } );

      /*************************************
       **** FUNCTION: CostFail
       *************************************/
MakeFunction( CostFail, [Suc Fail],
    {
    If ( ( Suc + Fail ) == 0 )
      Then 0
      Else ( Fail / ( Suc + Fail ) - 0.5 ) * 1.6;
    } );

      /*************************************
       **** FUNCTION: CostTime
       *************************************/
MakeFunction( CostTime, [CritT Time],
    {
    If ( CritT == 0 )
      Then ( Global:Cost = 1 )
      Else Global:Cost = 1 / CritT * Time;
    If ( Global:Cost > 1 )
      Then Global:Cost = 1;
    If ( Global:Cost < -1 )
```

```
  Then Global:Cost = -1;
 Global:Cost;
 } );

   /*************************************
    **** FUNCTION: CostSlf
    **** This function uses CostSlf to store it's data. It determines the cost of alering the effect.
    *************************************/
MakeFunction( CostSlf, [Caller],
 {
 Let [I Caller:Item]
   {
   Let [V Caller:Variable]
     {
     V;
     If Not( SendMessage( I, GetVal, V, Type )
           #= Analog )
      Then {
         CostSlf:Cost = 0;
         }
      Else {
         CostSlf:X = Caller:OriginalValue + Caller:Variation;
         Let [x CostSlf:X]
           {
           If Null?( x )
            Then {
               CostSlf:Cost = 1;
               }
            Else {
               If ( x >= SendMessage( I, GetVal,
                        V, SP ) )
                 Then {
                    If (   > SendMessage( I, GetVal,
                          V, UL ) )
                      Then {
                         CostSlf:Temp = SendMessage( I,
                                    GetVal,
                                    V,
                                    MAX )
                           - SendMessage( I,
                              GetVal, V,
                              UL );
                         If ( CostSlf:Temp
                              == 0 )
                          Then ( CostSlf:Cost
                               = 1 )
                          Else CostSlf:Cost
                               = 1 / CostSlf:Temp
                               * ( x -
                                    SendMessage( I,
                                      GetVal,
                                      V,
```

```
                                UL ) );
            }
          Else {
              CostSlf:Temp = SendMessage( I,
                                GetVal,
                                V,
                                UL )
                - SendMessage( I,
                    GetVal, V,
                    SP );
              If ( CostSlf:Temp
                    == 0 )
               Then ( CostSlf:Cost
                      = 1 )
               Else CostSlf:Cost
                      = ( x -
                            SendMessage( I,
                              GetVal,
                              V,
                              UL ) )
                      / CostSlf:Temp;
              };
        }
    Else {
        If ( x < SendMessage( I, GetVal,
                 V, LL ) )
         Then {
            CostSlf:Temp = SendMessage( I,
                              GetVal,
                              V,
                              MIN )
              - SendMessage( I,
                  GetVal, V,
                  LL );
            If ( CostSlf:Temp
                  == 0 )
             Then ( CostSlf:Cost
                    = 1 )
             Else CostSlf:Cost
                    = 1 / CostSlf:Temp
                    * ( x -
                          SendMessage( I,
                            GetVal,
                            V,
                            LL ) );
          }
         Else {
            CostSlf:Temp = SendMessage( I,
                              GetVal,
                              V,
                              SP )
              - SendMessage( I,
```

```
                                        GetVal, V,
                                        LL );
                                    If ( CostSlf:Temp
                                        == 0 )
                                     Then ( CostSlf:Cost
                                        = 1 )
                                     Else CostSlf:Cost
                                        = ( SendMessage( I,
                                            GetVal,
                                            V,
                                            LL )
                                            -
                                            x )
                                        / CostSlf:Temp;
                                    };
                                };
                            };
                        };
                    };
                };
            };
        If ( CostSlf:Cost > 1 )
         Then CostSlf:Cost = 1;
        If ( CostSlf:Cost < -1 )
         Then CostSlf:Cost = -1;
        CostSlf:Cost;
        } );
SetFunctionComment( CostSlf, "This function uses CostSlf to store it's data. It determines the cost of
alering the effect " );

    /************************************
     **** FUNCTION: List
     ************************************/
MakeFunction( List, [ASI API ASV APV ATREL AVSREL AVPREL MIN MAX WAIT Caller],
    {
    If Not( Class?( L_ # ASI ) )
     Then MakeClass( L_ # ASI, Links );
    ClearList( Global:NLinkList );
    GetInstanceList( L_ # ASI, Global:NLinkList );
    Global:ListTemp = TRUE;
    EnumList( Global:NLinkList, NL,
        {
        If ( ( NL:Creator #= Caller ) And ( NL:PItem #= API )
                And ( NL:PVariable #= APV ) And ( NL:SItem #=
                                        ASI )
                And ( NL:SVariable #= ASV ) And ( NL:MaxVar #=
                                        MAX )
                And ( NL:MinVar #= MIN ) )
         Then {
            Global:ListTemp = FALSE;
            If Not( ( NL:VPRelation #= AVPREL ) And ( NL:TRelation
                                        #=
```

```
                                                    ATREL )
                        And ( NL:VSRelation #== AVSREL ) )
                Then {
                    SendMessage( NL, Set, ASI, API, ASV, APV,
                        ATREL, AVSREL, AVPREL, MIN, MAX, WAIT,
                        Caller );
                    };
                };
        } );
    If Global:ListTemp
     Then {
        Global:Counter = 0;
        While  (Instance?( ASV # _ # Global:Counter ))
            {
            Global:Counter += 1;
            };
        MakeInstance( ASV # _ # Global:Counter,
            L_ # ASI );
        SendMessage( ASV # _ # Global:Counter, Set, ASI, API,
            ASV, APV, ATREL, AVSREL, AVPREL, MIN, MAX, WAIT,
            Caller );
        };
    } );

      /*************************************
       **** FUNCTION: ClearLinks
       *************************************/
MakeFunction( ClearLinks, [],
    {
    EnumSubClasses( Links, L,
        {
        GetInstanceList( L, Global:Init_List );
        EnumList( Global:Init_List, IL, SendMessage( IL, Init,
                                        IL ) );
        } );
    } );

      /*************************************
       **** FUNCTION: TimerFunc
       *************************************/
MakeFunction( TimerFunc, [TimerNumber],
    {
    If ( TimerNumber == 1 )
     Then {
        Clean( );
        KillTimer( 1 );
        SetTimer( 1, 30, 0 );
        };
    If ( TimerNumber == 2 )
     Then {
        Text2:Title = Time( );
        ResetImage( Text2 );
```

```
        };
    If ( TimerNumber == 3 )
     Then {
        SendMessage( File, BackUp );
        ClearTranscriptImage( Error );
        KillTimer( 3 );
        SetTimer( 3, Global:BackUpRate, 0 );
        };
    If ( TimerNumber == 4 )
     Then {
        ActionHandler( );
        KillTimer( 4 );
        SetTimer( 4, 0, Global:ActionSpacing );
        };
    If ( TimerNumber == 5 )
     Then {
        SendMessage( Simulation, Evaluate );
        };
    } );

      /*************************************
       **** FUNCTION: Quit
       *************************************/
MakeFunction( Quit, [],
    {
    HideWindow( Session1 );
    Global:Quit = TRUE;
    } );

      /*************************************
       **** FUNCTION: Update
       *************************************/
MakeFunction( Update, [],
    {
    EnumSubClasses( Image, I,
       {
       ClearList( Global:Init_List );
       GetInstanceList( I, Global:Init_List );
       EnumList( Global:Init_List, G, ResetImage( G ) );
       } );
    } );

      /*************************************
       **** FUNCTION: ActionHandler
       *************************************/
MakeFunction( ActionHandler, [],
    {
    If ( ( LengthList( Global:ActionList ) > 0 ) Or Not( Null?( Global:TempAction ) ) )
     Then {
        If ( LengthList( Global:ActionList ) > 0 )
         Then {
            SendMessage( Quicksort, Shell, Global:ActionList,
```

```
                Priority );
            ClearList( Global:ActionList );
            AppendToList( Global:ActionList, Quicksort:Item );
            };
        If Not( Null?( Global:TempAction ) And Not( Member?( Global:ActionList,
                                                Global:TempAction ) ) )
          Then AppendToList( Global:ActionList, Global:TempAction );
        If ( Not( Member?( SingleListBox1:AllowableValues, GetNthElem( Global:ActionList,
                                                1 ) ) )
              And Not( GetValue( GetNthElem( Global:ActionList,
                                1 ), Acknowledge ) ) )
          Then {
            AppendToList( SingleListBox1:AllowableValues,
              GetNthElem( Global:ActionList, 1 ) );
            ResetImage( SingleListBox1 );
            };
        If Not( Member?( ActionDisp:Actions, GetNthElem( Global:ActionList,
                                1 ) ) )
          Then {
            AppendToList( ActionDisp:Actions,
              GetNthElem( Global:ActionList, 1 ) );
            ActionDisp:CurrentChoice = GetNthElem( Global:ActionList,
                                1 );
            };
        If ( LengthList( Global:ActionList ) > 0 )
          Then {
            If ( SendMessage( GetNthElem( Global:ActionList,
                                1 ), Action )
                  #= FALSE )
              Then {
                Global:TempAction = GetNthElem( Global:ActionList,
                                1 );
                If ( LengthList( Global:ActionList )
                        > 0 )
                  Then RemoveNthElem( Global:ActionList,
                        1 );
                }
              Else {
                If ( LengthList( Global:ActionList )
                        > 0 )
                  Then RemoveNthElem( Global:ActionList,
                        1 );
                Global:TempAction = NULL;
                };
            };
        };
    } );

    /*****************************************\
     **** FUNCTION: Menu
     *****************************************/
MakeFunction( Menu, [],
```

```
    {
    Global:RunState = Pause;
    Global:Busy = PAUSE;
    Global:PauseTime = GetTime( );
    While (( Not( Null?( Global:RunState ) ) And ( Global:RunState
                                                  #= Pause ) ))
        {
        While (WaitForInput( ))
            {};
        If ( GetTime( ) - Global:PauseTime > 10 )
          Then Global:RunState = Running;
        };
    Global:Busy = BUSY;
    } );

      /*************************************
       **** FUNCTION: GetTime
       *************************************/
MakeFunction( GetTime, [],
    {
    Times:Time = GetClock( );
    If ( GetClock( ) < 0 )
      Then ResetClock( );
    If ( Times:Time >= Times:CyclePeriod )
      Then {
          ResetClock( );
          Times:CycleNo += 1;
          If ( Times:CycleNo >= Times:Max )
            Then Times:CycleNo = 1;
          };
    Times:Time + ( Times:CycleNo - 1 ) * Times:CyclePeriod;
    } );

      /*************************************
       **** FUNCTION: CompTime
       *************************************/
MakeFunction( CompTime, [T1 T2],
    {
    AdjTime( T1 ) > AdjTime( T2 );
    } );

      /*************************************
       **** FUNCTION: Terminate
       *************************************/
MakeFunction( Terminate, [],
    {
    DisplayText( HalEx, FormatValue( "\n  Ended HAL Expert.  " ) );
    KillTimer( 1 );
    KillTimer( 2 );
    KillTimer( 3 );
    KillTimer( 4 );
    CatchError( HBClose( ), Error( 20 ) );
```

```
Global:RunState = NULL;
} );

    /*************************************
     **** FUNCTION: Error
     *************************************/
MakeFunction( Error, [Num Caller Parameter],
  {
  If ( Num == 1 )
   Then SendMessage( Explanation, Error, FormatValue( "\n ERROR" ) );
  If ( Num == 2 )
   Then SendMessage( Explanation, Error, FormatValue( "\n File Error - Could not open file!" ) );
  If ( Num == 3 )
   Then SendMessage( Explanation, Error, FormatValue( "\n File Error - Error in interpreting
knowledge base !" ) );
  If ( Num == 4 )
   Then SendMessage( Explanation, Error, FormatValue( "\n Read Error - Can't read data file!" ) );
  If ( Num == 5 )
   Then SendMessage( Explanation, Error, FormatValue( "\n File Error - Can't Retrieve File!" ) );
  If ( Num == 6 )
   Then SendMessage( Explanation, Error, FormatValue( "\n File Error - Can't Save Explanation!" )
);
  If ( Num == 7 )
   Then SendMessage( Explanation, Error, FormatValue( "\n File Error - Can't Read Cell" ) );
  If ( Num == 8 )
   Then SendMessage( Explanation, Error, FormatValue( "\n Discrete variable being considered as a
continuous variable" ) );
  If ( Num == 11 )
   Then SendMessage( Explanation, Error, FormatValue( "\n File Error - could not record explanation
data" ) );
  If ( Num == 20 )
   Then SendMessage( Explanation, Error, FormatValue( " HB Error -"
                                                      #
                                                      HBErrorMsg( )
                                                      #
                                                      ", \n      re  g "
                                                      #
                                                      Parameter
                                                      #
                                                      ". " ) );
  If ( Num == 21 )
   Then SendMessage( Explanation, Error, "An action has been lost" );
  If ( Num == 22 )
   Then SendMessage( Explanation, Error, "Variable " # Parameter
                                      # " listed but does not exist in "
                                      # Caller );
  If ( Num == 24 )
   Then SendMessage( Explanation, Error, "Error writing to file "
                                      # Parameter );
  } );

    /*************************************
```

```
      **** FUNCTION: InterWrite
      ************************************/
MakeFunction( InterWrite, [],
   {
   Write_Data( Inter:TagName, Inter:Value );
   } );

      /************************************
      **** FUNCTION: InterRead
      ************************************/
MakeFunction( InterRead, [],
   {
   Inter:Value = Read_Data( Inter:TagName );
   } );

      /************************************
      **** FUNCTION: Write_Data
      ************************************/
MakeFunction( Write_Data, [Value_Tag Number],
   {
   If Not( Value_Tag #= NONE )
     Then {
        Global:ReadState = WRITE;
        Global:Temp = CatchError( {
                              HBWrite( Value_Tag, Number );
                              },
                          {
                          Error( 20, Write_Data, Value_Tag );
                          CatchError( HBClose( ),
                             Error( 20 ) );
                          Global:ReadState = CLOSE;
                          CatchError( HBOpen( Global:StationID ),
                             Error( 20, NULL, NULL ) );
                          Global:ReadState = OPEN;
                          FALSE;
                          } );
        }
     Else Global:Value = NULL;
   Global:ReadState = NULL;
   Global:Temp;
   } );

      /************************************
      **** FUNCTION: BarDraw
      ************************************/
MakeFunction( BarDraw, [Title Num Max Min],
   {
   SetLineStyle( 1, 1, 0 );
   MoveTo( 59, 20 );
   Rectangle( 25, 56 );
   SetLineStyle( 1, 1, 7 );
   For x From 61 To 81
```

```
        Do {
        SetLineStyle( 1, 1, 4 );
        MoveTo( x, 75 );
        LineTo( x, 75 - 55 * Num / Max );
        SetLineStyle( 1, 1, 2 );
        LineTo( x, 20 );
        };
    MoveTo( 30, 85 );
    DrawText( FormatValue( "%3.2f", Num ) );
    MoveTo( 0, 20 );
    DrawText( FormatValue( "%3.2f", Max ) );
    MoveTo( 0, 60 );
    DrawText( FormatValue( "%3.2f", Min ) );
    MoveTo( 15, 0 );
    DrawText( Title );
    } );

      /*************************************
       **** FUNCTION: FailDraw
       *************************************/
MakeFunction( FailDraw, [],
    {
    BarDraw( "% Failures", Global:Percent, 100, 0 );
    } );

      /*************************************
       **** FUNCTION: AVDraw
       *************************************/
MakeFunction( AVDraw, [],
    {
    SetLineStyle( 1, 1, 1 );
    MoveTo( 0, 10 );
    DrawText( "Average Process Time" );
    MoveTo( 20, 60 );
    DrawText( FormatValue( "%3.2f  Sec", Global:AVTime ) );
    } );

      /*************************************
       **** FUNCTION: Clean
       **** This function is used to stop the explanation window becoming too large.
       **** It will save the explanation window every two hours. It does not append at the end of the
file, yet.
       *************************************/
MakeFunction( Clean, [],
    {
    PostBusy( ON, "Cleaning up" );
    SendMessage( Times, Date );
    File:EFileName = File:Path # Hal # Times:Month # Times:Day
      # .Exp;
    If CatchError( OpenWriteFile( File:EFileName, APPEND ), FALSE )
     Then {
        SaveTranscriptImage( HalEx );
```

```
        CloseWriteFile( );
        }
      Else Error( 11, NULL, NULL );
    SendMessage( Explanation, Clear );
    ReclaimStringSpace( );
    MapEscapeKey( Menu );
    SetMenuBar( SESSION, HalEdit );
    SendMessage( Explanation, Add, " Cleaned Explanation Window - data stored in "
                                  # File:EFileName );
    Text1:Title = Date( );
    PostBusy( OFF );
    } );
SetFunctionComment( Clean, "This function is used to stop the explanation window becoming too large.
It will save the explanation window every two hours. It does not append at the end of the file, yet." );

     /*************************************
      **** FUNCTION: Convert
      *************************************/
MakeFunction( Convert, [Item Variable Value],
    {
    If ( Not( Null?( Value ) ) And Not( Number?( Value ) ) )
      Then SendMessage( GetValue( Item, N # {
                                     Variable;
                                     } ), Convert, Value )
      Else Value;
    } );

     /*************************************
      **** FUNCTION: AdjTime
      *************************************/
MakeFunction( AdjTime, [Time],
    {
    Times:TCNo = Floor( Time / Times:CyclePeriod ) - ( Times:CycleNo
                                                    - Floor( Times:Max
                                                             /
                                                             2 ) );
    If ( Times:TCNo > Times:Max )
      Then Times:TCNo = Times:TCNo - Times:Max;
    If ( Times:TCNo < 0 )
      Then Times:TCNo = Times:TCNo + Times:Max;
    Time = ( Floor( Time / Times:CyclePeriod ) - Times:TCNo )
       * Times:CyclePeriod;
    } );

     /*************************************
      **** FUNCTION: UpdateVar
      *************************************/
MakeFunction( UpdateVar, [],
    SendMessage( ItemVars, Update ) );

     /*************************************
      **** FUNCTION: GetDate
```

```
      ************************************/
MakeFunction( GetData, [],
   SendMessage( ItemVars, GetData ) );

      /************************************
       **** FUNCTION: RunApplication
       ************************************/
MakeFunction( RunApplication, [],
   {
   Clear_KB( );
   CatchError( InterpretFile( File:BackUpFile ), SendMessage( Explanation,
                                        Error, "Cannot interpret file "
                                                #
                                                File:BackUpFile ) );
   Global:RunState = NULL;
   Start( );
   } );

      /************************************
       **** FUNCTION: Acknowledge
       ************************************/
MakeFunction( Acknowledge, [],
   {
   If ( Not( Null?( ActionDisp:CurrentChoice ) ) And IsAKindOf?( ActionDisp:CurrentChoice,
                                           Actions ) )
    Then {
        SetValue( {
                ActionDisp:CurrentChoice;
                }, Acknowledge, TRUE );
        If Member?( SingleListBox1:AllowableValues, ActionDisp:CurrentChoice )
         Then RemoveFromList( SingleListBox1:AllowableValues,
                 ActionDisp:CurrentChoice );
        ResetImage( SingleListBox1 );
        ResetImage( StateBox1 );
        };
   Next( );
   } );

      /************************************
       **** FUNCTION: CloseAWin
       **** This function is called by the cancel button in the action window.
       ************************************/
MakeFunction( CloseAWin, [],
   HideWindow( Session3 ) );
SetFunctionComment( CloseAWin, "This function is called by the cancel button in the action window." );

      /************************************
       **** FUNCTION: Next
       ************************************/
MakeFunction( Next, [],
   {
   ActionDisp:ChoicePos = GetElemPos( ActionDisp:Actions, ActionDisp:CurrentChoice );
```

```
    If ( ( ActionDisp:ChoicePos + 1 ) > LengthList( ActionDisp:Actions ) )
      Then ActionDisp:ChoicePos = 0;
    If Not( LengthList( ActionDisp:Actions ) == 0 )
      Then ActionDisp:CurrentChoice = GetNthElem( ActionDisp:Actions,
                                        ActionDisp.ChoicePos
                                          + 1 );
    } );

      /*************************************
       **** FUNCTION: Prev
       *************************************/
MakeFunction( Prev, [],
    {
    ActionDisp:ChoicePos = GetElemPos( ActionDisp:Actions, ActionDisp:CurrentChoice );
    If ( ( ActionDisp:ChoicePos - 1 ) < 1 )
      Then ActionDisp:ChoicePos = LengthList( ActionDisp:Actions )
             + 1;
    ActionDisp:CurrentChoice = GetNthElem( ActionDisp:Actions,
                                 ActionDisp:ChoicePos - 1 );
    } );

      /*************************************
       **** FUNCTION: Obj
       *************************************/
MakeFunction( Obj, [Item Variable Priority Variation Critical_Time Repeat_Time Wait Message
Rulename RuleNo],
    SendMessage( Item, Request, Variable, Priority, Variation,
       Critical_Time, Repeat_Time, Wait, 0, Rulename, ACTIVE,
       Message, RuleNo ) );

      /*************************************
       **** FUNCTION: HBRead
       *************************************/
MakeFunction( HBRead, [Tag Num],
    {} );

      /*************************************
       **** FUNCTION: HBWrite
       *************************************/
MakeFunction( HBWrite, [Tag Num],
    {} );

      /*************************************
       **** FUNCTION: HBErrorMsg
       *************************************/
MakeFunction( HBErrorMsg, [],
    {} );

      /*************************************
       **** FUNCTION: HBError
       *************************************/
MakeFunction( HBError, [],
```

```
    0 );

        /*************************************
         **** FUNCTION: HBClose
         *************************************/
MakeFunction( HBClose, [],
    {} );

        /*************************************
         **** FUNCTION: HBOpen
         *************************************/
MakeFunction( HBOpen, [Num],
    {} );

        /*************************************
         **** FUNCTION: ShowInstanceMenu
         *************************************/
MakeFunction( ShowInstanceMenu, [obj],
    PopupMenu( ItemEdit, obj, BROWSER ) );

        /*************************************
         **** FUNCTION: ShowClassMenu
         *************************************/
MakeFunction( ShowClassMenu, [obj],
    PopupMenu( ItemEdit3, obj, BROWSER ) );

        /*************************************
         **** FUNCTION: HBType
         *************************************/
MakeFunction( HBType, [],
    Analog );



/*****************************************************************/
/**        ALL CLASSES ARE SAVED BELOW          **/
/*****************************************************************/

        /*************************************
         **** CLASS: Menu
         *************************************/
Menu:X = NULL;
Menu:Y = NULL;
SetSlotOption( Menu:Choices, MULTIPLE );
ClearList( Menu:Choices );
SetSlotOption( Menu:ChoiceNames, MULTIPLE );
ClearList( Menu:ChoiceNames );
SetSlotOption( Menu:Checked, MULTIPLE );
ClearList( Menu:Checked );
SetSlotOption( Menu:Disabled, MULTIPLE );
ClearList( Menu:Disabled );
Menu:DefaultMethod = DisplayCall;
```

```
/************************************
 **** CLASS: Links
 **** This object links actions to nodes. The Links contain
 ****  the knowlegde that links the actons to a particular node.
 ************************************/
MakeClass( Links, Root );
SetClassComment( Links, "This object links actions to nodes. The Links contain
 the knowlegdo that links the actons to a particular node." );

/************** METHOD: Set  **************/
MakeMethod( Links, Set, [ASI API ASV APV ATREL AVSREL AVPREL MIN MAX AWAIT ACaller
],
   {
   Self:Wait = AWAIT;
   Self:PItem = API;
   Self:PVariable = APV;
   Self:SItem = ASI;
   Self:SVariable = ASV;
   Self:VPRelation = AVPREL;
   Self:VSRelation = AVSREL;
   Self:TRelation = ATREL;
   Self:Creator = ACaller;
   Self:MinVar = MIN;
   Self:MaxVar = MAX;
   } );

/************** METHOD: Reply  **************/
MakeMethod( Links, Reply, [State ],
   {
   If ( State #= FAIL Or State #= SUCCESS )
     Then {
         If ( State #= FAIL )
           Then {
               Self:Failures += 1;
               If ( Self:Failures > 1000 )
                 Then {
                     Self:Successes = Self:Successes / 100;
                     Self:Failures = Self:Failures / 100;
                     };
               Global:Failures += 1;
               If ( Global:Failures > 1000 )
                 Then {
                     Global:Successes = Global:Successes /
                        100;
                     Global:Failures = Global:Failures / 100;
                     };
               Self:Status = FAILED;
               Self:Called_Time = GetTime( );
               };
         If ( State #= SUCCESS )
           Then {
```

```
            Self:Successes += 1;
            If ( Self:Successes > 1000 )
             Then {
                 Self:Successes = Self:Successes / 1000;
                 Self:Failures = Self:Failures / 1000;
                 };
            Global:Successes += 1;
            If ( Global:Successes > 1000 )
             Then {
                 Global:Successes = Global:Successes /
                    1000;
                 Global:Failures = Global:Failures / 1000;
                 };
            Self:Status = READY;
            Self:Called_Time = GetTime( );
            };
        }
     Else {
        If ( State #= FOUND )
         Then {
            SendMessage( Self:Caller, Reply, Self, FOUND );
            Self:Successes += 1;
            }
         Else {
            If ( State #= NOUSE )
             Then {
                 SendMessage( Self, Init, NORM^* ^.
                 Self:Failures += 1;
                 Self:Status = FAILED;
                 Self:Called_Time = GetTime( );
                 }
              Else PostError( "Incorrect state in " # Self
                           # ":Reply" );
            };
        };
   } );

/************** METHOD: Reset  **************/
MakeMethod( Links, Reset, [],
   {
   If CompTime( GetTime( ), Self:Called_Time + Self:Delay )
    Then SendMessage( Self, Init );
   } );

/************** METHOD: Expand  **************/
/* "This function is called from the control loop when the link
must expand itself." */
MakeMethod( Links, Expand, [],
   {
   RemoveFromList( Global:UnexpandedList, Self );
   If Instance?( Self:PItem )
    Then ( Self:Node = SendMessage( Self:PItem, Request, Self:PVariable,
```

```
                    GetValue( GetValue( Self:Caller ),
                       Priority ), Self:Variation, Self:Critical_Time
                                                    -
                                                    10,
                    Self:Hold_Time, Self:Wait, Self:TimeLSP,
                    Self, LOOK, NULL, 0 ) )
      Else {
          Self:Node = Dummy;
          SendMessage( Explanation, Error, "Item " # Self:PItem
                                    # " no longer exists!" );
          };
    If ( Self:Node #= USE )
      Then {
          AppendToList( Global:ExpandedList, Self );
          Self:Node = GetValue( {
                            Self:PItem;
                            }, NodeName );
          SendMessage( Self:Caller, Reply, Self, USE );
          Self:Temp = FOUND;
          }
      Else {
          If Instance?( Self:Node )
           Then {
              AppendToList( Global:ExpandedList, Self );
              Self:Temp = SendMessage( Self:Node, Seek, Self:NodeCostE );
              If ( Self:Temp Or Self:Temp #= FOUND )
               Then {}
               Else {
                  SendMessage( Self, Reply, NOUSE );
                  Self:Temp = FALSE;
                  };
              }
           Else {
              SendMessage( Self, Reply, NOUSE );
              Self:Temp = FALSE;
              };
          };
    Self:Temp;
    } );
SetMethodComment(Links, Expand, "This function is called from the control loop when the link
must expand itself." );

/****  ********* METHOD: Call  **************/
MakeMethod( Links, Call, [ACaller AItem AVariable AVariation Crt_Time Cost ],
    {
    If ( CompTime( GetTime( ), Self:Called_Time + Global:StdDelay )
          And ( Self:Status #= FAILED ) )
      Then Self:Status = READY;
    If ( ( AItem #= Self:SItem ) And ( AVariable #= Self:SVariable )
          And ( Self:Status #= READY ) And ( AVariation > Self:MinVar
                                        And AVariation
                                        < Self:MaxVar ) )
```

```
    Then {
        Self:Critical_Time = Crt_Time;
        SendMessage( Self, Eval, AVariation );
        Self:Caller = ACaller;
        Self:CostS = 0;
        Assert( Self:PItem );
        ForwardChain( NULL, Rule:CostRules );
        Self:CostDelay = CostTime( Crt_Time, Self:TimeLSP );
        Self:CostVar = CostVar( Self:SVariation, AVariation );
        Self:CostFail = CostFail( Self:Successes, Self:Failures );
        Self:CostP = CostAdd( CostAdd( Self:CostDelay, Self:CostVar ),
                    Self:CostS );
        Self:NodeCostE = CostAdd( Self:CostP, Cost );
        Self:ECost = CostAdd( Self:CostFail, Self:EstimatedCost );
        Self:ECost = CostAdd( CostAdd( CostAdd( CostAdd( Self:ECost,
                                        Cost ),
                                    Self:CostDelay ),
                                Self:CostVar ), Self:CostS );
        Self:Critical_Time = Crt_Time - Self:TimeLSP;
        Self:Hold_Time = Self:Critical_Time;
        If ( Self:CostS < 1 )
          Then {
            Self:Status = UNEXPANDED;
            AppendToList( Global, UnexpandedList, Self );
            SendMessage( Explanation, Display,
              FormatValue( "  - The " # Self:SVariable #
                          " of " # Self:SItem # " may be changed by changing the "
                          # Self:PVariable # " of\n          "
                          # Self:PItem # " by  %3.2f . Orginal value: %3.2f. Link: %s .",
                Self:Variation, GetValue( {
                                        Self:PItem;
                                        },
                                    {
                                    Self:PVariable;
                                    } ), Self ) );
            SendMessage( Explanation, Display,
              FormatValue( "      Node %1.3f, Estimated %1.3f , Var %1.3f, Delay %1.3f, \n
Fail %1.3f, Rules %1.3f, Total Cost %1.3f",
                    Cost, Self:EstimatedCost, Self:CostVar,
                    Self:CostDelay, Self:CostFail, Self:CostS,
                    Self:ECost ) );
            TRUE;
            }
          Else FALSE;
        }
      Else {
        FALSE;
        };
    } );

/************** METHOD: Eval  **************/
/* "This function uses the formula provided in the Link Rule to
```

```
 determine the delay between two variables and the variation
required of the primary variable to cause the desired change." */
MakeMethod( Links, Eval, [Val ],
   {
   Self:V = Val;
   Self:PX = SendMessage( Self:PItem, GetVal, Self:PVariable,
           VAR );
   Self:SX = SendMessage( Self:SItem, GetVal, Self:SVariable,
           VAR );
   Self:TimeLSP = Abs( EvaluateKAL( EvaluateKAL( "Let [v " # Self
                                        # ":V]
   [px "
                                        # Self #
                                        ":PX]
   [sx "
                                        # Self #
                                        ":SX]
   [crt "
                                        # Self #
                                        ":Critical_Time] {"
                                        # Self:TRelation
                                        # "; }; " )
                                # "; " ) );
   If ( Self:TimeLSP == 0 )
     Then Self:TimeLSP = 0.1;
   Self:Variation = EvaluateKAL( "Let [v " # Self # ":V]
   [px "
                           # Self # ":PX]
   [sx " #
                           Self # ":SX]
   [t " # Self
                           # ":TimeLSP] {" # Self:VPRelation
                           # "; }; " );
   Self:SVariation = EvaluateKAL( "Let [v " # Self # ":V]
   [px "
                            # Self # ":PX]
   [sx "
                            # Self # ":SX]
   [t " #
                            Self # ":TimeLSP] {" # Self:VSRelation
                            # "; }; " );
   } );
SetMethodComment(Links, Eval, "This function uses the formula provided in the Link Rule to
 determine the delay between two variables and the variation
required of the primary variable to cause the desired change." );

/************** METHOD: Init  **************/
/* "This function initilizes the link. It will also clear any expanded
 nodes linked to it." */
MakeMethod( Links, Init, [AStatus ],
   {
   If ( AStatus #= CLEAR Or AStatus #= NORMAL Or AStatus #= CLEARDOWN )
```

```
    Then {
        If ( ( AStatus #= NORMAL ) And Not( Null?( Self:Caller ) ) )
          Then SendMessage( Self:Caller, Reply, Self, NOLINK );
        If ( AStatus #= CLEARDOWN )
          Then If Not( Null?( Self:Node ) )
                 Then SendMessage( Self:Node, End, Self, CLEARDOWN );
        If Member?( Global:UnexpandedList, Self )
          Then RemoveFromList( Global:UnexpandedList, Self );
        If Member?( Global:ExpandedList, Self )
          Then RemoveFromList( Global:ExpandedList, Self );
        Self:Status = READY;
        Self:Node = NULL;
        Self:Caller = NULL;
        ResetValue( Self:Called_Time );
        ResetValue( Self:ECost );
        ResetValue( Self:NodeCostE );
        ResetValue( Self:TimeLSP );
        ResetValue( Self:Critical_Time );
        ResetValue( Self:Variation );
        }
      Else PostError( "Incorrect status call to " # Self # ":Init" );
    } );
SetMethodComment(Links, Init, "This function initilizes the link. It will also clear any expanded
 nodes linked to it." );

/************** METHOD: Update **************/
MakeMethod( Links, Update, [ACost ],
    {
    Self:EstimatedCost = ( Self:EstimatedCost + ACost )
       / 2;
    } );

/************** METHOD: AltCost **************/
MakeMethod( Links, AltCost, [Cost ],
    {
    Self:CostS = CostAdd( Self:CostS, Cost );
    } );
MakeSlot( Links:TimeLSP );
SetSlotOption( Links:TimeLSP, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:TimeLSP, VALUE_TYPE, NUMBER );
SetSlotOption( Links:TimeLSP, MINIMUM_VALUE, 0 );
Links:TimeLSP = 0;
SetSlotOption( Links:TimeLSP, IF_NEEDED, NULL );
SetSlotOption( Links:TimeLSP, WHEN_ACCESS, NULL );
SetSlotOption( Links:TimeLSP, BEFORE_CHANGE, NULL );
SetSlotOption( Links:TimeLSP, AFTER_CHANGE, NULL );
MakeSlot( Links:Variation );
SetSlotOption( Links:Variation, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Variation, VALUE_TYPE, NUMBER );
Links:Variation = 0;
SetSlotOption( Links:Variation, IF_NEEDED, NULL );
SetSlotOption( Links:Variation, WHEN_ACCESS, NULL );
```

```
SetSlotOption( Links:Variation, BEFORE_CHANGE, NULL );
SetSlotOption( Links:Variation, AFTER_CHANGE, NULL );
MakeSlot( Links:PItem );
SetSlotOption( Links:PItem, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:PItem, VALUE_TYPE, OBJECT );
SetSlotOption( Links:PItem, ALLOWABLE_CLASSES, Items );
SetSlotOption( Links:PItem, IF_NEEDED, NULL );
SetSlotOption( Links:PItem, WHEN_ACCESS, NULL );
SetSlotOption( Links:PItem, BEFORE_CHANGE, NULL );
SetSlotOption( Links:PItem, AFTER_CHANGE, NULL );
MakeSlot( Links:PVariable );
SetSlotOption( Links:PVariable, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:PVariable, IF_NEEDED, NULL );
SetSlotOption( Links:PVariable, WHEN_ACCESS, NULL );
SetSlotOption( Links:PVariable, BEFORE_CHANGE, NULL );
SetSlotOption( Links:PVariable, AFTER_CHANGE, NULL );
MakeSlot( Links:Successes );
SetSlotOption( Links:Successes, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Successes, VALUE_TYPE, NUMBER );
SetSlotOption( Links:Successes, MINIMUM_VALUE, 0 );
Links:Successes = 100;
MakeSlot( Links:Failures );
SetSlotOption( Links:Failures, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Failures, VALUE_TYPE, NUMBER );
SetSlotOption( Links:Failures, MINIMUM_VALUE, 0 );
Links:Failures = 100;
MakeSlot( Links:EstimatedCost );
SetSlotComment( Links:EstimatedCost, "This contains the latest estimate of the cost of using the link.
" );
SetSlotOption( Links:EstimatedCost, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:EstimatedCost, VALUE_TYPE, NUMBER );
SetSlotOption( Links:EstimatedCost, MINIMUM_VALUE, -1 );
SetSlotOption( Links:EstimatedCost, MAXIMUM_VALUE, 1 );
Links:EstimatedCost = 0;
MakeSlot( Links:VPRelation );
SetSlotOption( Links:VPRelation, NO_AUTO_ASK, TRUE );
Links:VPRelation = *;
SetSlotOption( Links:VPRelation, IF_NEEDED, NULL );
SetSlotOption( Links:VPRelation, WHEN_ACCESS, NULL );
SetSlotOption( Links:VPRelation, BEFORE_CHANGE, NULL );
SetSlotOption( Links:VPRelation, AFTER_CHANGE, NULL );
MakeSlot( Links:SItem );
SetSlotOption( Links:SItem, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:SItem, VALUE_TYPE, OBJECT );
SetSlotOption( Links:SItem, ALLOWABLE_CLASSES, Items );
SetSlotOption( Links:SItem, IF_NEEDED, NULL );
SetSlotOption( Links:SItem, WHEN_ACCESS, NULL );
SetSlotOption( Links:SItem, BEFORE_CHANGE, NULL );
SetSlotOption( Links:SItem, AFTER_CHANGE, NULL );
MakeSlot( Links:SVariable );
SetSlotOption( Links:SVariable, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:SVariable, IF_NEEDED, NULL );
```

```
SetSlotOption( Links:SVariable, WHEN_ACCESS, NULL );
SetSlotOption( Links:SVariable, BEFORE_CHANGE. NULL );
SetSlotOption( Links:SVariable, AFTER_CHANGE, NULL );
MakeSlot( Links:Status );
SetSlotOption( Links:Status, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Status, ALLOWABLE_VALUES, READY, ACTIVE, UNEXPANDED,
EXPANDED, FAILED );
Links:Status = READY;
MakeSlot( Links:Called_Time );
SetSlotOption( Links:Called_Time, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Called_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Links:Called_Time, MINIMUM_VALUE, 0 );
Links:Called_Time = 0;
SetSlotOption( Links:Called_Time, IF_NEEDED, NULL );
SetSlotOption( Links:Called_Time, WHEN_ACCESS, NULL );
SetSlotOption( Links:Called_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Links:Called_Time, AFTER_CHANGE, NULL );
MakeSlot( Links:TRelation );
SetSlotOption( Links:TRelation, NO_AUTO_ASK, TRUE );
Links:TRelation = 1;
SetSlotOption( Links:TRelation, IF_NEEDED, NULL );
SetSlotOption( Links:TRelation, WHEN_ACCESS, NULL );
SetSlotOption( Links:TRelation, BEFORE_CHANGE, NULL );
SetSlotOption( Links:TRelation, AFTER_CHANGE, NULL );
MakeSlot( Links:Caller );
MakeSlot( Links:ECost );
SetSlotComment( Links:ECost, "This is a combination of the cost to get to the link
and the estimate cost to carry on further. This cost will
indicate if the link is good to use or not." );
SetSlotOption( Links:ECost, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:ECost, VALUE_TYPE, NUMBER );
SetSlotOption( Links:ECost, MINIMUM_VALUE, -1 );
SetSlotOption( Links:ECost, MAXIMUM_VALUE, 1 );
Links:ECost = 0;
MakeSlot( Links:NodeCostE );
SetSlotComment( Links:NodeCostE, "This cost element is the combination of the cost of the node
that called it, thus far and success rate. It also contains the
cost of the variation and time as well as link cost rules. This
cost" );
SetSlotOption( Links:NodeCostE, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:NodeCostE, VALUE_TYPE, NUMBER );
SetSlotOption( Links:NodeCostE, MINIMUM_VALUE, -1 );
SetSlotOption( Links:NodeCostE, MAXIMUM_VALUE, 1 );
Links:NodeCostE = 0;
MakeSlot( Links:Critical_Time );
SetSlotOption( Links:Critical_Time, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Critical_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Links:Critical_Time, MINIMUM_VALUE, 0 );
Links:Critical_Time = 0;
SetSlotOption( Links:Critical_Time, IF_NEEDED, NULL );
SetSlotOption( Links:Critical_Time, WHEN_ACCESS, NULL );
SetSlotOption( Links:Critical_Time, BEFORE_CHANGE, NULL );
```

```
SetSlotOption( Links:Critical_Time, AFTER_CHANGE, NULL );
MakeSlot( Links:Node );
SetSlotOption( Links:Node, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:Node, VALUE_TYPE, OBJECT );
SetSlotOption( Links:Node, ALLOWABLE_CLASSES, Nodes );
SetSlotOption( Links:Node, IF_NEEDED, NULL );
SetSlotOption( Links:Node, WHEN_ACCESS, NULL );
SetSlotOption( Links:Node, BEFORE_CHANGE, NULL );
SetSlotOption( Links:Node, AFTER_CHANGE, NULL );
MakeSlot( Links:Temp );
MakeSlot( Links:Creator );
MakeSlot( Links:X );
SetSlotOption( Links:X, IF_NEEDED, NULL );
SetSlotOption( Links:X, WHEN_ACCESS, NULL );
SetSlotOption( Links:X, BEFORE_CHANGE, NULL );
SetSlotOption( Links:X, AFTER_CHANGE, NULL );
MakeSlot( Links:T );
SetSlotOption( Links:T, VALUE_TYPE, NUMBER );
Links:T = 0;
SetSlotOption( Links:T, IF_NEEDED, NULL );
SetSlotOption( Links:T, WHEN_ACCESS, NULL );
SetSlotOption( Links:T, BEFORE_CHANGE, NULL );
SetSlotOption( Links:T, AFTER_CHANGE, NULL );
MakeSlot( Links:DV );
SetSlotOption( Links:DV, VALUE_TYPE, NUMBER );
SetSlotOption( Links:DV, IF_NEEDED, NULL );
SetSlotOption( Links:DV, WHEN_ACCESS, GetDV );
SetSlotOption( Links:DV, BEFORE_CHANGE, NULL );
SetSlotOption( Links:DV, AFTER_CHANGE, NULL );
MakeSlot( Links:TempString );
MakeSlot( Links:V );
SetSlotOption( Links:V, VALUE_TYPE, NUMBER );
Links:V = 0;
SetSlotOption( Links:V, IF_NEEDED, NULL );
SetSlotOption( Links:V, WHEN_ACCESS, NULL );
SetSlotOption( Links:V, BEFORE_CHANGE, NULL );
SetSlotOption( Links:V, AFTER_CHANGE, NULL );
MakeSlot( Links:MaxVar );
SetSlotOption( Links:MaxVar, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:MaxVar, VALUE_TYPE, NUMBER );
Links:MaxVar = 100;
SetSlotOption( Links:MaxVar, IF_NEEDED, NULL );
SetSlotOption( Links:MaxVar, WHEN_ACCESS, NULL );
SetSlotOption( Links:MaxVar, BEFORE_CHANGE, NULL );
SetSlotOption( Links:MaxVar, AFTER_CHANGE, NULL );
MakeSlot( Links:MinVar );
SetSlotOption( Links:MinVar, NO_AUTO_ASK, TRUE );
SetSlotOption( Links:MinVar, VALUE_TYPE, NUMBER );
Links:MinVar = -100;
SetSlotOption( Links:MinVar, IF_NEEDED, NULL );
SetSlotOption( Links:MinVar, WHEN_ACCESS, NULL );
SetSlotOption( Links:MinVar, BEFORE_CHANGE, NULL );
```

```
SetSlotOption( Links:MinVar, AFTER_CHANGE, NULL );
MakeSlot( Links:PX );
SetSlotOption( Links:PX, VALUE_TYPE, NUMBER );
SetSlotOption( Links:PX, IF_NEEDED, NULL );
SetSlotOption( Links:PX, WHEN_ACCESS, NULL );
SetSlotOption( Links:PX, BEFORE_CHANGE, NULL );
SetSlotOption( Links:PX, AFTER_CHANGE, NULL );
MakeSlot( Links:SX );
SetSlotOption( Links:SX, VALUE_TYPE, NUMBER );
SetSlotOption( Links:SX, IF_NEEDED, NULL );
SetSlotOption( Links:SX, WHEN_ACCESS, NULL );
SetSlotOption( Links:SX, BEFORE_CHANGE, NULL );
SetSlotOption( Links:SX, AFTER_CHANGE, NULL );
MakeSlot( Links:VSRelation );
Links:VSRelation = v;
MakeSlot( Links:SVariation );
SetSlotOption( Links:SVariation, VALUE_TYPE, NUMBER );
MakeSlot( Links:CostS );
SetSlotComment( Links:CostS, "This is the cost from the link cost rules" );
SetSlotOption( Links:CostS, VALUE_TYPE, NUMBER );
SetSlotOption( Links:CostS, MINIMUM_VALUE, -1 );
SetSlotOption( Links:CostS, MAXIMUM_VALUE, 1 );
Links:CostS = 0;
MakeSlot( Links:CostP );
SetSlotComment( Links:CostP, "This cost element describes how closely the node fits the
desired variable variation and how fast it is. It also contains
a cost element from the link cost rules." );
SetSlotOption( Links:CostP, VALUE_TYPE, NUMBER );
SetSlotOption( Links:CostP, MINIMUM_VALUE, -1 );
SetSlotOption( Links:CostP, MAXIMUM_VALUE, 1 );
Links:CostP = 0;
MakeSlot( Links:CostFail );
SetSlotComment( Links:CostFail, "This is a cost element derived form the success rate of the
 link." );
SetSlotOption( Links:CostFail, VALUE_TYPE, NUMBER );
SetSlotOption( Links:CostFail, MINIMUM_VALUE, -1 );
SetSlotOption( Links:CostFail, MAXIMUM_VALUE, 1 );
Links:CostFail = 0;
MakeSlot( Links:CostDelay );
SetSlotComment( Links:CostDelay, "This is the cost due to the time lag of the link." );
SetSlotOption( Links:CostDelay, VALUE_TYPE, NUMBER );
SetSlotOption( Links:CostDelay, MINIMUM_VALUE, -1 );
SetSlotOption( Links:CostDelay, MAXIMUM_VALUE, 1 );
Links:CostDelay = 0;
MakeSlot( Links:CostVar );
SetSlotOption( Links:CostVar, VALUE_TYPE, NUMBER );
SetSlotOption( Links:CostVar, MINIMUM_VALUE, -1 );
SetSlotOption( Links:CostVar, MAXIMUM_VALUE, 1 );
Links:CostVar = 0;
MakeSlot( Links:Hold_Time );
SetSlotOption( Links:Hold_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Links:Hold_Time, MINIMUM_VALUE, 0 );
```

```
Links:Hold_Time = 0;
MakeSlot( Links:Wait );
SetSlotOption( Links:Wait, VALUE_TYPE, BOOLEAN );
Links:Wait = FALSE;

      /*************************************
       **** CLASS: Copies
       **** This object handles any duplication of the nodes.
       *************************************/
MakeClass( Copies, Root );
SetClassComment( Copies, "This object handles any duplication of the nodes." );

/************ METHOD: Insert  ************/
MakeMethod( Copies, Insert, [Node ],
   {
   If Not( Member?( Self:Copies, Node ) )
     Then AppendToList( Self:Copies, Node );
   SendMessage( Explanation, Display, Node # " has been copied" );
   } );

/************ METHOD: Remove  ************/
MakeMethod( Copies, Remove, [Node ],
   {
   If Member?( Self:Copies, Node )
     Then RemoveFromList( Self:Copies, Node );
   If ( LengthList( Self:Copies ) == 0 )
     Then DeleteInstance( Self );
   } );

/************ METHOD: Ok?  ************/
MakeMethod( Copies, Ok?, [Node ],
   {
   Self:Temp = TRUE;
   EnumList( Self:Copies, N,
      {
      If ( Instance?( N ) And Not( N #= Node ) )
        Then {
            If ( ( N:Status #= ACTIVATED ) Or ( N:Status #=
                                                HOLDING )
                  Or ( N:Status #= LOOKING ) )
              Then {
                  If ( Node:Priority <= N:Priority )
                    Then {
                        SendMessage( N, End, N, TERMINATE );
                        }
                    Else {
                        Self:Temp = FALSE;
                        };
                  }
              Else {};
            }
        Else {};
```

```
    } );
  Self:Temp;
  } );
MakeSlot( Copies:Copies );
SetSlotOption( Copies:Copies, NO_AUTO_ASK, TRUE );
SetSlotOption( Copies:Copies, MULTIPLE );
SetSlotOption( Copies:Copies, VALUE_TYPE, OBJECT );
SetSlotOption( Copies:Copies, ALLOWABLE_CLASSES, Nodes );
ClearList( Copies:Copies );
SetSlotOption( Copies:Copies, IF_NEEDED, NULL );
SetSlotOption( Copies:Copies, WHEN_ACCESS, NULL );
SetSlotOption( Copies:Copies, BEFORE_CHANGE, NULL );
SetSlotOption( Copies:Copies, AFTER_CHANGE, NULL );
MakeSlot( Copies:Temp );
SetSlotOption( Copies:Temp, VALUE_TYPE, BOOLEAN );
Copies:Temp = FALSE;
SetSlotOption( Copies:Temp, IF_NEEDED, NULL );
SetSlotOption( Copies:Temp, WHEN_ACCESS, NULL );
SetSlotOption( Copies:Temp, BEFORE_CHANGE, NULL );
SetSlotOption( Copies:Temp, AFTER_CHANGE, NULL );

    /*********************************  **
     **** CLASS: Actions
     **** The class contains all the actions that can be taken.
     **** This class contains user defined knowledge.
     *********************************/
MakeClass( Actions, Root );
SetClassComment( Actions, "The class contains all the actions that can be taken.
This class contains user defined knowledge." );

/************ METHOD: Init ************/
/* "This method is called during initilization." */
MakeMethod( Actions, Init, [],
  {
  If Member?( Global:ActionList, Self )
    Then RemoveFromList( Global:ActionList, Self );
  If Member?( SingleListBox1:AllowableValues, Self )
    Then RemoveFromList( SingleListBox1:AllowableValues, Self );
  If Member?( ActionDisp:Actions, Self )
    Then RemoveFromList( ActionDisp:Actions, Self );
  ResetImage( SingleListBox1 );
  ResetValue( Self:Caller );
  ResetValue( Self:Called_Time );
  ResetValue( Self:Priority );
  ResetValue( Self:Cost );
  ResetValue( Self:ActivatedTime );
  ResetValue( Self:ActivatedDate );
  Self:Activated = FALSE;
  Self:Status = READY;
  Self:Acknowledge = FALSE;
  } );
SetMethodComment(Actions, Init, "This method is called during initilization." );
```

```
/************** METHOD: Set  **************/
/* "This slot sets Status slot to READY." */
MakeMethod( Actions, Set, [],
   {
   If ( Self:Status #= ACTIVATED )
    Then {
        SendMessage( Self, Terminate );
        Self:Status = READY;
        };
   If Member?( Global:ActionList, Self )
    Then RemoveFromList( Global:ActionList, Self );
   If Member?( SingleListBox1:AllowableValues, Self )
    Then RemoveFromList( SingleListBox1 AllowableValues, Self );
   If Member?( ActionDisp:Actions, Self )
    Then RemoveFromList( ActionDisp:Actions, Self );
   If ( Not( Null?( Global:TempAction ) ) And ( Global:TempAction
                                        #= Self ) )
    Then Global:TempAction = NULL;
   Self:Activated = FALSE;
   ResetImage( SingleListBox1 );
   } );
SetMethodComment(Actions, Set, "This slot sets Status slot to READY." );

/************** METHOD: Activate  **************/
MakeMethod( Actions, Activate, [ACaller APriority ARoot ],
   {
   If ( Self:Status #= READY )
    Then {
        Self:Caller = ACaller;
        Self:Priority = APriority;
        Self:ActivatedTime = Time( );
        Self:ActivatedDate = Date( );
        Self:Status = ACTIVATED;
        Self:Root = ARoot;
        Self:Acknowledge = FALSE;
        If Not( Member?( Global:ActionList, Self ) )
         Then AppendToList( Global:ActionList, Self );
        TRUE;
        };
   Self:Status #= ACTIVATED Or Self:Status #= READY;
   } );

/************** METHOD: Action  **************/
MakeMethod( Actions, Action, [],
   {
   If ( Self:ActionType #= MANUAL )
    Then {
        Global:Response = FALSE;
        If ( Not( Null?( Self:Root ) ) And Instance?( Self:Root ) )
         Then {
            If Write_Data( Self:Tag, Self:ActionNo )
             Then If Write_Data( RULES.X, Self:Root:RuleNo )
```

```
                    Then If Write_Data( ITEMS.X, 0.1 *
                                        Self:Root:Item:ItemNo
                                        +
                                        10
                                        *
                                        SendMessage( Self:Root:Item,
                                           GetVal,
                                           Self:Root:Variable,
                                           NO ) )
                                Then {
                                    Self:Activated = TRUE;
                                    SendMessage( Explanation,
                                       Action, Self, Self:Root );
                                    };
                        }
                    Else {
                        SendMessage( Self, Init );
                        };
                }
            Else {
                Self:Acknowledge = TRUE;
                If ( Self:Mode #= FIXED )
                 Then {
                    SendMessage( Self:Item, SetVal, Self:Variable,
                       Self:AcVariation, VAR );
                    }
                 Else {
                    SendMessage( Self:Item, SetVal, Self:Variable,
                       GetValue( {
                                Self:Item;
                                },
                           {
                           Self:Variable;
                           } ) + Self:AcVariation, VAR );
                    };
                TRUE;
                };
        } );

/************** METHOD: Terminate  **************/
MakeMethod( Actions, Terminate, [],
    {
    SendMessage( Explanation, Add, "  Action " # Self # " has been terminated" );
    } );

/************** METHOD: Reply  **************/
MakeMethod( Actions, Reply, [Message ],
    {
    If ( Self:Status = ACTIVATED )
     Then {
        If ( Message #= SUCCESS )
         Then {
```

```
        If Self:Acknowledge
          Then {
              Self:Successes += 1;
              If ( Self:Successes > 1000 )
                Then {
                    Self:Successes = Self:Successes
                        / 100;
                    Self:Failures = Self:Failures
                        / 100;
                    };
              Global:Successes += 1;
              If ( Global:Failures > 1000 )
                Then {
                    Global:Successes = Global:Successes
                        / 100;
                    Global:Failures = Global:Failures
                        / 100;
                    };
              };
          Self:Called_Time = GetTime( );
          Self:Status = READY;
          SendMessage( Self, Set );
          };
      If ( Message #= FAIL )
        Then {
          If Self:Acknowledge
            Then {
              Self:Failures += 1;
              If ( Self:Failures > 1000 )
                Then {
                    Self:Successes = Self:Successes
                        / 100;
                    Self:Failures = Self:Failures
                        / 100;
                    };
              Global:Failures += 1;
              If ( Global:Successes > 1000 )
                Then {
                    Global:Successes = Global:Successes
                        / 100;
                    Global:Failures = Global:Failures
                        / 100;
                    };
              };
          Self:Status = FAILED;
          Self:Called_Time = GetTime( );
          SendMessage( Self, Set );
          };
      If ( Message #= ACKNOWLEDGE )
        Then {
          Self:Acknowledge = TRUE;
          };
```

```
        };
    } );

/************** METHOD: Precondition  **************/
MakeMethod( Actions, Precondition, [AVariation ],
    {
    If ( CompTime( GetTime( ), Self:Called_Time + Self:Delay )
          And ( Self:Status #= FAILED ) )
     Then SendMessage( Self, Init );
    If ( Self:Status #= FAILED )
     Then {
          Self:Cost = 1;
          }
     Else Self:Cost = CostVar( Self:Variation, AVariation );
    If Not( Self:Cost == 1 )
     Then {
          Self:PreCost = Self:InitCost;
          Assert( Self:Item );
          Assert( Self:Variable );
          ForwardChain( NULL, Rule:ActionRules );
          Self:Cost = CostAdd( Self:Cost, CostFail( Self:Successes,
                                   Self:Failures ) );
          Self:Cost = CostAdd( Self:Cost, Self:PreCost );
          SendMessage( Explanation, Display, FormatValue( "      Rule:  %1.3f, Var: %1.3f, Fail:
%1.3f, Total Cost %1.3f \n      Requested Variation %3.2f",
                                   Self:PreCost,
                                   CostVar( Self:Variation,
                                      AVariation ),
                                   CostFail( Self:Successes,
                                      Self:Failures ),
                                   Self:Cost, AVariation ) );
          }
     Else {
          SendMessage( Explanation, Display, "Action not avaliable." );
          };
    Self:Cost;
    } );

/************** METHOD: Variation  **************/
MakeMethod( Actions, Variation, [],
    {
    If ( Self:Mode #= FIXED )
     Then {
          Convert( Self:Item, Self:Variable, Self:AcVariation )
             - Convert( Self:Item, Self:Variable,
                  GetValue( {
                            Self:Item;
                            },
                       {
                       Self:Variable;
                       } ) );
          }
```

```
    Else Convert( Self:Item, Self:V     'e, Self:AcVariation );
  } );

/************** METHOD: VarType  **************/
MakeMethod( Actions, VarType, [],
  {
  Self:VariableType = CatchError( SendMessage( Self:Item, GetVal,
                          Self:Variable, TYPE ),
                    {
                    SendMessage( Explanation, Error, " Error determining variable type in action "
                                                    #
                                                    Self );
                    Analog;
                    } );
  } );

/************** METHOD: AltCost  **************/
MakeMethod( Actions, AltCost, [Cost ],
  {
  Self:PreCost = CostAdd( Self:PreCost, Cost );
  } );
MakeSlot( Actions:Caller );
SetSlotOption( Actions:Caller, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Caller, VALUE_TYPE, OBJECT );
SetSlotOption( Actions:Caller, ALLOWABLE_CLASSES, Nodes );
SetSlotOption( Actions:Caller, IF_NEEDED, NULL );
SetSlotOption( Actions:Caller, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Caller, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Caller, AFTER_CHANGE, NULL );
MakeSlot( Actions:Cost );
SetSlotComment( Actions:Cost, "Total cost of using an action excluding any time considerations
" );
SetSlotOption( Actions:Cost, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Cost, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:Cost, MINIMUM_VALUE, -1 );
SetSlotOption( Actions:Cost, MAXIMUM_VALUE, 1 );
Actions:Cost = 0;
MakeSlot( Actions:Failures );
SetSlotOption( Actions:Failures, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Failures, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:Failures, MINIMUM_VALUE, 0 );
Actions:Failures = 100;
MakeSlot( Actions:InitCost );
SetSlotOption( Actions:InitCost, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:InitCost, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:InitCost, MINIMUM_VALUE, -1 );
SetSlotOption( Actions:InitCost, MAXIMUM_VALUE, 1 );
Actions:InitCost = 0;
SetSlotOption( Actions:InitCost, IF_NEEDED, NULL );
SetSlotOption( Actions:InitCost, WHEN_ACCESS, NULL );
SetSlotOption( Actions:InitCost, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:InitCost, AFTER_CHANGE, NULL );
```

```
MakeSlot( Actions:Item );
SetSlotOption( Actions:Item, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Item, VALUE_TYPE, OBJECT );
SetSlotOption( Actions:Item, ALLOWABLE_CLASSES, Items );
SetSlotOption( Actions:Item, IF_NEEDED, NULL );
SetSlotOption( Actions:Item, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Item, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Item, AFTER_CHANGE, NULL );
MakeSlot( Actions:Priority );
SetSlotOption( Actions:Priority, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Priority, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:Priority, MINIMUM_VALUE, 0 );
SetSlotOption( Actions:Priority, MAXIMUM_VALUE, 10 );
Actions:Priority = 10;
SetSlotOption( Actions:Priority, IF_NEEDED, NULL );
SetSlotOption( Actions:Priority, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Priority, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Priority, AFTER_CHANGE, NULL );
MakeSlot( Actions:Status );
SetSlotOption( Actions:Status, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Status, ALLOWABLE_VALUES, READY, ACTIVATED, FAILED );
Actions:Status = READY;
MakeSlot( Actions:Successes );
SetSlotOption( Actions:Successes, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Successes, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:Successes, MINIMUM_VALUE, 0 );
Actions:Successes = 100;
MakeSlot( Actions:TimeLSP );
SetSlotOption( Actions:TimeLSP, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:TimeLSP, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:TimeLSP, MINIMUM_VALUE, 0 );
Actions:TimeLSP = 60;
MakeSlot( Actions:Variable );
SetSlotOption( Actions:Variable, IF_NEEDED, NULL );
SetSlotOption( Actions:Variable, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Variable, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Variable, AFTER_CHANGE, VarType );
MakeSlot( Actions:Variation );
SetSlotOption( Actions:Variation, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Variation, WHEN_ACCESS, Variation );
MakeSlot( Actions:Delay );
SetSlotOption( Actions:Delay, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Delay, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:Delay, MINIMUM_VALUE, 0 );
Actions:Delay = 0;
MakeSlot( Actions:Called_Time );
SetSlotOption( Actions:Called_Time, NO_AUTO_ASK, TRUE );
SetSlotOption( Actions:Called_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:Called_Time, MINIMUM_VALUE, 0 );
Actions:Called_Time = 0;
SetSlotOption( Actions:Called_Time, IF_NEEDED, NULL );
SetSlotOption( Actions:Called_Time, WHEN_ACCESS, NULL );
```

```
SetSlotOption( Actions:Called_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Called_Time, AFTER_CHANGE, NULL );
MakeSlot( Actions:ActionMesg );
SetSlotOption( Actions:ActionMesg, IF_NEEDED, NULL );
SetSlotOption( Actions:ActionMesg, WHEN_ACCESS, NULL );
SetSlotOption( Actions:ActionMesg, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:ActionMesg, AFTER_CHANGE, NULL );
MakeSlot( Actions:Mode );
SetSlotOption( Actions:Mode, ALLOWABLE_VALUES, DELTA, FIXED );
Actions:Mode = DELTA;
SetSlotOption( Actions:Mode, IF_NEEDED, NULL );
SetSlotOption( Actions:Mode, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Mode, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Mode, AFTER_CHANGE, NULL );
MakeSlot( Actions:AcVariation );
SetSlotOption( Actions:AcVariation, VALUE_TYPE, NUMBER );
Actions:AcVariation = 0;
SetSlotOption( Actions:AcVariation, IF_NEEDED, NULL );
SetSlotOption( Actions:AcVariation, WHEN_ACCESS, NULL );
SetSlotOption( Actions:AcVariation, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:AcVariation, AFTER_CHANGE, NULL );
MakeSlot( Actions:ActivatedTime );
SetSlotOption( Actions:ActivatedTime, IF_NEEDED, NULL );
SetSlotOption( Actions:ActivatedTime, WHEN_ACCESS, NULL );
SetSlotOption( Actions:ActivatedTime, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:ActivatedTime, AFTER_CHANGE, NULL );
MakeSlot( Actions:ActivatedDate );
MakeSlot( Actions:VariableType );
SetSlotOption( Actions:VariableType, ALLOWABLE_VALUES, Analog, Discrete );
Actions:VariableType = Analog;
SetSlotOption( Actions:VariableType, IF_NEEDED, NULL );
SetSlotOption( Actions:VariableType, WHEN_ACCESS, NULL );
SetSlotOption( Actions:VariableType, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:VariableType, AFTER_CHANGE, NULL );
MakeSlot( Actions:PreCost );
SetSlotOption( Actions:PreCost, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:PreCost, MINIMUM_VALUE, -1 );
SetSlotOption( Actions:PreCost, MAXIMUM_VALUE, 1 );
Actions:PreCost = 0;
SetSlotOption( Actions:PreCost, IF_NEEDED, NULL );
SetSlotOption( Actions:PreCost, WHEN_ACCESS, NULL );
SetSlotOption( Actions:PreCost, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:PreCost, AFTER_CHANGE, NULL );
MakeSlot( Actions:Tag );
Actions:Tag = NONE;
SetSlotOption( Actions:Tag, IF_NEEDED, NULL );
SetSlotOption( Actions:Tag, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Tag, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Tag, AFTER_CHANGE, NULL );
MakeSlot( Actions:ActionNo );
SetSlotOption( Actions:ActionNo, VALUE_TYPE, NUMBER );
SetSlotOption( Actions:ActionNo, MINIMUM_VALUE, 0 );
```

```
Actions:ActionNo = 0;
SetSlotOption( Actions:ActionNo, IF_NEEDED, NULL );
SetSlotOption( Actions:ActionNo, WHEN_ACCESS, NULL );
SetSlotOption( Actions:ActionNo, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:ActionNo, AFTER_CHANGE, NULL );
MakeSlot( Actions:Acknowledge );
SetSlotOption( Actions:Acknowledge, VALUE_TYPE, BOOLEAN );
Actions:Acknowledge = FALSE;
SetSlotOption( Actions:Acknowledge, IF_NEEDED, NULL );
SetSlotOption( Actions:Acknowledge, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Acknowledge, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Acknowledge, AFTER_CHANGE, NULL );
MakeSlot( Actions:Root );
SetSlotOption( Actions:Root, IF_NEEDED, NULL );
SetSlotOption( Actions:Root, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Root, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Root, AFTER_CHANGE, NULL );
MakeSlot( Actions:Activated );
SetSlotOption( Actions:Activated, VALUE_TYPE, BOOLEAN );
SetSlotOption( Actions:Activated, IF_NEEDED, NULL );
SetSlotOption( Actions:Activated, WHEN_ACCESS, NULL );
SetSlotOption( Actions:Activated, BEFORE_CHANGE, NULL );
SetSlotOption( Actions:Activated, AFTER_CHANGE, NULL );
MakeSlot( Actions:ActionType );
SetSlotOption( Actions:ActionType, ALLOWABLE_VALUES, AUTO, MANUAL );
Actions:ActionType = MANUAL;

      /*************************************
       **** CLASS: Default
       *************************************/
MakeClass( Default, Actions );

      /*************************************
       **** CLASS: Nodes
       **** This class is an internal object that is used to find
       ****  the best action for a particular situation.
       *************************************/
MakeClass( Nodes, Root );
SetClassComment( Nodes, "This class is an internal object that is used to find
 the best action for a particular situation." );

/************ METHOD: Init  ************/
/* "Initilization routine that calls clear and then finds the increasing_Actions and decreasing_Actions." */
MakeMethod( Nodes, Init, [AStatus ],
   {
   If ( AStatus #= NORMAL )
    Then {
        If Not( Null?( Self:Calling_Link ) )
          Then SendMessage( Self:Calling_Link, Reply, NOUSE );
        }
    Else {
        If ( AStatus #= CLEARDOWN )
```

```
            Then {
               If Instance?( Self:Activated_Node )
                 Then SendMessage( Self:Activated_Node, End,
                        Self, CLEARDOWN );
               }
            Else {
               If ( AStatus #= TERMINATE )
                 Then {
                    EnumList( Self:Caller, Call,
                       If ( Instance?( Call ) And IsAKindOf?( Call,
                                                  Nodes ) )
                         Then SendMessage( Call, End, Self,
                                TERMINATE ) );
                    }
                 Else {
                    PostError( "Incorrect call to " # Self
                                # ":Init" );
                    };
               };
         };
      If Not( Null?( Self:Applied_Action ) )
        Then SendMessage( Self:Applied_Action, Set );
      EnumList( Self:NewLinks, NL,
         {
         If Instance?( NL )
           Then SendMessage( NL, Init, CLEARDOWN );
         } );
      Self:Applied_Action = NULL;
      Self:Applied_Link = NULL;
      Self:Activated_Node = NULL;
      If Member?( Global:NewSetNodes, Self )
        Then RemoveFromList( Global:NewSetNodes, Self );
      If Member?( Global:SetNodes, Self )
        Then RemoveFromList( Global:SetNodes, Self );
      If Member?( Global:NewLinks, Self )
        Then RemoveFromList( Global:NewLinks, Self );
      If Member?( Global:Activated_Nodes, Self )
        Then RemoveFromList( Global:Activated_Nodes, Self );
      ResetValue( Self:Copies );
      Self:Status = READY;
      Self:BestType = ACTIONS;
      } );
SetMethodComment(Nodes, Init, "Initilization routine that calls clear and then finds the
increasing_Actions and decreasing_Actions." );

/************** METHOD: Set  **************/
/* "See notes. Main procedure for setting up a effect." */
MakeMethod( Nodes, Set, [AItem AVariable APriority AVariation ACritical_Time ARepeatTime Wait
ATimeLSC ACaller AMessage Explanation RuleNo ],
   {
   If ( Self:Status #= UNSOLVED )
     Then {
```

```
        If CompTime( GetTime( ), Self:Called_Time + Self:Repeat_Time )
          Then Self:Status = READY;
        If Opposite?( AVariation, Self:Variation )
          Then Self:Status = READY;
        };
    Self:Item = AItem;
    Self:Variable = AVariable;
    If ( Self:Status #= READY Or Self:Priority > APriority )
      Then {
        If ( ( Self:Status #= READY ) Or ( Self:Status #= SET
                                           And AMessage
                                           #= ACTIVE ) )
          Then {
            SendMessage( Self, Store, APriority, AVariation,
              ACritical_Time, ARepeatTime, Wait, ATimeLSC,
              ACaller, AMessage, Explanation, RuleNo );
            }
          Else {
            If ( Self:Status #= UNSOLVED )
              Then {
                FALSE;
                }
              Else COPY;
            };
        }
      Else {
        If ( ( APriority == Self:Priority ) And Not( Member?( Self:Caller,
                                                   ACaller ) ) )
          Then {
            If ( Self:Status #= ACTIVATED Or Self:Status
                  #= HOLDING )
              Then {
                USE;
                }
              Else {
                If ( Self:Status #= SET )
                  Then SendMessage( Self, Store, APriority,
                         AVariation, ACritical_Time,
                         ARepeatTime, Wait, ATimeLSC,
                         ACaller, AMessage, Explanation,
                         RuleNo );
                };
            }
          Else {
            FALSE;
            };
        };
    } );
SetMethodComment(Nodes, Set, "See notes. Main procedure for setting up a effect." );

/************ METHOD: Activate ************/
/* "Called to activate the rule finding procedures." */
```

```
MakeMethod( Nodes, Activate, [Root ],
  {
  If Not( Null?( Self:Copies ) )
   Then {
       Self:Temp = SendMessage( Self:Copies, Ok?, Self );
       }
   Else {
       Self:Temp = TRUE;
       };
  If Self:Temp
   Then {
       If ( Self:BestType #= LINK And Self:Status #= USE )
        Then {
            If SendMessage( Self:Activated_Node, Use, Self )
             Then {
                 Self:Status = HOLDING;
                 Self:TimeLBS = Self:Applied_Link:TimeLSP;
                 Self:Variation = GetValue( GetValue( Self:Applied_Link ),
                                  SVariation );
                 Self:CostP = Self:Applied_Link:CostP;
                 Self:CostPT = Self:Activated_Node:CostST;
                 Self:CostST = CostAdd( Self:CostP, Self:CostPT );
                 If Not( Null?( Self:Calling_Link ) )
                  Then {
                      SendMessage( Self:Calling_Link,
                         Update, Self:CostST );
                      };
                 If Not( Member?( Global:Activated_Nodes,
                          Self ) )
                  Then AppendToList( Global:Activated_Nodes,
                         Self );
                 TRUE;
                 };
            }
        Else {
            If ( Self:BestType #= ACTION And Self:Status
                 #= FOUND )
             Then {
                 Self:Applied_Action = Self:Best;
                 Self:Applied_Link = NULL;
                 Self:Activated_Node = NULL;
                 If Instance?( Self:Applied_Action )
                  Then {
                      If Not( SendMessage( Self:Applied_Action,
                                Activate, Self, Self:Priority,
                                Root ) )
                       Then {
                           SendMessage( Explanation,
                              Add, FormatValue( " Could not activate the action "
                                                #
                                                Self:Applied_Action ) );
                           SendMessage( Self, End,
```

```
                    Self, NORMAL );
                FALSE;
                }
            Else {
                Self:Status = ACTIVATED;
                Self:Variation = GetValue( GetValue( Self:Applied_Action ),
                                Variation );
                Self:Calling_Time = GetTime( );
                AppendToList( Global:Activated_Nodes,
                    Self );
                SendMessage( Explanation,
                    Add, FormatValue( " -  The action "
                                #
                                Self:Applied_Action
                                #
                                " was found, which will cause the "
                                #
                                Self:Variable
                                #
                                " of \n          "
                                #
                                Self:Item
                                #
                                " to change by %3.2f. The original value is %3.2f
units.\n
    The critical time is %3.2f",
                                Self:Variation,
                                Self:OriginalValue,
                                Self:Critical_Time ) );
                SendMessage( Explanation,
                    Display, FormatValue( " Node "
                                #
                                Self
                                #
                                " to be solved after %3.2f sec",
                            Self:TimeLBS ) );
                If Not( Null?( Self:Calling_Link ) )
                  Then {
                      SendMessage( Self:Calling_Link,
                          Update, Self:CostST );
                      };
                TRUE;
                };
            };
        }
    Else {
        If ( Self:BestType #= LINK And Self:Status
                #= FOUND )
          Then {
              If SendMessage( Self:Activated_Node,
                  Activate, Root )
                Then {
```

```
                                If Not( Member?( Global:Activated_Nodes,
                                        Self ) )
                                  Then AppendToList( Global:Activated_Nodes,
                                        Self );
                                Self:Status = HOLDING;
                                Self.TimeLBS = Self:Applied_Link:TimeLSP;
                                Self:Variation = GetValue( GetValue( Self:Applied_Link ),
                                        SVariation );
                                Self:CostP = Self:Applied_Link:CostP;
                                Self:CostPT = Self:Activated_Node:CostST;
                                Self:CostST = CostAdd( Self:CostP,
                                        Self:CostPT );
                                SendMessage( Explanation,
                                   Display, FormatValue( " Critical time for "
                                                #
                                                Self
                                                #
                                                " is %3.2f",
                                        Self:Critical_Time ) );
                                If Not( Null?( Self:Calling_Link ) )
                                  Then {
                                     SendMessage( Self:Calling_Link,
                                        Update, Self:CostST );
                                     };
                                TRUE;
                                }
                             Else {
                                SendMessage( Explanation,
                                   Display, "Could not activate "
                                        # Self:Activated_Node
                                        # . );
                                SendMessage( Self, NoLinks );
                                FALSE;
                                };
                          };
                     };
                };
          }
       Else {
          SendMessage( Explanation, Display, "Could not activate "
                        # Self # " . More important copies exist." );
          SendMessage( Self, End, Self, NORMAL );
          FALSE;
          };
    } );
SetMethodComment(Nodes, Activate, "Called to activate the rule finding procedures." );

/************ METHOD: GetAction  ************/
/* "See notes for functional description." */
MakeMethod( Nodes, GetAction, [],
   {
   Self:Best = NULL;
```

```
    Self:BestCost = Self:MaxCost;
    Self:BestType = ACTION;
    ClearList( Self:Valid_Actions );
    If Class?( A_ # Self:Item )
      Then GetInstanceList( A_ # Self:Item, Self:Valid_Actions );
    EnumList( Self:Valid_Actions, VA,
       {
       If ( VA:Item #= Self:Item And VA:Variable #= Self:Variable )
        Then {
            Self:TempCost = SendMessage( VA, Precondition, Self:Variation );
            If ( ( Self:TempCost < Self:MaxCost )
                  And ( Self:TempCost < 1 ) )
             Then {
                Self:TempCost = CostAdd( Self:TempCost, CostTime( Self:Critical_Time,
                                                    VA:TimeLSP ) );
                };
            If ( Self:TempCost < 1.1 )
             Then SendMessage( Explanation, Display,
                    FormatValue( "  - Considering action "
                                 # VA # " to change the "
                                 # Self:Variable # " of "
                                 # Self:Item # ". \n       The delay cost is %1.3f cost of the action is
%1.3f.",
                        CostTime( Self:Critical_Time, VA:TimeLSP ),
                        Self:TempCost ) );
            If ( Self:TempCost < Self:BestCost )
             Then {
                Self:Best = VA;
                Self:BestCost = Self:TempCost;
                Self:CostP = VA:Cost;
                Self:CostST = Self:TempCost;
                Self:TimeLBS = VA:TimeLSP;
                };
            };
       } );
    Not( Null?( Self:Best ) );
    } );
SetMethodComment(Nodes, GetAction, "See notes for functional description." );

/************** METHOD: Fail  **************/
MakeMethod( Nodes, Fail, [],
    {
    SendMessage( Explanation, Add, "  - The " # Self:Variable #
                                " of " # Self:Item # " has not changed satisfactorily!" );
    If Not( Self:Wait )
     Then {
        If Not( Null?( Self:Applied_Link ) )
         Then {
            SendMessage( Self:Applied_Link, Reply, FAIL );
            };
        If Not( Null?( Self:Applied_Action ) )
         Then {
```

```
            SendMessage( Self:Applied_Action, Reply, FAIL );
            };
        };
    If Member?( Global:Activated_Nodes, Self )
      Then RemoveFromList( Global:Activated_Nodes, Self );
    SendMessage( Self, End, Self, TERMINATE );
    } );

/************** METHOD: Reset  **************/
MakeMethod( Nodes, Reset, [],
    {
    If ( CompTime( GetTime( ), Self:Repeat_Time + Self:Called_Time )
          And ( Self:Wait #= FALSE ) )
      Then {
          SendMessage( Self, Fail );
          }
      Else {
          If ( ( Self:Status #= ACTIVATED ) Or ( Self:Status
                                          #= HOLDING ) )
            Then {
              If ( Self:Wait #= TRUE )
                Then
                    Self:Boolean = CompTime( GetTime( ),
                                  Self:Repeat_Time +
                                    Self:Called_Time );
                  }
                Else Self:Boolean = ( ( ( Self:Variation <
                                    0 ) And SendMessage( Self:Item,
                                              GetVal,
                                              Self:Variable,
                                              VAR )
                                    <= Self:OriginalValue
                                    + Self:Variation
                                    * 0.75 ) Or ( ( Self:Variation
                                                    >
                                                    0 )
                                                  And
                                                  SendMessage( Self:Item,
                                                    GetVal,
                                                    Self:Variable,
                                                    VAR )
                                                  >=
                                                  Self:OriginalValue
                                                  +
                                                  Self:Variation
                                                  *
                                                  0.75 ) );
              If ( Not( Null?( Self:Boolean ) )
                    And Self:Boolean )
                Then {
                    SendMessage( Self, Success );
                    }
```

```
                Else {
                    If ( CompTime( GetTime( ),
                            Self:Calling_Time + Self:TimeLBS )
                            And ( Self:Status #= ACTIVATED )
                            And Not( Self:Wait ) )
                      Then {
                          SendMessage( Self, Fail );
                          }
                      Else {};
                    };
                };
            };
    } );

/*** ********** METHOD: End  **********/
MakeMethod( Nodes, End, [ACaller AStatus ],
    {
    If ( AStatus #= CLEARDOWN )
     Then If Member?( Self:Caller, ACaller )
            Then RemoveFromList( Self:Caller, ACaller );
    If ( LengthList( Self:Caller ) == 0 Or Not( AStatus #= CLEARDOWN )
            Or ACaller #= Self )
     Then {
          If Not( Null?( Self:Copies ) )
           Then SendMessage( Self:Copies, Remove, Self );
          SendMessage( Self, Init, AStatus );
          DeleteInstance( Self );
          };
    } );

/********** METHOD: Reply  **********/
MakeMethod( Nodes, Reply, [ACaller AStatus ],
    {
    If ( AStatus #= NOLINK )
     Then {
          If Member?( Self:NewLinks, ACaller )
           Then RemoveFromList( Self:NewLinks, ACaller );
          If ( Self:Status #= PENDING Or Self:Status #= LOOKING )
           Then SendMessage( Self, NoLinks );
          }
     Else {
          If ( AStatus #= FOUND )
           Then {
               Self:Status = FOUND;
               Self:Applied_Link = ACaller;
               Self:Activated_Node = ACaller:Node;
               Self:TimeLBS = ACaller:TimeLSP;
               If ( Instance?( GetNthElem( Self:Caller, i ) )
                       And IsAKindOf?( GetNthElem( Self:Caller,
                                   1 ), Nodes ) )
                 Then SendMessage( Self:Calling_Link, Reply,
                         FOUND );
```

```
            }
          Else {
            If ( AStatus #= SUCCESS )
              Then {
                Self:Calling_Time = GetTime( );
                Self:Status = ACTIVATED;
                If Not( Member?( Global:Activated_Nodes,
                        Self ) )
                  Then AppendToList( Global:Activated_Nodes,
                         Self );
                }
              Else If ( AStatus #= USE )
                   Then {
                     Self:Status = USE;
                     Self:Applied_Link = ACaller;
                     Self:Activated_Node = ACaller:Node;
                     Self:TimeLBS = ACaller:TimeLSP;
                     If ( Instance?( GetNthElem( Self:Caller,
                                     1 ) ) And
                          IsAKindOf?( GetNthElem( Self:Caller,
                                      1 ), Nodes ) )
                       Then SendMessage( Self:Calling_Link,
                              Reply, FOUND );
                     }
                   Else PostError( "Incorrect call to "
                                   # Self # ":Reply" );
            };
        };
    } );

/************** METHOD: NoSolution  **************/
MakeMethod( Nodes, NoSolution, [],
    {
    SendMessage( Explanation, Add, "No method could be found to alter the "
                                   # Self:Variable # " of "
                                   # Self:Item # "
      or it did not change with in the avaliable time" );
    Assert( Self:Item );
    ForwardChain( NULL, Rule:DefaultRules );
    SendMessage( Self, Init, CLEARDOWN );
    Self:Called_Time = GetTime( );
    Self:Status = UNSOLVED;
    } );

/************** METHOD: GetLink  **************/
/* "This tries to find the best avaliable AE." */
MakeMethod( Nodes, GetLink, [],
    {
    Self:BestType = LINK;
    ClearList( Self:Valid_Links );
    ClearList( Self:NewLinks );
    If Class?( L_ # Self:Item )
```

```
    Then GetInstanceList( L_ # Self:Item, Self:Valid_Links );
   If ( LengthList( Self:Valid_Links ) > 0 )
    Then {
        EnumList( Self:Valid_Links, VA,
           {
           If SendMessage( VA, Call, Self, Self:Item, Self:Variable,
                Self:Variation, Self:Critical_Time, Self:CostT )
            Then AppendToList( Self:NewLinks, VA );
           } );
       };
   Not( LengthList( Self:NewLinks ) == 0 );
   } );
SetMethodComment(Nodes, GetLink, "This tries to find the best avaliable AE." );

/************** METHOD: GetPath  **************/
MakeMethod( Nodes, GetPath, [],
   {
   Assert( Self:Item );
   Assert( Self:Variable );
   ForwardChain( NULL, Rule:LinkRules );
   SendMessage( Self, GetLink );
   } );

/************** METHOD: Search  **************/
/* "This method is called by Seek to search for a solution to the
node." */
MakeMethod( Nodes, Search, [],
   {
   ResetValue( Self:Best );
   ResetValue( Self:BestType );
   If Not( SendMessage( Self, GetAction ) )
    Then {
        If ( Global:LinkChain #= ALWAYS )
         Then {
             If Not( SendMessage( Self, GetPath ) )
              Then {
                  FALSE;
                  }
              Else {
                  TRUE;
                  };
             }
         Else {
             If Not( SendMessage( Self, GetLink ) )
              Then {
                  If Not( SendMessage( Self, GetPath ) )
                   Then {
                       FALSE;
                       }
                   Else {
                       TRUE;
                       };
```

```
                }
            Else {
                TRUE;
                };
            };
        }
    Else {
        Self:Status = FOUND;
        Self:TimeLSP = Self:TimeLBS + Self:TimeLSC;
        If Instance?( GetNthElem( Self:Caller, 1 ) )
         Then {
            If IsAKindOf?( GetNthElem( Self:Caller, 1 ),
                Nodes )
             Then SendMessage( Self:Calling_Link, Reply,
                 FOUND );
            };
        FOUND;
        };
  } );
SetMethodComment(Nodes, Search, "This method is called by Seek to search for a solution to the
node." );

/************** METHOD: Insert  **************/
MakeMethod( Nodes, Insert, [Copie ],
  {
  Self:Copies = Copie;
  } );

/************** METHOD: Store  **************/
MakeMethod( Nodes, Store, [APriority AVariation ACrt_Time ARepeatTime AWait ATimeLSC ACaller
AMessage Explain RuleNum ],
  {
  Self:Priority = APriority;
  Self:Variation = AVariation;
  Self:Wait = AWait;
  Self:Repeat_Time = ARepeatTime;
  ClearList( Self:Caller );
  If ( Instance?( ACaller ) And IsAKindOf?( ACaller, Links ) )
   Then {
      Self:Calling_Link = ACaller;
      If Not( Member?( Self:Caller, ACaller:Caller ) )
       Then AppendToList( Self:Caller, ACaller:Caller );
      }
   Else {
      If Not( Member?( Self:Caller, ACaller ) )
       Then AppendToList( Self:Caller, ACaller );
      Self:Calling_Link = NULL;
      Self:OriginalValue = Convert( Self:Item, Self:Variable,
                          GetValue( {
                                Self:Item;
                                },
                           {
```

```
                                    Self:Variable;
                                    } ) );
            Self:Message = Explain;
            SendMessage( Explanation, Add, FormatValue( "  Rule "
                                                    # ACaller
                                                    # " has initiated a change to  "
                                                    # Self:Variable
                                                    # " of "
                                                    # Self:Item
                                                    # " \n      by %3.2f. The original value is %3.2f units.",
                                            Self:Variation, Self:OriginalValue ) );
            };
        Self:OriginalValue = SendMessage( Self:Item, GetVal, Self:Variable,
                                VAR );
        Self:VariableType = SendMessage( Self:Item, GetVal, Self:Variable,
                                TYPE ):
        Self:Called_Time = GetTime( );
        Self:Critical_Time = ACrt_Time;
        Self:TimeLSC = ATimeLSC;
        Self:MaxCost = Min( 1 - ( Self:Priority / 20 ) + 0.5, 1 );
        Self:Mode = AMessage;
        Self:Status = SET;
        Self:RuleNo = RuleNum;
        If ( AMessage #= LOOK )
          Then {
              If Member?( Global:NewSetNodes, Self )
                Then RemoveFromList( Global:NewSetNodes, Self );
              }
          Else {
              If ( Not( Member?( Global:NewSetNodes, Self ) )
                      And ( AMessage #= ACTIVE ) )
                Then {
                    AppendToList( Global:NewSetNodes, Self );
                    };
              };
        TRUE;
        } );

/************ METHOD: Seek  ************/
MakeMethod( Nodes, Seek, [Cost ],
    {
    Self:Status = LOOKING;
    If Member?( Global:SetNodes, Self )
      Then RemoveFromList( Global:SetNodes, Self );
    If Member?( Global:NewSetNodes, Self )
      Then RemoveFromList( Global:NewSetNodes, Self );
    Self:CostT = Cost;
    Self:Temp = SendMessage( Self, Search );
    If ( Self:Temp #= FALSE )
      Then {
          If Not( SendMessage( Self, Root? ) )
            Then SendMessage( Self, Fnd, Self, CLEARDOWN );
```

```
        FALSE;
        }
     Else {
        If ( Self:Temp #= TRUE )
          Then Self:Status = PENDING;
        Self:Temp;
        };
   } );

/************** METHOD: NoLinks  **************/
MakeMethod( Nodes, NoLinks, [],
   {
   If ( LengthList( Self:NewLinks ) == 0 )
    Then {
        SendMessage( Self, GetPath );
        If ( LengthList( Self:NewLinks ) == 0 )
         Then {
            If Not( SendMessage( Self, Root? ) )
             Then {
                SendMessage( Self, End, Self, NORMAL );
                }
             Else {
                Self:Activated_Node = NULL;
                Self.Status = PENDING;
                };
            };
        };
   } );

/************** METHOD: Success  **************/
MakeMethod( Nodes, Success, [],
   {
   SendMessage( Explanation, Add, " - The " # Self:Variable #
                                  " of " # Self:Item # " was changed satisfactorily." );
   If Not( Self:Wait )
    Then {
        If Not( Null?( Self:Applied_Link ) )
         Then SendMessage( Self:Applied_Link, Reply, SUCCESS );
        If Not( Null?( Self:Applied_Action ) )
         Then SendMessage( Self:Applied_Action, Reply, SUCCESS );
        EnumList( Self:Caller, Call, If ( Instance?( Call )
                                          And IsAKindOf?( Call,
                                                Nodes ) )
                                       Then SendMessage( Call,
                                             Reply, Self,
                                             SUCCESS ) );
        };
   SendMessage( Self, End, Self, CLEARDOWN );
   } );

/************** METHOD: Root?  **************/
MakeMethod( Nodes, Root?, [],
```

```
    {
    EnumList( Self:Caller, Call,
       {
       Self:Root = TRUE;
       If Instance?( Call )
         Then If IsAKindOf?( Call, Nodes )
              Then Self:Root = FALSE;
       } );
    Self:Root,
    } );

/************** METHOD: Use **************/
MakeMethod( Nodes, Use, [ACaller ],
    {
    If Not( Member?( Self:Caller, ACaller ) )
      Then AppendToList( Self:Caller, ACaller );
    SendMessage( Explanation, Display, "   " # ACaller # " using "
                                  # Self # " as well!" );
    TRUE;
    } );
MakeSlot( Nodes:Priority );
SetSlotOption( Nodes:Priority, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:Priority, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:Priority, MINIMUM_VALUE, 0 );
SetSlotOption( Nodes:Priority, MAXIMUM_VALUE, 10 );
Nodes:Priority = 10;
SetSlotOption( Nodes:Priority, IF_NEEDED, NULL );
SetSlotOption( Nodes:Priority, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Priority, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Priority, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Variation );
Nodes:Variation = 0;
SetSlotOption( Nodes:Variation, IF_NEEDED, NULL );
SetSlotOption( Nodes:Variation, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Variation, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Variation, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Valid_Actions );
SetSlotOption( Nodes:Valid_Actions, MULTIPLE );
SetSlotOption( Nodes:Valid_Actions, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Valid_Actions, ALLOWABLE_CLASSES, Actions );
ClearList( Nodes:Valid_Actions );
SetSlotOption( Nodes:Valid_Actions, IF_NEEDED, NULL );
SetSlotOption( Nodes:Valid_Actions, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Valid_Actions, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Valid_Actions, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Status );
SetSlotOption( Nodes:Status, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:Status, ALLOWABLE_VALUES, READY, SET, LOOKING, FOUND,
PENDING, HOLDING, ACTIVATED, UNSOLVED, USE );
Nodes:Status = READY;
SetSlotOption( Nodes:Status, IF_NEEDED, NULL );
SetSlotOption( Nodes:Status, WHEN_ACCESS, NULL );
```

```
SetSlotOption( Nodes:Status, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Status, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Applied_Action );
SetSlotOption( Nodes:Applied_Action, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Applied_Action, ALLOWABLE_CLASSES, Actions );
SetSlotOption( Nodes:Applied_Action, IF_NEEDED, NULL );
SetSlotOption( Nodes:Applied_Action, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Applied_Action, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Applied_Action, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Caller );
SetSlotOption( Nodes:Caller, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:Caller, MULTIPLE );
SetSlotOption( Nodes:Caller, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Caller, ALLOWABLE_CLASSES, Links, Nodes );
ClearList( Nodes:Caller );
SetSlotOption( Nodes:Caller, IF_NEEDED, NULL );
SetSlotOption( Nodes:Caller, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Caller, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Caller, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Item );
SetSlotOption( Nodes:Item, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Item, ALLOWABLE_CLASSES, Items );
SetSlotOption( Nodes:Item, IF_NEEDED, NULL );
SetSlotOption( Nodes:Item, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Item, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Item, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Variable );
SetSlotOption( Nodes:Variable, IF_NEEDED, NULL );
SetSlotOption( Nodes:Variable, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Variable, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Variable, AFTER_CHANGE, NULL );
MakeSlot( Nodes:OriginalValue );
SetSlotOption( Nodes:OriginalValue, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes.OriginalValue, VALUE_TYPE, NUMBER );
Nodes:OriginalValue = 0;
SetSlotOption( Nodes:OriginalValue, IF_NEEDED, NULL );
SetSlotOption( Nodes:OriginalValue, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:OriginalValue, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:OriginalValue, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Critical_Time );
SetSlotOption( Nodes:Critical_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:Critical_Time, MINIMUM_VALUE, 0 );
Nodes:Critical_Time = 0;
SetSlotOption( Nodes:Critical_Time, IF_NEEDED, NULL );
SetSlotOption( Nodes:Critical_Time, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Critical_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Critical_Time, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Called_Time );
SetSlotOption( Nodes:Called_Time, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:Called_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:Called_Time, MINIMUM_VALUE, 0 );
Nodes:Called_Time = 0;
```

```
SetSlotOption( Nodes:Called_Time, IF_NEEDED, NULL );
SetSlotOption( Nodes:Called_Time, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Called_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Called_Time, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Calling_Time );
SetSlotOption( Nodes:Calling_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:Calling_Time, MINIMUM_VALUE, 0 );
Nodes:Calling_Time = 0;
SetSlotOption( Nodes:Calling_Time, IF_NEEDED, NULL );
SetSlotOption( Nodes:Calling_Time, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Calling_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Calling_Time, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Valid_Links );
SetSlotOption( Nodes:Valid_Links, MULTIPLE );
SetSlotOption( Nodes:Valid_Links, VALUE_TYPE, NUMBER );
ClearList( Nodes:Valid_Links );
MakeSlot( Nodes:Applied_Link );
SetSlotOption( Nodes:Applied_Link, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:Applied_Link, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Applied_Link, ALLOWABLE_CLASSES, Action_Effect );
SetSlotOption( Nodes:Applied_Link, IF_NEEDED, NULL );
SetSlotOption( Nodes:Applied_Link, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Applied_Link, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Applied_Link, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Best );
SetSlotOption( Nodes:Best, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Best, ALLOWABLE_CLASSES, Actions );
MakeSlot( Nodes:BestType );
SetSlotOption( Nodes:BestType, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:BestType, ALLOWABLE_VALUES, ACTION, PATH, LINK );
MakeSlot( Nodes:BestCost );
SetSlotOption( Nodes:BestCost, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:BestCost, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:BestCost, MAXIMUM_VALUE, 1 );
Nodes:BestCost = 0;
SetSlotOption( Nodes:BestCost, IF_NEEDED, NULL );
SetSlotOption( Nodes:BestCost, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:BestCost, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:BestCost, AFTER_CHANGE, NULL );
MakeSlot( Nodes:CostS );
SetSlotComment( Nodes:CostS, "No longer used. It contained the cost of moving the variable
from it's set point. It was found to be confusing." );
SetSlotOption( Nodes:CostS, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:CostS, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:CostS, MAXIMUM_VALUE, 1 );
Nodes:CostS = 0;
MakeSlot( Nodes:CostST );
SetSlotComment( Nodes:CostST, "Total cost for self to be changed (value passed back)" );
SetSlotOption( Nodes:CostST, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:CostST, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:CostST, MAXIMUM_VALUE, 1 );
Nodes:CostST = 0;
```

```
MakeSlot( Nodes:FinalCost );
SetSlotOption( Nodes:FinalCost, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:FinalCost, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:FinalCost, MAXIMUM_VALUE, 1 );
Nodes:FinalCost = 0;
SetSlotOption( Nodes:FinalCost, IF_NEEDED, NULL );
SetSlotOption( Nodes:FinalCost, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:FinalCost, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:FinalCost, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Mode );
SetSlotOption( Nodes:Mode, ALLOWABLE_VALUES, ACTIVE, LOOK );
Nodes:Mode = LOOK;
SetSlotOption( Nodes:Mode, IF_NEEDED, NULL );
SetSlotOption( Nodes:Mode, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Mode, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Mode, AFTER_CHANGE, NULL );
MakeSlot( Nodes:TempCost );
SetSlotOption( Nodes:TempCost, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:TempCost, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:TempCost, MAXIMUM_VALUE, 1 );
Nodes:TempCost = 0;
SetSlotOption( Nodes:TempCost, IF_NEEDED, NULL );
SetSlotOption( Nodes:TempCost, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:TempCost, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:TempCost, AFTER_CHANGE, NULL );
MakeSlot( Nodes:TimeLSC );
SetSlotComment( Nodes:TimeLSC, "Time lag between self and caller." );
SetSlotOption( Nodes:TimeLSC, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:TimeLSC, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:TimeLSC, MAXIMUM_VALUE, 1 );
Nodes:TimeLSC = 0;
MakeSlot( Nodes:TimeLSP );
SetSlotComment( Nodes:TimeLSP, "The time for self to change parent. " );
Nodes:TimeLSP = 0;
MakeSlot( Nodes:Copies );
SetSlotOption( Nodes:Copies, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Copies, ALLOWABLE_CLASSES, Copies );
MakeSlot( Nodes:MaxCost );
SetSlotOption( Nodes:MaxCost, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:MaxCost, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:MaxCost, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:MaxCost, MAXIMUM_VALUE, 1 );
Nodes:MaxCost = 1;
SetSlotOption( Nodes:MaxCost, IF_NEEDED, NULL );
SetSlotOption( Nodes:MaxCost, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:MaxCost, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:MaxCost, AFTER_CHANGE, NULL );
MakeSlot( Nodes:NewLinks );
SetSlotOption( Nodes:NewLinks, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:NewLinks, MULTIPLE );
SetSlotOption( Nodes:NewLinks, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:NewLinks, ALLOWABLE_CLASSES, Links );
```

```
ClearList( Nodes:NewLinks );
SetSlotOption( Nodes:NewLinks, IF_NEEDED, NULL );
SetSlotOption( Nodes:NewLinks, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:NewLinks, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:NewLinks, AFTER_CHANGE, NULL );
MakeSlot( Nodes:TimeLBS );
SetSlotOption( Nodes:TimeLBS, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:TimeLBS, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:TimeLBS, MINIMUM_VALUE, 0 );
Nodes:TimeLBS = 0;
SetSlotOption( Nodes:TimeLBS, IF_NEEDED, NULL );
SetSlotOption( Nodes:TimeLBS, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:TimeLBS, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:TimeLBS, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Activated_Node );
SetSlotOption( Nodes:Activated_Node, IF_NEEDED, NULL );
SetSlotOption( Nodes:Activated_Node, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Activated_Node, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Activated_Node, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Temp );
MakeSlot( Nodes:Calling_Link );
SetSlotOption( Nodes:Calling_Link, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:Calling_Link, VALUE_TYPE, OBJECT );
SetSlotOption( Nodes:Calling_Link, ALLOWABLE_CLASSES, Links );
SetSlotOption( Nodes:Calling_Link, IF_NEEDED, NULL );
SetSlotOption( Nodes:Calling_Link, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Calling_Link, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Calling_Link, AFTER_CHANGE, NULL );
MakeSlot( Nodes:CostP );
SetSlotComment( Nodes:CostP, "This is the cost of using the primary node excluding
the cost of actually implementing that node." );
SetSlotOption( Nodes:CostP, NO_AUTO_ASK, TRUE );
SetSlotOption( Nodes:CostP, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:CostP, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:CostP, MAXIMUM_VALUE, 1 );
Nodes:CostP = 0;
MakeSlot( Nodes:VariableType );
SetSlotOption( Nodes:VariableType, ALLOWABLE_VALUES, Analogue, Discrete );
Nodes:VariableType = Analogue;
SetSlotOption( Nodes:VariableType, IF_NEEDED, NULL );
SetSlotOption( Nodes:VariableType, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:VariableType, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:VariableType, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Boolean );
SetSlotOption( Nodes:Boolean, VALUE_TYPE, BOOLEAN );
SetSlotOption( Nodes:Boolean, IF_NEEDED, NULL );
SetSlotOption( Nodes:Boolean, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Boolean, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Boolean, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Root );
SetSlotOption( Nodes:Root, VALUE_TYPE, BOOLEAN );
Nodes:Root = TRUE;
```

```
SetSlotOption( Nodes:Root, IF_NEEDED, NULL );
SetSlotOption( Nodes:Root, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:Root, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:Root, AFTER_CHANGE, NULL );
MakeSlot( Nodes:Message );
MakeSlot( Nodes:RuleNo );
SetSlotOption( Nodes:RuleNo, VALUE_TYPE, NUMBER );
Nodes:RuleNo = 0;
SetSlotOption( Nodes:RuleNo, IF_NEEDED, NULL );
SetSlotOption( Nodes:RuleNo, WHEN_ACCESS, NULL );
SetSlotOption( Nodes:RuleNo, BEFORE_CHANGE, NULL );
SetSlotOption( Nodes:RuleNo, AFTER_CHANGE, NULL );
MakeSlot( Nodes:CostPT );
SetSlotComment( Nodes:CostPT, "The cost of the secondary node that will cause this node
to change." );
SetSlotOption( Nodes:CostPT, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:CostPT, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:CostPT, MAXIMUM_VALUE, 1 );
Nodes:CostPT = 0;
MakeSlot( Nodes:CostT );
SetSlotComment( Nodes:CostT, "Cost of path thus far." );
SetSlotOption( Nodes:CostT, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:CostT, MINIMUM_VALUE, -1 );
SetSlotOption( Nodes:CostT, MAXIMUM_VALUE, 1 );
Nodes:CostT = 0;
MakeSlot( Nodes:Wait );
SetSlotOption( Nodes:Wait, VALUE_TYPE, BOOLEAN );
Nodes:Wait = FALSE;
MakeSlot( Nodes:Repeat_Time );
SetSlotOption( Nodes:Repeat_Time, VALUE_TYPE, NUMBER );
SetSlotOption( Nodes:Repeat_Time, MINIMUM_VALUE, 0 );
Nodes:Repeat_Time = 0;

      /*************************************
       **** CLASS: Items
       **** This class contains all the knowledge about the physical
       **** layout of the plant.
       *************************************/
MakeClass( Items, Root );
SetClassComment( Items, "This class contains all the knowledge about the physical
layout of the plant." );

/************** METHOD: Get_Value  **************/
/* "This function calls the value_list, finds all the values required." */
MakeMethod( Items, Get_Value, [],
   {
   GetSlotList( Self, Self:Value_List );
   EnumList( Self:Value_List, Value,
      {
      If Member?( Self:ValidVars, Value )
        Then {
            Assert( Self:Value );
```

```
            If Slot?( Self,
                {
                N # Value;
                } )
              Then {
                If Not( GetValue( GetValue( Self, N # Value ),
                        Tag ) #= NONE )
                  Then {
                    SendMessage( Self, SetVar, Value,
                      Read_Data( GetValue( GetValue( Self,
                                           N #
                                             Value ),
                                    Tag ) ) );
                    };
                }
              Else {
                Error( 22, Self, Value );
                };
            };
      } );
  } );
SetMethodComment(Items, Get_Value, "This function calls the value_list, finds all the values required."
);

/*************** METHOD: Request  ***************/
MakeMethod( Items, Request, [Variable Priority Variation Crt_Time RepeatTime AWait TimeLSC
Caller Message Explanation RuleNo ],
  {
  Self:Counter = 0;
  Self:NodeName = N # Self # _ # Variable;
  If Not( Instance?( Self:NodeName ) )
    Then MakeInstance( Self:NodeName, Nodes );
  While ({
        Global:Temp = SendMessage( Self:NodeName, Set, Self,
                      Variable, Priority, Variation, Crt_Time,
                      RepeatTime, AWait, TimeLSC, Caller,
                      Message, Explanation, RuleNo );
        Global:Temp #= COPY;
        })
       {
       If Not( Instance?( C_ # Self # _ # Variable ) )
         Then MakeInstance( C_ # Self # _ # Variable, Copies );
       SendMessage( C_ # Self # _ # Variable, Insert, Self:NodeName );
       SendMessage( Self:NodeName, Insert, C_ # Self # _ #
                                           Variable );
       Self:Counter += 1;
       Self:NodeName = Self # _ # Variable # _ # Self:Counter;
       If Not( Instance?( Self:NodeName ) )
         Then MakeInstance( Self:NodeName, Nodes );
       };
  If Instance?( C_ # Self # _ # Variable )
    Then {
```

```
        SendMessage( C_ # Self # _ # Variable, Insert, Self:NodeName );
        SendMessage( Self:NodeName, Insert, C_ # Self # _ #
                                        Variable );
        };
    If ( Global:Temp #= TRUE )
     Then Self:NodeName
     Else Global:Temp;
    } );

/*************** METHOD: GetVar  **************/
MakeMethod( Items, GetVar, [Name ],
    {
    GetValue( GetValue( Self, N # Name ), Variable );
    } );

/*************** METHOD: SetVar  **************/
MakeMethod( Items, SetVar, [Name Parameter ],
    {
    SendMessage( GetValue( Self, N # Name ), SetVal, Parameter );
    Self:Name = GetValue( GetValue( Self, N # Name ), Variable );
    } );

/*************** METHOD: GetVal  **************/
MakeMethod( Items, GetVal, [Name Parameter ],
    {
    If ( Parameter #= SP )
     Then {
         GetValue( GetValue( Self, N # Name ), SetPoint );
         }
     Else {
         If ( Parameter #= UL )
          Then {
              GetValue( GetValue( Self, N # Name ), UpperLevel );
              }
          Else {
              If ( Parameter #= LL )
               Then {
                   GetValue( GetValue( Self, N # Name ),
                      LowerLevel );
                   }
               Else {
                   If ( Parameter #= VAR )
                    Then {
                        Convert( Self, Name, GetValue( GetValue( Self,
                                                    N
                                                      #
                                                      Name ),
                                            Variable ) );
                    }
                   Else {
                       If ( Parameter #= TAG )
                        Then {
```

```
    GetValue( GetValue( Self,
              N # Name ),
      Tag );
    }
  Else {
    If ( Parameter #= MAX )
     Then {
        GetSlotOption( Self,
          Name, MAXIMUM_VALUE );
        }
     Else {
        If ( Parameter #=
               MIN )
         Then {
            GetSlotOption( Self,
              Name,
              MAXIMUM_VALUE );
            }
         Else {
            If ( Parameter
                   #=
                   TYPE )
             Then {
                GetValue( GetValue( Self,
                            N
                              #
                              Name ),
                  Type );
                }
             Else {
                If ( Parameter
                       #=
                       DIM )
                 Then {
                    GetValue( GetValue( Self,
                                N
                                  #
                                  Name ),
                      Units );
                    }
                 Else {
                    If ( Parameter
                           #=
                           NO )
                     Then {
                        GetValue( GetValue( Self,
                                    N
                                      #
                                      Name ),
                          Number );
                        }
                     Else {};
```

```
                                                    };
                                                };
                                            };
                                        };
                                    };
                                };
                            };
                        };
                    } );

/************** METHOD: SetVal  **************/
MakeMethod( Items, SetVal, [Name Value Parameter ],
    {
    If ( Parameter #= SP )
     Then {
         SetValue( GetValue( Self, N # Name ), SetPoint, Value );
         }
     Else {
         If ( Parameter #= UL )
          Then {
              SetValue( GetValue( Self, N # Name ), UpperLevel,
                 Value );
              }
          Else {
              If ( Parameter #= LL )
               Then {
                   SetValue( GetValue( Self, N # Name ),
                      LowerLevel, Value );
                   }
               Else {
                   If ( Parameter #= VAR )
                    Then {
                        SetValue( GetValue( Self, N #
                                                Name ),
                           Variable, Value );
                        }
                    Else {
                        If ( Parameter #= TAG )
                         Then {
                             SetValue( GetValue( Self,
                                      N # Name ),
                                Tag, Value );
                             }
                         Else {
                             If ( Parameter #= MAX )
                              Then {
                                  SetSlotOption( Self,
                                     Name, MAXIMUM_VALUE,
                                     Value );
                                  }
                              Else {
```

```
                                    If ( Parameter #=
                                          MIN )
                                     Then {
                                        SetSlotOption( Self,
                                           Name,
                                           MAXIMUM_VALUE,
                                           Value );
                                        }
                                     Else {
                                        If ( Parameter
                                              #=
                                              TYPE )
                                         Then {
                                            SetValue( GetValue( Self,
                                                        N
                                                           #
                                                           Name ),
                                              Type,
                                              Value );
                                            }
                                         Else {};
                                        };
                                    };
                                };
                            };
                        };
                    };
                };
            };
        } );
MakeSlot( Items:Inputs );
SetSlotOption( Items:Inputs, MULTIPLE );
SetSlotOption( Items:Inputs, VALUE_TYPE, OBJECT );
SetSlotOption( Items:Inputs, ALLOWABLE_CLASSES, Items );
ClearList( Items:Inputs );
SetSlotOption( Items:Inputs, IF_NEEDED, NULL );
SetSlotOption( Items:Inputs, WHEN_ACCESS, NULL );
SetSlotOption( Items:Inputs, BEFORE_CHANGE, NULL );
SetSlotOption( Items:Inputs, AFTER_CHANGE, NULL );
MakeSlot( Items:Outputs );
SetSlotOption( Items:Outputs, MULTIPLE );
SetSlotOption( Items:Outputs, VALUE_TYPE, OBJECT );
SetSlotOption( Items:Outputs, ALLOWABLE_CLASSES, Items );
ClearList( Items:Outputs );
SetSlotOption( Items:Outputs, IF_NEEDED, NULL );
SetSlotOption( Items:Outputs, WHEN_ACCESS, NULL );
SetSlotOption( Items:Outputs, BEFORE_CHANGE, NULL );
SetSlotOption( Items:Outputs, AFTER_CHANGE, NULL );
MakeSlot( Items:ValidVars );
SetSlotOption( Items:ValidVars, MULTIPLE );
SetValue( Items:ValidVars, Level, Temp, Acid, Flow, Settling, SLevel, Zn, Status, Flow, Basicity,
Speed, Screw1, Screw2, Screw12, Running, PumpAB, PumpAB, PumpAOrB, PumpA, PumpB,
PumpAB, PumpAOrB, PumpAB, PumpAOrB, PumpAB, PumpAOrB, PumpAB, PumpAOrB, PumpAB,
```

```
PumpAOrB, ScrewCon, ScrewConv, DLevel, Frequency, FTime, DLevSetpoint, DLevMax, DLevMin,
cle );
SetSlotOption( Items:ValidVars, IF_NEEDED, NULL );
SetSlotOption( Items:ValidVars, WHEN_ACCESS, NULL );
SetSlotOption( Items:ValidVars, BEFORE_CHANGE, NULL );
SetSlotOption( Items:ValidVars, AFTER_CHANGE, NULL );
MakeSlot( Items:Value_List );
SetSlotOption( Items:Value_List, MULTIPLE );
SetValue( Items:Value_List, Inputs, Outputs, ValidVars, Value_List, NodeName, Counter, ValidSlots,
ItemNo );
MakeSlot( Items:NodeName );
SetSlotOption( Items:NodeName, NO_AUTO_ASK, TRUE );
SetSlotOption( Items:NodeName, IF_NEEDED, NULL );
SetSlotOption( Items:NodeName, WHEN_ACCESS, NULL );
SetSlotOption( Items:NodeName, BEFORE_CHANGE, NULL );
SetSlotOption( Items:NodeName, AFTER_CHANGE, NULL );
MakeSlot( Items:Counter );
SetSlotComment( Items:Counter, "A counter used by request." );
SetSlotOption( Items:Counter, VALUE_TYPE, NUMBER );
SetSlotOption( Items:Counter, MINIMUM_VALUE, 0 );
MakeSlot( Items:ValidSlots );
SetSlotOption( Items:ValidSlots, NO_AUTO_ASK, TRUE );
SetSlotOption( Items:ValidSlots, MULTIPLE );
SetValue( Items:ValidSlots, Hi );
SetSlotOption( Items:ValidSlots, IF_NEEDED, NULL );
SetSlotOption( Items:ValidSlots, WHEN_ACCESS, NULL );
SetSlotOption( Items:ValidSlots, BEFORE_CHANGE, NULL );
SetSlotOption( Items:ValidSlots, AFTER_CHANGE, NULL );
MakeSlot( Items:ItemNo );

    /*****************************************
     **** CLASS: Variables
     **** This class contains all the variables in the plant.
     *****************************************/
MakeClass( Variables, Root );
SetClassComment( Variables, "This class contains all the variables in the plant." );

/************** METHOD: Set  **************/
MakeMethod( Variables, Set, [slot ],
  {
  SetSlotOption( Self, slot, MAXIMUM_VALUE, GetSlotOption( Owner,
                                    GetParent( Self ),
                                    MAXIMUM_VALUE ) );
  SetSlotOption( Self, slot, MINIMUM_VALUE, GetSlotOption( Owner,
                                    GetParent( Self ),
                                    MINIMUM_VALUE ) );
  } );

/************** METHOD: Convert  **************/
MakeMethod( Variables, Convert, [Value ],
  {
  If Member?( Self:StateList, Value )
```

```
    Then {
        GetNthElem( Self:StateValues, GetElemPos( Self:StateList,
                                        Value ) );
        }
    Else NULL;
  } );

/************ METHOD: SetVal  ************/
MakeMethod( Variables, SetVal, [Value ],
  {
  If ( Null?( Value ) Or ( Not( Self:Type #= Discrete )
                          Or Not( Number?( Value ) ) ) )
    Then ( Self:Variable = Value )
    Else {
        If ( GetElemPos( Self:StateValues, Value )
              == 0 )
          Then ( Self:Variable = NULL )
          Else Self:Variable = GetNthElem( Self:StateList, GetElemPos( Self:StateValues,
                                                Value ) );
        };
  } );
MakeSlot( Variables:Variable );
SetSlotOption( Variables:Variable, NO_AUTO_ASK, TRUE );
Variables:Variable = 0;
MakeSlot( Variables:SetPoint );
SetSlotOption( Variables:SetPoint, NO_AUTO_ASK, TRUE );
SetSlotOption( Variables:SetPoint, VALUE_TYPE, NUMBER );
Variables:SetPoint = 50;
SetSlotOption( Variables:SetPoint, IF_NEEDED, NULL );
SetSlotOption( Variables:SetPoint, WHEN_ACCESS, NULL );
SetSlotOption( Variables:SetPoint, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:SetPoint, AFTER_CHANGE, NULL );
MakeSlot( Variables:UpperLevel );
SetSlotOption( Variables:UpperLevel, NO_AUTO_ASK, TRUE );
SetSlotOption( Variables:UpperLevel, VALUE_TYPE, NUMBER );
Variables:UpperLevel = 100;
SetSlotOption( Variables:UpperLevel, IF_NEEDED, NULL );
SetSlotOption( Variables:UpperLevel, WHEN_ACCESS, NULL );
SetSlotOption( Variables:UpperLevel, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:UpperLevel, AFTER_CHANGE, NULL );
MakeSlot( Variables:LowerLevel );
SetSlotOption( Variables:LowerLevel, NO_AUTO_ASK, TRUE );
SetSlotOption( Variables:LowerLevel, VALUE_TYPE, NUMBER );
Variables:LowerLevel = 0;
SetSlotOption( Variables:LowerLevel, IF_NEEDED, NULL );
SetSlotOption( Variables:LowerLevel, WHEN_ACCESS, NULL );
SetSlotOption( Variables:LowerLevel, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:LowerLevel, AFTER_CHANGE, NULL );
MakeSlot( Variables:Owner );
SetSlotOption( Variables:Owner, NO_AUTO_ASK, TRUE );
SetSlotOption( Variables:Owner, IF_NEEDED, NULL );
SetSlotOption( Variables:Owner, WHEN_ACCESS, NULL );
```

```
SetSlotOption( Variables:Owner, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:Owner, AFTER_CHANGE, NULL );
MakeSlot( Variables:Condition );
MakeSlot( Variables:Tag );
SetSlotOption( Variables:Tag, NO_AUTO_ASK, TRUE );
Variables:Tag = NONE;
SetSlotOption( Variables:Tag, IF_NEEDED, NULL );
SetSlotOption( Variables:Tag, WHEN_ACCESS, NULL );
SetSlotOption( Variables:Tag, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:Tag, AFTER_CHANGE, NULL );
MakeSlot( Variables:Units );
Variables:Units = FormatValue ( "\%" );
SetSlotOption( Variables:Units, IF_NEEDED, NULL );
SetSlotOption( Variables:Units, WHEN_ACCESS, NULL );
SetSlotOption( Variables:Units, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:Units, AFTER_CHANGE, NULL );
MakeSlot( Variables:Type );
SetSlotOption( Variables:Type, ALLOWABLE_VALUES, Discrete, Analog );
Variables:Type = Analog;
SetSlotOption( Variables:Type, IF_NEEDED, NULL );
SetSlotOption( Variables:Type, WHEN_ACCESS, NULL );
SetSlotOption( Variables:Type, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:Type, AFTER_CHANGE, NULL );
MakeSlot( Variables:StateList );
SetSlotOption( Variables:StateList, MULTIPLE );
ClearList( Variables:StateList );
SetSlotOption( Variables:StateList, IF_NEEDED, NULL );
SetSlotOption( Variables:StateList, WHEN_ACCESS, NULL );
SetSlotOption( Variables:StateList, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:StateList, AFTER_CHANGE, NULL );
MakeSlot( Variables:StateValues );
SetSlotOption( Variables:StateValues, MULTIPLE );
SetSlotOption( Variables:StateValues, VALUE_TYPE, NUMBER );
ClearList( Variables:StateValues );
SetSlotOption( Variables:StateValues, IF_NEEDED, NULL );
SetSlotOption( Variables:StateValues, WHEN_ACCESS, NULL );
SetSlotOption( Variables:StateValues, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:StateValues, AFTER_CHANGE, NULL );
MakeSlot( Variables:LongText );
MakeSlot( Variables:ShortText );
MakeSlot( Variables:PlantArea );
Variables:PlantArea = P;
SetSlotOption( Variables:PlantArea, IF_NEEDED, NULL );
SetSlotOption( Variables:PlantArea, WHEN_ACCESS, NULL );
SetSlotOption( Variables:PlantArea, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:PlantArea, AFTER_CHANGE, NULL );
MakeSlot( Variables:MesType );
SetSlotOption( Variables:MesType, ALLOWABLE_VALUES, ANALOG, COUNTER, BINARY );
Variables:MesType = ANALOG;
SetSlotOption( Variables:MesType, IF_NEEDED, NULL );
SetSlotOption( Variables:MesType, WHEN_ACCESS, NULL );
SetSlotOption( Variables:MesType, BEFORE_CHANGE, NULL );
```

```
SetSlotOption( Variables:MesType, AFTER_CHANGE, NULL );
MakeSlot( Variables:Number );
SetSlotOption( Variables:Number, VALUE_TYPE, NUMBER );
Variables:Number = 0;
SetSlotOption( Variables:Number, IF_NEEDED, NULL );
SetSlotOption( Variables:Number, WHEN_ACCESS, NULL );
SetSlotOption( Variables:Number, BEFORE_CHANGE, NULL );
SetSlotOption( Variables:Number, AFTER_CHANGE, NULL );



/*****************************************************************/
/**        ALL INSTANCES ARE SAVED BELOW                        **/
/*****************************************************************/

/************ METHOD: Update  ************/
MakeMethod( Global, Update, [],
   {
   Update( );
   } );

/************ METHOD: Percent  ************/
MakeMethod( Global, Percent, [],
   {
   If ( Global:Successes + Global:Failures == 0 )
     Then ( Global:Percent = 100 )
     Else {
         Global:Percent = Global:Failures / ( Global:Failures
                                    + Global:Successes )
           * 100;
         };
   } );

/************ METHOD: AVtime  ************/
MakeMethod( Global, AVtime, [New Old ],
   {
   Average( Self:New, Old );
   } );

/************ METHOD: CalcAV  ************/
MakeMethod( Global, CalcAV, [],
   Self:AVTime = Self:AVTimeF - Self:AVTimeS );

/************ METHOD: BackUpSet  ************/
MakeMethod( Global, BackUpSet, [],
   {
   KillTimer( 3 );
   SetTimer( 3, Global:BackUpRate, 0 );
   } );
MakeSlot( Global:SetNodes );
SetSlotOption( Global:SetNodes, INHERIT, FALSE );
SetSlotOption( Global:SetNodes, NO_AUTO_ASK, TRUE );
```

```
SetSlotOption( Global:SetNodes, MULTIPLE );
SetSlotOption( Global:SetNodes, VALUE_TYPE, OBJECT );
SetSlotOption( Global:SetNodes, ALLOWABLE_CLASSES, Nodes );
ClearList( Global:SetNodes );
SetSlotOption( Global:SetNodes, IF_NEEDED, NULL );
SetSlotOption( Global:SetNodes, WHEN_ACCESS, NULL );
SetSlotOption( Global:SetNodes, BEFORE_CHANGE, NULL );
SetSlotOption( Global:SetNodes, AFTER_CHANGE, NULL );
MakeSlot( Global:Init_List );
SetSlotOption( Global:Init_List, INHERIT, FALSE );
SetSlotOption( Global:Init_List, MULTIPLE );
SetSlotOption( Global:Init_List, VALUE_TYPE, OBJECT );
SetSlotOption( Global:Init_List, ALLOWABLE_CLASSES, Action_Effects );
SetValue( Global:Init_List, TKP01_Level );
SetSlotOption( Global:Init_List, IF_NEEDED, NULL );
SetSlotOption( Global:Init_List, WHEN_ACCESS, NULL );
SetSlotOption( Global:Init_List, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Init_List, AFTER_CHANGE, NULL );
MakeSlot( Global:Item_List );
SetSlotOption( Global:Item_List, INHERIT, FALSE );
SetSlotOption( Global:Item_List, MULTIPLE );
SetSlotOption( Global:Item_List, VALUE_TYPE, OBJECT );
SetSlotOption( Global:Item_List, ALLOWABLE_CLASSES, Items );
SetValue( Global:Item_List, P01, P02 );
SetSlotOption( Global:Item_List, IF_NEEDED, NULL );
SetSlotOption( Global:Item_List, WHEN_ACCESS, NULL );
SetSlotOption( Global:Item_List, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Item_List, AFTER_CHANGE, NULL );
MakeSlot( Global:Temp );
SetSlotOption( Global:Temp, INHERIT, FALSE );
SetSlotOption( Global:Temp, NO_AUTO_ASK, TRUE );
Global:Temp = FALSE;
SetSlotOption( Global:Temp, IF_NEEDED, NULL );
SetSlotOption( Global:Temp, WHEN_ACCESS, NULL );
SetSlotOption( Global:Temp, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Temp, AFTER_CHANGE, NULL );
MakeSlot( Global:Continue );
SetSlotOption( Global:Continue, INHERIT, FALSE );
SetSlotOption( Global:Continue, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:Continue, ALLOWABLE_VALUES, Yes, No );
Global:Continue = Yes;
MakeSlot( Global:Time );
SetSlotOption( Global:Time, VALUE_TYPE, NUMBER );
Global:Time = 505.910000;
SetSlotOption( Global:Time, IMAGE, Edit3 );
MakeSlot( Global:Value );
SetSlotComment( Global:Value, "This slot is used by Read_Data and is a temporary affair." );
SetSlotOption( Global:Value, VALUE_TYPE, NUMBER );
SetSlotOption( Global:Value, MINIMUM_VALUE, -100 );
SetSlotOption( Global:Value, MAXIMUM_VALUE, 100 );
Global:Value = 13.7306694674992;
MakeSlot( Global:State );
```

```
SetSlotOption( Global:State, INHERIT, FALSE );
SetSlotOption( Global:State, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:State, ALLOWABLE_VALUES, READY, INITIAL, READING, WAITING,
CHAIN, RESET, SORTING, SEARCHING, ACTIVATING, CLEARING );
Global:State = READING;
SetSlotOption( Global:State, IF_NEEDED, NULL );
SetSlotOption( Global:State, WHEN_ACCESS, NULL );
SetSlotOption( Global:State, BEFORE_CHANGE, NULL );
SetSlotOption( Global:State, AFTER_CHANGE, NULL );
SetSlotOption( Global:State, IMAGE, StateBox14, ComboBox1 );
MakeSlot( Global:Temp_List );
SetSlotOption( Global:Temp_List, INHERIT, FALSE );
SetSlotOption( Global:Temp_List, MULTIPLE );
SetSlotOption( Global:Temp_List, VALUE_TYPE, OBJECT );
SetSlotOption( Global:Temp_List, ALLOWABLE_CLASSES, Actions );
ClearList( Global:Temp_List );
SetSlotOption( Global:Temp_List, IF_NEEDED, NULL );
SetSlotOption( Global:Temp_List, WHEN_ACCESS, NULL );
SetSlotOption( Global:Temp_List, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Temp_List, AFTER_CHANGE, NULL );
MakeSlot( Global:NewSetNodes );
SetSlotOption( Global:NewSetNodes, INHERIT, FALSE );
SetSlotOption( Global:NewSetNodes, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:NewSetNodes, MULTIPLE );
SetSlotOption( Global:NewSetNodes, VALUE_TYPE, OBJECT );
SetSlotOption( Global:NewSetNodes, ALLOWABLE_CLASSES, Nodes );
ClearList( Global:NewSetNodes );
SetSlotOption( Global:NewSetNodes, IF_NEEDED, NULL );
SetSlotOption( Global:NewSetNodes, WHEN_ACCESS, NULL );
SetSlotOption( Global:NewSetNodes, BEFORE_CHANGE, NULL );
SetSlotOption( Global:NewSetNodes, AFTER_CHANGE, NULL );
MakeSlot( Global:Activated_Nodes );
SetSlotOption( Global:Activated_Nodes, INHERIT, FALSE );
SetSlotOption( Global:Activated_Nodes, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:Activated_Nodes, MULTIPLE );
SetSlotOption( Global:Activated_Nodes, VALUE_TYPE, OBJECT );
SetSlotOption( Global:Activated_Nodes, ALLOWABLE_CLASSES, Nodes );
ClearList( Global:Activated_Nodes );
SetSlotOption( Global:Activated_Nodes, IF_NEEDED, NULL );
SetSlotOption( Global:Activated_Nodes, WHEN_ACCESS, NULL );
SetSlotOption( Global:Activated_Nodes, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Activated_Nodes, AFTER_CHANGE, NULL );
MakeSlot( Global:Cost );
SetSlotComment( Global:Cost, "Cost used by cost functions
" );
SetSlotOption( Global:Cost, VALUE_TYPE, NUMBER );
SetSlotOption( Global:Cost, MINIMUM_VALUE, -1 );
SetSlotOption( Global:Cost, MAXIMUM_VALUE, 1 );
Global:Cost = 0.4316546762.8993;
MakeSlot( Global:UnexpandedList );
SetSlotOption( Global:UnexpandedList, INHERIT, FALSE );
SetSlotOption( Global:UnexpandedList, NO_AUTO_ASK, TRUE );
```

```
SetSlotOption( Global:UnexpandedList, MULTIPLE );
SetSlotOption( Global:UnexpandedList, VALUE_TYPE, OBJECT );
SetSlotOption( Global:UnexpandedList, ALLOWABLE_CLASSES, Links );
ClearList( Global:UnexpandedList );
SetSlotOption( Global:UnexpandedList, IF_NEEDED, NULL );
SetSlotOption( Global:UnexpandedList, WHEN_ACCESS, NULL );
SetSlotOption( Global:UnexpandedList, BEFORE_CHANGE, NULL );
SetSlotOption( Global:UnexpandedList, AFTER_CHANGE, NULL );
MakeSlot( Global:ExpandedList );
SetSlotOption( Global:ExpandedList, INHERIT, FALSE );
SetSlotOption( Global:ExpandedList, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:ExpandedList, MULTIPLE );
SetSlotOption( Global:ExpandedList, VALUE_TYPE, OBJECT );
SetSlotOption( Global:ExpandedList, ALLOWABLE_CLASSES, Links );
ClearList( Global:ExpandedList );
SetSlotOption( Global:ExpandedList, IF_NEEDED, NULL );
SetSlotOption( Global:ExpandedList, WHEN_ACCESS, NULL );
SetSlotOption( Global:ExpandedList, BEFORE_CHANGE, NULL );
SetSlotOption( Global:ExpandedList, AFTER_CHANGE, NULL );
MakeSlot( Global:X );
SetSlotOption( Global:X, INHERIT, FALSE );
SetSlotOption( Global:X, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:X, VALUE_TYPE, NUMBER );
Global:X = -10;
MakeSlot( Global:NLinkList );
SetSlotOption( Global:NLinkList, INHERIT, FALSE );
SetSlotOption( Global:NLinkList, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:NLinkList, MULTIPLE );
SetValue( Global:NLinkList, Flow_5, Flow_6, Flow_7 );
SetSlotOption( Global:NLinkList, IF_NEEDED, NULL );
SetSlotOption( Global:NLinkList, WHEN_ACCESS, NULL );
SetSlotOption( Global:NLinkList, BEFORE_CHANGE, NULL );
SetSlotOption( Global:NLinkList, AFTER_CHANGE, NULL );
MakeSlot( Global:Active );
SetSlotOption( Global:Active, INHERIT, FALSE );
SetSlotOption( Global:Active, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:Active, VALUE_TYPE, BOOLEAN );
Global:Active = FALSE;
SetSlotOption( Global:Active, IF_NEEDED, NULL );
SetSlotOption( Global:Active, WHEN_ACCESS. NULL );
SetSlotOption( Global:Active, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Active, AFTER_CHANGE, NULL );
MakeSlot( Global:CTime );
SetSlotOption( Global:CTime, INHERIT, FALSE );
SetSlotOption( Global:CTime, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:CTime, MULTIPLE );
SetSlotOption( Global:CTime, VALUE_TYPE, NUMBER );
SetSlotOption( Global:CTime, MINIMUM_VALUE, 0 );
ClearList( Global:CTime );
MakeSlot( Global:StdDelay );
SetSlotOption( Global:StdDelay, INHERIT, FALSE );
SetSlotOption( Global:StdDelay, NO_AUTO_ASK, TRUE );
```

```
SetSlotOption( Global:StdDelay, VALUE_TYPE, NUMBER );
SetSlotOption( Global:StdDelay, MINIMUM_VALUE, 0 );
Global:StdDelay = 300;
SetSlotOption( Global:StdDelay, IF_NEEDED, NULL );
SetSlotOption( Global:StdDelay, WHEN_ACCESS, NULL );
SetSlotOption( Global:StdDelay, BEFORE_CHANGE, NULL );
SetSlotOption( Global:StdDelay, AFTER_CHANGE, NULL );
MakeSlot( Global:ListTemp );
SetSlotOption( Global:ListTemp, INHERIT, FALSE );
SetSlotOption( Global:ListTemp, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:ListTemp, VALUE_TYPE, BOOLEAN );
Global:ListTemp = FALSE;
SetSlotOption( Global:ListTemp, IF_NEEDED, NULL );
SetSlotOption( Global:ListTemp, WHEN_ACCESS, NULL );
SetSlotOption( Global:ListTemp, BEFORE_CHANGE, NULL );
SetSlotOption( Global:ListTemp, AFTER_CHANGE, NULL );
MakeSlot( Global:Counter );
SetSlotOption( Global:Counter, INHERIT, FALSE );
SetSlotOption( Global:Counter, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:Counter, VALUE_TYPE, NUMBER );
Global:Counter = 8;
SetSlotOption( Global:Counter, IF_NEEDED, NULL );
SetSlotOption( Global:Counter, WHEN_ACCESS, NULL );
SetSlotOption( Global:Counter, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Counter, AFTER_CHANGE, NULL );
MakeSlot( Global:Quit );
SetSlotOption( Global:Quit, INHERIT, FALSE );
SetSlotOption( Global:Quit, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:Quit, VALUE_TYPE, BOOLEAN );
Global:Quit = TRUE;
SetSlotOption( Global:Quit, IF_NEEDED, NULL );
SetSlotOption( Global:Quit, WHEN_ACCESS, NULL );
SetSlotOption( Global:Quit, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Quit, AFTER_CHANGE, NULL );
MakeSlot( Global:Item );
Global:Item = P02;
MakeSlot( Global:ActionList );
SetSlotOption( Global:ActionList, INHERIT, FALSE );
SetSlotOption( Global:ActionList, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:ActionList, MULTIPLE );
SetSlotOption( Global:ActionList, VALUE_TYPE, OBJECT );
ClearList( Global:ActionList );
SetSlotOption( Global:ActionList, IMAGE, MultipleListBox1 );
MakeSlot( Global:ActionSpacing );
SetSlotOption( Global:ActionSpacing, INHERIT, FALSE );
SetSlotOption( Global:ActionSpacing, VALUE_TYPE, NUMBER );
SetSlotOption( Global:ActionSpacing, MINIMUM_VALUE, 0 );
Global:ActionSpacing = 30;
SetSlotOption( Global:ActionSpacing, IF_NEEDED, NULL );
SetSlotOption( Global:ActionSpacing, WHEN_ACCESS, NULL );
SetSlotOption( Global:ActionSpacing, BEFORE_CHANGE, NULL );
SetSlotOption( Global:ActionSpacing, AFTER_CHANGE, NULL );
```

```
MakeSlot( Global:PauseTime );
SetSlotOption( Global:PauseTime, INHERIT, FALSE );
SetSlotOption( Global:PauseTime, VALUE_TYPE, NUMBER );
Global:PauseTime = 334938.091209;
SetSlotOption( Global:PauseTime, IF_NEEDED, NULL );
SetSlotOption( Global:PauseTime, WHEN_ACCESS, NULL );
SetSlotOption( Global:PauseTime, BEFORE_CHANGE, NULL );
SetSlotOption( Global:PauseTime, AFTER_CHANGE, NULL );
MakeSlot( Global:ReadList );
SetSlotOption( Global:ReadList, INHERIT, FALSE );
SetSlotOption( Global:ReadList, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:ReadList, MULTIPLE );
SetValues( Global:ReadList, LIP001, FIP006, TIP016, TIP017, TIP018, TIP019, FIP010, FIP030,
FIP012 );
SetSlotOption( Global:ReadList, IF_NEEDED, NULL );
SetSlotOption( Global:ReadList, WHEN_ACCESS, NULL );
SetSlotOption( Global:ReadList, BEFORE_CHANGE, NULL );
SetSlotOption( Global:ReadList, AFTER_CHANGE, NULL );
MakeSlot( Global:StartTime );
SetSlotOption( Global:StartTime, INHERIT, FALSE );
SetSlotOption( Global:StartTime, VALUE_TYPE, NUMBER );
SetSlotOption( Global:StartTime, MINIMUM_VALUE, 0 );
Global:StartTime = 338410.114286;
SetSlotOption( Global:StartTime, IF_NEEDED, NULL );
SetSlotOption( Global:StartTime, WHEN_ACCESS, NULL );
SetSlotOption( Global:StartTime, BEFORE_CHANGE, NULL );
SetSlotOption( Global:StartTime, AFTER_CHANGE, NULL );
MakeSlot( Global:Failures );
SetSlotOption( Global:Failures, INHERIT, FALSE );
SetSlotOption( Global:Failures, VALUE_TYPE, NUMBER );
SetSlotOption( Global:Failures, MINIMUM_VALUE, 0 );
Global:Failures = 115;
SetSlotOption( Global:Failures, IF_NEEDED, NULL );
SetSlotOption( Global:Failures, WHEN_ACCESS, NULL );
SetSlotOption( Global:Failures, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Failures, AFTER_CHANGE, Percent );
MakeSlot( Global:Successes );
SetSlotOption( Global:Successes, INHERIT, FALSE );
SetSlotOption( Global:Successes, VALUE_TYPE, NUMBER );
Global:Successes = 178;
SetSlotOption( Global:Successes, IF_NEEDED, NULL );
SetSlotOption( Global:Successes, WHEN_ACCESS, NULL );
SetSlotOption( Global:Successes, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Successes, AFTER_CHANGE, Percent );
MakeSlot( Global:Percent );
SetSlotOption( Global:Percent, INHERIT, FALSE );
SetSlotOption( Global:Percent, NO_AUTO_ASK, TRUE );
SetSlotOption( Global:Percent, VALUE_TYPE, NUMBER );
SetSlotOption( Global:Percent, MINIMUM_VALUE, 0 );
SetSlotOption( Global:Percent, MAXIMUM_VALUE, 100 );
Global:Percent = 39.2491467576792;
SetSlotOption( Global:Percent, IF_NEEDED, NULL );
```

```
SetSlotOption( Global:Percent, WHEN_ACCESS, NULL );
SetSlotOption( Global:Percent, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Percent, AFTER_CHANGE, NULL );
MakeSlot( Global:AVTime );
SetSlotOption( Global:AVTime, INHERIT, FALSE );
SetSlotOption( Global:AVTime, VALUE_TYPE, NUMBER );
Global:AVTime = 34.4251162506422;
SetSlotOption( Global:AVTime, IF_NEEDED, NULL );
SetSlotOption( Global:AVTime, WHEN_ACCESS, NULL );
SetSlotOption( Global:AVTime, BEFORE_CHANGE, AVtime );
SetSlotOption( Global:AVTime, AFTER_CHANGE, NULL );
MakeSlot( Global:AVTimeS );
SetSlotOption( Global:AVTimeS, INHERIT, FALSE );
SetSlotOption( Global:AVTimeS, VALUE_TYPE, NUMBER );
SetSlotOption( Global:AVTimeS, MINIMUM_VALUE, 0 );
Global:AVTimeS = 334856.278022;
SetSlotOption( Global:AVTimeS, IF_NEEDED, NULL );
SetSlotOption( Global:AVTimeS, WHEN_ACCESS, NULL );
SetSlotOption( Global:AVTimeS, BEFORE_CHANGE, NULL );
SetSlotOption( Global:AVTimeS, AFTER_CHANGE, NULL );
MakeSlot( Global:AVTimeF );
SetSlotOption( Global:AVTimeF, INHERIT, FALSE );
SetSlotOption( Global:AVTimeF, VALUE_TYPE, NUMBER );
SetSlotOption( Global:AVTimeF, MINIMUM_VALUE, 0 );
Global:AVTimeF = 334920.289011;
SetSlotOption( Global:AVTimeF, IF_NEEDED, NULL );
SetSlotOption( Global:AVTimeF, WHEN_ACCESS, NULL );
SetSlotOption( Global:AVTimeF, BEFORE_CHANGE, NULL );
SetSlotOption( Global:AVTimeF, AFTER_CHANGE, CalcAV );
MakeSlot( Global:StationID );
SetSlotOption( Global:StationID, INHERIT, FALSE );
Global:StationID = 00;
SetSlotOption( Global:StationID, IF_NEEDED, NULL );
SetSlotOption( Global:StationID, WHEN_ACCESS, NULL );
SetSlotOption( Global:StationID, BEFORE_CHANGE, NULL );
SetSlotOption( Global:StationID, AFTER_CHANGE, NULL );
MakeSlot( Global:Read );
SetSlotOption( Global:Read, INHERIT, FALSE );
SetSlotOption( Global:Read, VALUE_TYPE, BOOLEAN );
Global:Read = FALSE;
SetSlotOption( Global:Read, IF_NEEDED, NULL );
SetSlotOption( Global:Read, WHEN_ACCESS, NULL );
SetSlotOption( Global:Read, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Read, AFTER_CHANGE, NULL );
SetSlotOption( Global:Read, IMAGE, CheckBox5, CheckBox1 );
MakeSlot( Global:Write );
SetSlotOption( Global:Write, INHERIT, FALSE );
SetSlotOption( Global:Write, VALUE_TYPE, BOOLEAN );
Global:Write = FALSE;
SetSlotOption( Global:Write, IF_NEEDED, NULL );
SetSlotOption( Global:Write, WHEN_ACCESS, NULL );
SetSlotOption( Global:Write, BEFORE_CHANGE, NULL );
```

```
SetSlotOption( Global:Write, AFTER_CHANGE, NULL );
SetSlotOption( Global:Write, IMAGE, CheckBox6 );
MakeSlot( Global:ReadState );
SetSlotOption( Global:ReadState, INHERIT, FALSE );
SetSlotOption( Global:ReadState, ALLOWABLE_VALUES, READ, WRITE, OPEN, CLOSE );
Global:ReadState = READ;
SetSlotOption( Global:ReadState, IF_NEEDED, NULL );
SetSlotOption( Global:ReadState, WHEN_ACCESS, NULL );
SetSlotOption( Global:ReadState, BEFORE_CHANGE, NULL );
SetSlotOption( Global:ReadState, AFTER_CHANGE, NULL );
SetSlotOption( Global:ReadState, IMAGE, RadioButtonGroup2, RadioButtonGroup4 );
MakeSlot( Global:RunState );
SetSlotOption( Global:RunState, INHERIT, FALSE );
SetSlotOption( Global:RunState, ALLOWABLE_VALUES, Running, Pause );
Global:RunState = Running;
SetSlotOption( Global:RunState, IF_NEEDED, NULL );
SetSlotOption( Global:RunState, WHEN_ACCESS, NULL );
SetSlotOption( Global:RunState, BEFORE_CHANGE, NULL );
SetSlotOption( Global:RunState, AFTER_CHANGE, NULL );
SetSlotOption( Global:RunState, IMAGE, RadioButtonGroup5 );
MakeSlot( Global:BackUpRate );
SetSlotOption( Global:BackUpRate, INHERIT, FALSE );
SetSlotOption( Global:BackUpRate, VALUE_TYPE, NUMBER );
SetSlotOption( Global:BackUpRate, MINIMUM_VALUE, 0 );
Global:BackUpRate = 20;
SetSlotOption( Global:BackUpRate, IF_NEEDED, NULL );
SetSlotOption( Global:BackUpRate, WHEN_ACCESS, NULL );
SetSlotOption( Global:BackUpRate, BEFORE_CHANGE, NULL );
SetSlotOption( Global:BackUpRate, AFTER_CHANGE, BackUpSet );
MakeSlot( Global:Response );
SetSlotOption( Global:Response, INHERIT, FALSE );
SetSlotOption( Global:Response, VALUE_TYPE, BOOLEAN );
Global:Response = FALSE;
SetSlotOption( Global:Response, IF_NEEDED, NULL );
SetSlotOption( Global:Response, WHEN_ACCESS, NULL );
SetSlotOption( Global:Response, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Response, AFTER_CHANGE, NULL );
MakeSlot( Global:Busy );
SetSlotOption( Global:Busy, INHERIT, FALSE );
SetSlotOption( Global:Busy, ALLOWABLE_VALUES, BUSY, PAUSE );
Global:Busy = BUSY;
SetSlotOption( Global:Busy, IF_NEEDED, NULL );
SetSlotOption( Global:Busy, WHEN_ACCESS, NULL );
SetSlotOption( Global:Busy, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Busy, AFTER_CHANGE, NULL );
SetSlotOption( Global:Busy, IMAGE, StateBox2 );
MakeSlot( Global:ActionFlag );
SetSlotOption( Global:ActionFlag, INHERIT, FALSE );
SetSlotOption( Global:ActionFlag, VALUE_TYPE, BOOLEAN );
Global:ActionFlag = FALSE;
MakeSlot( Global:TempAction );
Global:TempAction = NULL;
```

```
MakeSlot( Global:ItemNo );
SetSlotOption( Global:ItemNo, INHERIT, FALSE );
Global:ItemNo = 4;
SetSlotOption( Global:ItemNo, IF_NEEDED, NULL );
SetSlotOption( Global:ItemNo, WHEN_ACCESS, NULL );
SetSlotOption( Global:ItemNo, BEFORE_CHANGE, NULL );
SetSlotOption( Global:ItemNo, AFTER_CHANGE, NULL );
MakeSlot( Global:VarNo );
SetSlotOption( Global:VarNo, INHERIT, FALSE );
Global:VarNo = 3;
MakeSlot( Global:LinkChain );
SetSlotOption( Global:LinkChain, ALLOWABLE_VALUES, ALWAYS, NEEDED );
Global:LinkChain = ALWAYS;
MakeSlot( Global:NewLinks );
SetSlotOption( Global:NewLinks, MULTIPLE );
SetSlotOption( Global:NewLinks, VALUE_TYPE, OBJECT );
SetSlotOption( Global:NewLinks, ALLOWABLE_CLASSES, Links );
ClearList( Global:NewLinks );
MakeSlot( Global:Change );
SetSlotOption( Global:Change, VALUE_TYPE, BOOLEAN );
Global:Change = FALSE;
MakeSlot( Global:SimulationSpacing );
SetSlotOption( Global:SimulationSpacing, INHERIT, FALSE );
SetSlotOption( Global:SimulationSpacing, VALUE_TYPE, NUMBER );
Global:SimulationSpacing = 1;
SetSlotOption( Global:SimulationSpacing, IF_NEEDED, NULL );
SetSlotOption( Global:SimulationSpacing, WHEN_ACCESS, NULL );
SetSlotOption( Global:SimulationSpacing, BEFORE_CHANGE, NULL );
SetSlotOption( Global:SimulationSpacing, AFTER_CHANGE, NULL );
MakeSlot( Global:Simulation );
SetSlotOption( Global:Simulation, INHERIT, FALSE );
SetSlotOption( Global:Simulation, VALUE_TYPE, BOOLEAN );
Global:Simulation = TRUE;
SetSlotOption( Global:Simulation, IF_NEEDED, NULL );
SetSlotOption( Global:Simulation, WHEN_ACCESS, NULL );
SetSlotOption( Global:Simulation, BEFORE_CHANGE, NULL );
SetSlotOption( Global:Simulation, AFTER_CHANGE, NULL );

    /*************************************
     **** INSTANCE: SESSION
     *************************************/
SESSION:X = 2;
SESSION:Y = 2;
SESSION:Title = "HAL EXPERT Ver 1.4";
SESSION:SessionNumber = 0;
SESSION:Width = 532;
SESSION:Height = 485;
SESSION:Visible = FALSE;
SESSION:State = HIDDEN;
SESSION:Menu = TRUE;
SESSION:Titlebar = TRUE;
SESSION:Sizebox = FALSE;
```

```
SetValue( SESSION:BackgroundColor, 0, 128, 255 );
SESSION:Freeze = FALSE;
 ResetWindow ( SESSION );

     /*****************************************
      **** INSTANCE: Choice
      *****************************************/
MakeInstance( Choice, Menu );

/************** METHOD: DisplayCall  **************/
MakeMethod( Choice, DisplayCall, [owner window choice ],
   {
   choice;
   } );
Choice:X = 421;
Choice:Y = 238;
SetValue( Choice:Choices, Edit, Delete, Rename );
SetValue( Choice:ChoiceNames, &Edit, &Delete, &Rename );

     /*****************************************
      **** INSTANCE: Quicksort
      **** Sort routine. It uses the shell short.
      *****************************************/
MakeInstance( Quicksort, Root );
SetInstanceComment( Quicksort, "Sort routine. It uses the shell short." );

/************** METHOD: Shell  **************/
MakeMethod( Quicksort, Shell, [List Slot ],
   {
   ClearList( Self:Item );
   AppendToList( Self:Item, List );
   Self:Count = LengthList( Self:Item );
   EnumList( Self:a, gap,
      {
      If ( gap < Self:Count )
       Then {
           For i From gap To Self:Count By 1
                Do {
               Self:x = GetNthElem( Self:Item, i );
               Self:j = i - gap;
               While  (( ( Self:j - gap >= 0 ) And ( GetValue( GetValue( Self:x ),
                                                    Slot )
                                                    <
                                                    GetValue( GetNthElem( Self:Item,
                                                                Self:j ),
                                                       Slot ) ) ))
                     {
                     SetNthElem( Self:Item, Self:j + gap,
                        GetNthElem( Self:Item, Self:j ) );
                     SetNthElem( Self:Item,
                        {
                        Self:j;
```

```
                    }, Self:x );
                Self:j = Self:j - gap;
                };
            };
        };
    } );
  Self:Item;
  } );
MakeSlot( Quicksort:j );
SetSlotOption( Quicksort:j, VALUE_TYPE, NUMBER );
Quicksort:j = 4;
MakeSlot( Quicksort:x );
SetSlotOption( Quicksort:x, INHERIT, FALSE );
Quicksort:x = Flow_8;
SetSlotOption( Quicksort:x, IF_NEEDED, NULL );
SetSlotOption( Quicksort:x, WHEN_ACCESS, NULL );
SetSlotOption( Quicksort:x, BEFORE_CHANGE, NULL );
SetSlotOption( Quicksort:x, AFTER_CHANGE, NULL );
MakeSlot( Quicksort:k );
MakeSlot( Quicksort:a );
SetSlotOption( Quicksort:a, INHERIT, FALSE );
SetSlotOption( Quicksort:a, NO_AUTO_ASK, TRUE );
SetSlotOption( Quicksort:a, MULTIPLE );
SetSlotOption( Quicksort:a, VALUE_TYPE, NUMBER );
SetSlotOption( Quicksort:a, MINIMUM_VALUE, 1 );
SetValue( Quicksort:a, 9, 5, 3, 2, 1 );
SetSlotOption( Quicksort:a, IF_NEEDED, NULL );
SetSlotOption( Quicksort:a, WHEN_ACCESS, NULL );
SetSlotOption( Quicksort:a, BEFORE_CHANGE, NULL );
SetSlotOption( Quicksort:a, AFTER_CHANGE, NULL );
MakeSlot( Quicksort:Count );
SetSlotOption( Quicksort:Count, INHERIT, FALSE );
SetSlotOption( Quicksort:Count, NO_AUTO_ASK, TRUE );
SetSlotOption( Quicksort:Count, VALUE_TYPE, NUMBER );
Quicksort:Count = 1;
SetSlotOption( Quicksort:Count, IF_NEEDED, NULL );
SetSlotOption( Quicksort:Count, WHEN_ACCESS, NULL );
SetSlotOption( Quicksort:Count, BEFORE_CHANGE, NULL );
SetSlotOption( Quicksort:Count, AFTER_CHANGE, NULL );
MakeSlot( Quicksort:Item );
SetSlotOption( Quicksort:Item, INHERIT, FALSE );
SetSlotOption( Quicksort:Item, NO_AUTO_ASK, TRUE );
SetSlotOption( Quicksort:Item, MULTIPLE );
SetSlotOption( Quicksort:Item, VALUE_TYPE, OBJECT );
SetSlotOption( Quicksort:Item, ALLOWABLE_CLASSES, Links, Nodes );
SetValue( Quicksort:Item, CVP01Switch );
SetSlotOption( Quicksort:Item, IF_NEEDED, NULL );
SetSlotOption( Quicksort:Item, WHEN_ACCESS, NULL );
SetSlotOption( Quicksort:Item, BEFORE_CHANGE, NULL );
SetSlotOption( Quicksort:Item, AFTER_CHANGE, NULL );

/*****************************************
```

```
      **** INSTANCE: CostSlf
      **** Data used by function CostSlf.
      ***************************************/
MakeInstance( CostSlf, Root );
SetInstanceComment( CostSlf, "Data used by function CostSlf." );
MakeSlot( CostSlf:Cost );
SetSlotOption( CostSlf:Cost, INHERIT, FALSE );
SetSlotOption( CostSlf:Cost, NO_AUTO_ASK, TRUE );
SetSlotOption( CostSlf:Cost, VALUE_TYPE, NUMBER );
SetSlotOption( CostSlf:Cost, MINIMUM_VALUE, -1 );
SetSlotOption( CostSlf:Cost, MAXIMUM_VALUE, 1 );
CostSlf:Cost = 0;
SetSlotOption( CostSlf:Cost, IF_NEEDED, NULL );
SetSlotOption( CostSlf:Cost, WHEN_ACCESS, NULL );
SetSlotOption( CostSlf:Cost, BEFORE_CHANGE, NULL );
SetSlotOption( CostSlf:Cost, AFTER_CHANGE, NULL );
MakeSlot( CostSlf:X );
SetSlotComment( CostSlf:X, "OrignalValue + Variation" );
SetSlotOption( CostSlf:X, VALUE_TYPE, NUMBER );
CostSlf:X = -1;
MakeSlot( CostSlf:Temp );
SetSlotOption( CostSlf:Temp, INHERIT, FALSE );
SetSlotOption( CostSlf:Temp, NO_AUTO_ASK, TRUE );
SetSlotOption( CostSlf:Temp, VALUE_TYPE, NUMBER );
CostSlf:Temp = NULL;
SetSlotOption( CostSlf:Temp, IF_NEEDED, NULL );
SetSlotOption( CostSlf:Temp, WHEN_ACCESS, NULL );
SetSlotOption( CostSlf:Temp, BEFORE_CHANGE, NULL );
SetSlotOption( CostSlf:Temp, AFTER_CHANGE, NULL );

    /***************************************
     **** INSTANCE: RuleEdit
     ***************************************/
MakeInstance( RuleEdit, Menu );

/************ METHOD: Rule ************/
MakeMethod( RuleEdit, Rule, [owner window choice ],
  {
  Self:Choice = PostMenu( RuleChoices );
  If ( Self:Choice #= Delete Or Self:Choice #= Rename )
   Then {
      If ( LengthList( Rule:choice ) == 0 )
       Then {
          PostMessage( "No Rules were found" );
          Self:Selection = NULL;
          }
       Else {
          Self:Selection = PostMenu( "Select Rule", Rule:choice );
          };
      };
  If ( ( Self:Choice #= Delete ) And Not( Null?( Self:Selection ) ) )
   Then {
```

```
        SendMessage( Self, Delete, owner, window, choice );
        };
    If ( ( Self:Choice #= Rename ) And Not( Null?( Self:Selection ) ) )
      Then SendMessage( Self, Rename, owner, window, choice );
    If ( Self:Choice #= New )
      Then SendMessage( Self, New, owner, window, choice );
    If ( Self:Choice #= Edit )
      Then SendMessage( Self, Edit );
    } );

/************** METHOD: New  ***************/
MakeMethod( RuleEdit, New, [owner window choice ],
    {
    Global:Change = TRUE;
    Self:Check = TRUE;
    While  (Self:Check)
        {
        Self:name = NULL;
        PostInputForm( " Name Rule ", Self, name, Name );
        If ( Not( Null?( Self.name ) ) And Not( Rule?( Self:name ) ) )
          Then {
            Self:Check = FALSE;
            AppendToList( Rule:choice, Self:name );
            If ( choice #= ControlRules )
              Then MakeRule( Self:name, [ Item | Items ],
                    Item:Variable < ?, Obj( Item?, Variable?,
                                            Priority?, Variation?,
                                            CriticalTime?,
                                            RepeatTime?,
                                            FALSE, Message?,
                                            Name?, Num? ) );
            If ( choice #= LinkRules )
              Then MakeRule( Self:name, [ Node | Nodes ],
                    Node:Item #= ?, List( Node:Item, Primary
                                                    -
                                                    Item?,
                                          Node:Variable,
                                          Primary_Var?, TREL?,
                                          VSREL?, VPREL?,
                                          FALSE, -100?, 100?,
                                          L? ) );
            If ( choice #= DefaultRules )
              Then MakeRule( Self:name, [ Node | Nodes ],
                    Node:Item #= ? And Node:Variable #=
                      ?, SendMessage( D_ActionName_,
                          Activate, Control, 10 ) );
            If ( choice #= CostRules )
              Then MakeRule( Self:name, [ Link | Links ],
                    Link:PItem #= ? And Link:PVariable
                      #= ? And Link:Variation > 0, SendMessage( Link,
                                                    AltCost,
                                                    ? ) );
```

```
            If ( choice #= ValueRules )
              Then MakeRule( Self:name, [ Item | Items ],
                    Item?:Variable? #= ?, SendMessage( Item,
                                          SetVar, Variable?,
                                          Value? ) );
            If ( choice #= ActionRules )
              Then MakeRule( Self:name, [ Action | Actions ],
                    Action:Item #= ? And Action:Variable
                      #= ? And Action:Variation < >,
                    SendMessage( Action, AltCost, ? ) );
            };
        If Null?( Self:name )
          Then {
            Self:Check = FALSE;
            };
        If Self:Check
          Then PostMessage( "ERROR - Rule already exists" );
        };
    HideWindow( RULERELATIONS );
    ShowWindow( RULERELATIONS );
    } );

/************ METHOD: Edit ************/
MakeMethod( RuleEdit, Edit, [],
    {
    Global:Change = TRUE;
    HideWindow( RULERELATIONS );
    ShowWindow( RULERELATIONS );
    } );

/************ METHOD: Rename ************/
MakeMethod( RuleEdit, Rename, [owner window choice ],
    {
    Global:Change = TRUE;
    Self:Check = TRUE;
    While (Self:Check)
        {
        Self:name = Self:Selection;
        PostInputForm( " Rename Rule " # Self:Selection, Self:name,
            "New name" );
        If ( Not( Rule?( Self:name ) ) And Not( Null?( Self:name ) )
              And RenameRule( Self:Selection, Self:name ) )
          Then {
            Self:Check = FALSE;
            RemoveFromList( Rule:choice, Self:Selection );
            AppendToList( Rule:choice, Self:name );
            };
        If ( Null?( Self:name ) Or Self:name #= Self:Selection )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR - Rule already exists" );
        };
```

```
  } );

/************** METHOD: Delete  **************/
MakeMethod( RuleEdit, Delete, [owner window choice ],
  {
  Global:Change = TRUE;
  If ( PostMenu( "Delete rule " # Self:Selection # ?, YES, NO )
       #= YES )
    Then {
      If Rule?( Self:Selection )
        Then DeleteRule( Self:Selection );
      If Member?( Rule:choice, Self:Selection )
        Then RemoveFromList( Rule:choice, Self:Selection );
      };
  } );
RuleEdit:X = 157;
RuleEdit:Y = 26;
SetValue( RuleEdit:Choices, ControlRules, ValueRules, LinkRules, DefaultRules, ActionRules,
CostRules );
SetValue( RuleEdit:ChoiceNames, &Control, &Value, &Link, &Default, "Ac&tion Cost", "L&ink Cost"
);
RuleEdit:DefaultMethod = Rule;
MakeSlot( RuleEdit:Selection );
RuleEdit:Selection = L_LevelDLevel_1;
MakeSlot( RuleEdit:Choice );
RuleEdit:Choice = New;
MakeSlot( RuleEdit:name );
RuleEdit:name = L_FlowOut;
MakeSlot( RuleEdit:Check );
SetSlotOption( RuleEdit:Check, INHERIT, FALSE );
SetSlotOption( RuleEdit:Check, VALUE_TYPE, BOOLEAN );
RuleEdit:Check = FALSE;
SetSlotOption( RuleEdit:Check, IF_NEEDED, NULL );
SetSlotOption( RuleEdit:Check, WHEN_ACCESS, NULL );
SetSlotOption( RuleEdit:Check, BEFORE_CHANGE, NULL );
SetSlotOption( RuleEdit:Check, AFTER_CHANGE, NULL );

    /*************************************
     **** INSTANCE: RuleChoice
     *************************************/
MakeInstance( RuleChoice, Menu );

/************** METHOD: InitMenu  **************/
MakeMethod( RuleChoice, InitMenu, [],
  {} );

/************** METHOD: PrepareMenu  **************/
MakeMethod( RuleChoice, PrepareMenu, [],
  NULL );

/************** METHOD: Call  **************/
MakeMethod( RuleChoice, Call, [owner window choice ],
```

```
    {
    choice;
    } );
RuleChoice:X = 234;
RuleChoice:Y = 61;
SetValue( RuleChoice:Choices, New, Edit, Delete, Rename );
SetValue( RuleChoice:ChoiceNames, &New, &Edit, &Delete, &Rename );
ClearList( RuleChoice:Checked );
RuleChoice:DefaultMethod = Call;

      /*************************************
       **** INSTANCE: ItemEdit
       *************************************/
MakeInstance( ItemEdit, Menu );

/************** METHOD: Delete  **************/
MakeMethod( ItemEdit, Delete, [owner window choice ],
   {
   If ( PostMenu( "Delete " # owner # " ?", YES, NO ) #= YES )
    Then {
        If Class?( A_ # owner )
         Then {
             ClearList( Self:TempList );
             GetInstanceList( A_ # owner, Self:TempList );
             EnumList( Self:TempList, A,
                {
                DeleteInstance( A );
                } );
             DeleteClass( A_ # owner );
             };
        If Class?( L_ # owner )
         Then {
             ClearList( Self:TempList );
             GetInstanceList( L_ # owner, Self:TempList );
             EnumList( Self:TempList, A,
                {
                DeleteInstance( A );
                } );
             DeleteClass( L_ # owner );
             };
        EnumList( owner:ValidVars, VD,
           {
           If Slot?( owner, N # VD )
            Then {
                If ( Not( Null?( GetValue( owner, N # VD ) ) )
                       And ( GetValue( owner, N # VD )
                              #= owner # _ # VD ) )
                 Then {
                     If Instance?( owner # _ # VD )
                      Then DeleteInstance( owner # _
                                           # VD );
                     If ( Class?( VD ) And ( CountInstances( VD )
```

```
                                                == 0 ) )
                        Then {
                            DeleteClass( VD );
                            RemoveFromList( Items:ValidVars,
                               VD );
                            };
                        };
                    };
                } );
            DeleteInstance( owner );
            };
    } );

/************** METHOD: Edit  **************/
MakeMethod( ItemEdit, Edit, [],
    NULL );

/************** METHOD: Move  **************/
MakeMethod( ItemEdit, Move, [owner window choice ],
    {
    Self:Check = TRUE;
    While  (Self:Check)
        {
        PostInputForm( "Move " # owner, Self:name, "New parent" );
        If ( Class?( Self:name ) And Not( Null?( Self:name ) )
                And MoveInstance( owner, Self:name ) )
          Then Self:Check = FALSE;
        If ( Null?( Self:name ) Or Self:name #= GetParent( owner ) )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR -  Class does not exists" );
        };
    } );

/************** METHOD: Rename  **************/
MakeMethod( ItemEdit, Rename, [owner window choice ],
    {
    Self:Check = TRUE;
    While  (Self:Check)
        {
        Self:name = owner;
        PostInputForm( " Rename Instance " # owner, Self:name,
            "New name" );
        If ( Not( Instance?( Self:name ) ) And Not( Null?( Self:name ) )
                And RenameInstance( owner, Self:name ) )
          Then {
              Self:Check = FALSE;
              If Class?( A_ # owner )
                Then RenameClass( A_ # owner, A_ # Self:name );
              If Class?( L_ # owner )
                Then RenameClass( L_ # owner, L_ # Self:name );
              EnumList( Self:name:ValidVars, VD,
```

```
                {
                If Slot?( GetValue( Self:name ),
                      N # VD )
                  Then {
                      If ( Not( Null?( GetValue( GetValue( Self:name ),
                                  N # VD ) ) )
                            And ( GetValue( GetValue( Self:name ),
                                  N # VD ) #= owner
                                  # _ # VD ) )
                        Then {
                            SetValue( GetValue( Self:name ),
                              N # VD, RenameInstance( owner
                                                      #
                                                      _
                                                      #
                                                      VD,
                                                Self:name #
                                                  _ # VD ) );
                            };
                      };
                  } );
                };
        If ( Null?( Self:name ) Or Self:name #= owner )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR - Instance already exists" );
        };
    } );

/************** METHOD: ActionEd **************/
MakeMethod( ItemEdit, ActionEd, [owner window choice ],
    {
    If Not( Class?( A_ # owner ) )
      Then MakeClass( A_ # owner, Actions );
    HideWindow( BROWSER );
    If Function?( ShowInstanceMenu )
      Then DeleteFunction( ShowInstanceMenu );
    MakeFunction( ShowInstanceMenu, [ obj ], PopupMenu( ActionEdit,
                                      obj, BROWSER ) );
    ShowInstanceMenu;
    If Function?( ShowClassMenu )
      Then DeleteFunction( ShowClassMenu );
    MakeFunction( ShowClassMenu, [ obj ], PopupMenu( ActionEdit2,
                                    obj, BROWSER ) );
    ShowClassMenu;
    PositionWindow( BROWSER, 70, 70, 500, 300 );
    Global:Item = owner;
    SetBrowserFocus( A_ # owner );
    SetWindowTitle( BROWSER, "Action Browser" );
    ShowWindow( BROWSER );
    } );
```

```
/************** METHOD: LinkEd  **************/
MakeMethod( ItemEdit, LinkEd, [owner window choice ],
   {
   If Class?( L_ # owner )
    Then {
        HideWindow( BROWSER );
        If Function?( ShowInstanceMenu )
          Then DeleteFunction( ShowInstanceMenu );
        MakeFunction( ShowInstanceMenu, [ obj ],
           PopupMenu( LinkEdit, obj, BROWSER ) );
        ShowInstanceMenu;
        If Function?( ShowClassMenu )
          Then DeleteFunction( ShowClassMenu );
        MakeFunction( ShowClassMenu, [ obj ], PopupMenu( LinkEdit2,
                                        obj, BROWSER ) );
        ShowClassMenu;
        PositionWindow( BROWSER, 70, 70, 500, 300 );
        SetMenuBar( BROWSER, BrowserMenu );
        SetBrowserFocus( L_ # owner );
        SetWindowTitle( BROWSER, "Link Browser" );
        ShowWindow( BROWSER );
        }
     Else {
        PostMessage( "ERROR - There are no links avaliable" );
        };
   } );

/************** METHOD: DefaultEd  **************/
MakeMethod( ItemEdit, DefaultEd, [owner window choice ],
   {
   If Not( Class?( Default ) )
     Then MakeClass( Default, Actions );
   HideWindow( BROWSER );
   If Function?( ShowInstanceMenu )
     Then DeleteFunction( ShowInstanceMenu );
   MakeFunction( ShowInstanceMenu, [ obj ], PopupMenu( ActionEdit,
                                        obj, BROWSER ) );
   ShowInstanceMenu;
   If Function?( ShowClassMenu )
     Then DeleteFunction( ShowClassMenu );
   MakeFunction( ShowClassMenu, [ obj ], PopupMenu( ActionEdit2,
                                      obj, BROWSER ) );
   ShowClassMenu;
   PositionWindow( BROWSER, 70, 70, 500, 300 );
   Global:Item = owner;
   SetBrowserFocus( Default );
   SetWindowTitle( BROWSER, "Default Actions" );
   ShowWindow( BROWSER );
   } );
SetSlotOption( ItemEdit:X, NO_AUTO_ASK, TRUE );
ItemEdit:X = 296;
ItemEdit:Y = 106;
```

```
SetValue( ItemEdit:Choices, ItemEdit2, Rename, Delete, Move, -, ActionEd, LinkEd, DefaultEd );
SetSlotOption( ItemEdit:ChoiceNames, NO_AUTO_ASK, TRUE );
SetValue( ItemEdit:ChoiceNames, &Edit, &Rename, &Delete, &Move, -, &Actions, &Links, &Default
);
MakeSlot( ItemEdit:Check );
SetSlotOption( ItemEdit:Check, INHERIT, FALSE );
SetSlotOption( ItemEdit:Check, NO_AUTO_ASK, TRUE );
SetSlotOption( ItemEdit:Check, VALUE_TYPE, BOOLEAN );
ItemEdit:Check <- FALSE;
SetSlotOption( ItemEdit:Check, IF_NEEDED, NULL );
SetSlotOption( ItemEdit:Check, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit:Check, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit:Check, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit:name );
MakeSlot( ItemEdit:TempList );
SetSlotOption( ItemEdit:TempList, INHERIT, FALSE );
SetSlotOption( ItemEdit:TempList, MULTIPLE
SetValue( ItemEdit:TempList, PUP18Off );
SetSlotOption( ItemEdit:TempList, IF_NEEDED, NULL );
SetSlotOption( ItemEdit:TempList, WH      CCESS, NULL );
SetSlotOption( ItemEdit:TempList, BEF     CHANGE, NULL );
SetSlotOption( ItemEdit:TempList, AFTER_CHANGE, NULL );

    /*************************************
     **** INSTANCE: HalEdit
     *************************************/
MakeInstance( HalEdit, Menu );

/************** METHOD: Run  ***************/
MakeMethod( HalEdit, Run, [],
  {
  If ( Null?( Global:RunState ) Or Not      RunState #= Running ) )
    Then Start( );
  } );

/************** METHOD: Normal  **************/
MakeMethod( HalEdit, Normal, [],
  {
  HideWindow( BROWSER );
  If Function?( ShowInstanceMenu )
    Then DeleteFunction( ShowInstanceMenu );
  If Function?( ShowClassMenu )
    Then DeleteFunction( ShowClassMenu );
  ShowWindow( BROWSER );
  SetBrowserFocus( Root );
  SetMenuBar( SESSION );
  SetMenuBar( BROWSER );
  ShowWindow( KTOOLS );
  } );

/************** METHOD: InitMenu  **************/
MakeMethod( HalEdit, InitMenu, [owner ],
```

```
    0 );

/************** METHOD: Stop  **************/
MakeMethod( HalEdit, Stop, [],
    {
    Global:Continue = PostMenu( "Continue? ", Yes, No );
    If ( Global:Continue #= No )
     Then {
          DisplayText( HalEx, FormatValue( "\n  Ending.... \n" ) );
          If Member?( HalEdit:Disabled, Run )
            Then RemoveFromList( HalEdit:Disabled, Run );
          };
    Global:RunState = Running;
    SetMenuBar( SESSION, HalEdit );
    } );

/************** METHOD: Interface  **************/
MakeMethod( HalEdit, Interface, [],
    {
    SetMenuBar( Session1, InterfaceMenu );
    ShowWindow( Session1 );
    } );

/************** METHOD: Node  **************/
MakeMethod( HalEdit, Node, [],
    {
    HideWindow( BROWSER );
    PositionWindow( BROWSER, 70, 70, 500, 300 );
    SetBrowserFocus( Nodes );
    ShowWindow( BROWSER );
    } );

/************** METHOD: Action  **************/
MakeMethod( HalEdit, Action, [],
    {
    MaximizeWindow( Session3 );
    ShowWindow( Session3 );
    } );
SetSlotOption( HalEdit:Choices, NO_AUTO_ASK, TRUE );
SetValue( HalEdit:Choices, Files, Run, Stop, HalEdit1, Explain, Interface, Normal, Node, Action );
SetSlotOption( HalEdit:ChoiceNames, NO_AUTO_ASK, TRUE );
SetValue( HalEdit:ChoiceNames, &File, &Run, &Stop, &Edit, &Explain, &Interface, N&ormal,
&Nodes, &Action, Win );
SetValue( HalEdit:Disabled, Run );

    /*************************************
     **** INSTANCE: HalEdit1
     *************************************/
MakeInstance( HalEdit1, Menu );

/************** METHOD: Frame  **************/
MakeMethod( HalEdit1, Frame, [],
```

```
    {
    Global.Change = TRUE;
    KillTimer( 2 );
    HideWindow( BROWSER );
    If Function?( ShowInstanceMenu )
      Then DeleteFunction( ShowInstanceMenu );
    MakeFunction( ShowInstanceMenu, [ obj ], PopupMenu( ItemEdit,
                                        obj, BROWSER ) );
    ShowInstanceMenu;
    If Function?( ShowClassMenu )
      Then DeleteFunction( ShowClassMenu );
    MakeFunction( ShowClassMenu, [ obj ], PopupMenu( ItemEdit3,
                                      obj, BROWSER ) );
    ShowClassMenu;
    PositionWindow( BROWSER, 70, 70, 500, 300 );
    SetBrowserFocus( Items );
    SetWindowTitle( BROWSER, "Item Browser" );
    ShowWindow( BROWSER );
    } );

/************** METHOD: InitMenu  **************/
MakeMethod( HalEdit1, InitMenu, [owner ],
   {} );

/************** METHOD: PrepareMenu  **************/
MakeMethod( HalEdit1, PrepareMenu, [],
   {} );
SetValue( HalEdit1:Choices, &Name, RuleEdit );
SetSlotOption( HalEdit1:ChoiceNames, NO_AUTO_ASK, TRUE );
SetValue( HalEdit1:ChoiceNames, &Frame, R&ule );

     /*************************************
      ****  INSTANCE: ItemEdit2
      *************************************/
MakeInstance( ItemEdit2, Menu );

/************** METHOD: Input  **************/
MakeMethod( ItemEdit2, Input, [owner window choice ],
   {
   Self:Check = TRUE;
   While  (Self:Check)
       {
       PostInputForm( " Edit " # owner # " Inputs ", owner:Inputs,
          Inputs );
       Self:Check = FALSE;
       EnumList( owner:Inputs, I,
          {
          If Not( Instance?( I ) And Is.KindOf?( I, Items ) )
            Then Self:Check = TRUE;
          } );
       If ( LengthList( owner:Inputs ) == 0 )
         Then Self:Check = FALSE;
```

```
        If Self:Check
          Then PostMessage( "ERROR - Input Item does not exist " );
        };
    } );

/************** METHOD: Output  **************/
MakeMethod( ItemEdit2, Output, [owner window choice ],
    {
    Self:Check = TRUE;
    While (Self:Check)
        {
        PostInputForm( " Edit " # owner # " Outputs", owner, Outputs,
            Outputs );
        Self:Check = FALSE;
        EnumList( owner:Inputs, I,
            {
            If Not( Instance?( I ) And IsAKindOf?( I, Items ) )
              Then Self:Check = TRUE;
            } );
        If ( LengthList( owner:Outputs ) == 0 )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR - Output Item does not exist " );
        };
    } );

/************** METHOD: AddVar  **************/
MakeMethod( ItemEdit2, AddVar, [owner window choice ],
    {
    Self:Check = TRUE;
    While (Self:Check)
        {
        Self:Check = FALSE;
        Self:name = NULL;
        PostInputForm( " Add Variable - " # owner, Self:name, "Variable Name" );
        EnumList( Self:VarList, V,
            {
            If ( Not( Null?( Self:name ) ) And ( V #= Self:name ) )
              Then {
                  Self:Check = TRUE;
                  Self:Error = 1;
                  };
            } );
        If Member?( owner:ValidSlots,
              {
              Self:name;
              } )
          Then {
              Self:Check = TRUE;
              Self:Error = 2;
              };
        If ( Not( Null?( Self:name ) ) And Self:Check #= FALSE )
```

```
Then {
    If Not( Class?( Self:name ) )
     Then {
        MakeClass( Self:name, Variables );
        Global:VarNo += 1;
        SetValue( {
                Self:name;
                }, Number, Global:VarNo );
        If Not( Member?( Items:ValidVars, Self:name ) )
         Then AppendToList( Items:ValidVars, Self:name );
        };
    If IsAKindOf?( Self:name, Variables )
     Then {
        If Not( Instance?( owner # _ # Self:name ) )
         Then MakeInstance( owner # _ # Self:name,
                Self:name );
        If IsAKindOf?( owner # _ # Self:name, Self:name )
         Then {
            MakeSlot( owner, Self:name );
            SetSlotOption( owner, Self:name,
               VALUE_TYPE, NUMBER );
            SetSlotOption( owner, Self:name,
               WHEN_ACCESS, GetVar );
            MakeSlot( owner, N # Self:name );
            SetValue( owner, N # Self:name,
               owner # _ # Self:name );
            SetValue( owner # _ # Self:name,
               Owner, owner );
            SetValue( GetValue( owner, N #
                                    {
                                    Self:name;
                                    } ),
               Type, Analog );
            SendMessage( Self, EditVar, owner,
               window, Self:name );
            }
         Else {
            Self:Error = 2;
            Self:Check = TRUE;
            };
        };
    }
 Else {
    Self:Check = TRUE;
    Self:Error = 2;
    };
If Null?( Self:name )
 Then Self:Check = FALSE;
If Self:Check
 Then {
    If ( Self:Error == 1 )
     Then PostMessage( "ERROR - Variable already exists " );
```

```
            If ( Self:Error == 2 )
              Then PostMessage( "ERROR - Reserved name" );
            };
        };
    } );

/************** METHOD: EditVar  **************/
MakeMethod( ItemEdit2, EditVar, [owner window choice ],
    {
    Self:NVar = GetValue( owner, N # choice );
    If ( Not( Null?( Self:NVar ) ) And ( Self:NVar #= owner #
                                        _ # choice ) )
      Then {
          SendMessage( ItemVars, Init, owner, choice, Self:NVar );
          }
      Else {
          If ( PostMenu( " Make Local?", YES, NO ) #= YES )
            Then {
                If Not( Instance?( owner # _ # choice ) )
                  Then {
                      MakeInstance( owner # _ # choice, choice );
                      };
                MakeSlot( owner, choice );
                SetSlotOption( owner, choice, VALUE_TYPE, NUMBER );
                SetSlotOption( owner, choice, WHEN_ACCESS, GetVar );
                MakeSlot( owner # choice );
                If Not( Member?( Items:ValidVars, Self:name ) )
                  Then AppendToList( Items:ValidVars, choice );
                SetSlotOption( owner, choice, VALUE_TYPE, NUMBER );
                SetValue( owner # _ # choice, Owner, owner );
                SetValue( owner, N # choice, owner # _ # choice );
                SetValue( GetValue( owner,
                          {
                          N # choice;
                          } ), Type, Analog );
                PostMessage( "  Making local" );
                SendMessage( Self, EditVar, owner, window, choice );
                };
          };
    } );

/************** METHOD: DisplayCall  ******** ******/
MakeMethod( ItemEdit2, DisplayCall, [owner window choice ],
    {
    PostMessage( FormatValue( "Menu:     %s
Owner:  %s
Choice:  %s", Self,
                  owner, choice ) );
    choice;
    } );

/************** METHOD: InitMenu  **************/
```

```
MakeMethod( ItemEdit2, InitMenu, [owner ],
   {
   ClearList( Self:ChoiceNames );
   ClearList( Self:Choices );
   AppendToList( Self:ChoiceNames, Self:TempChoiceNames );
   AppendToList( Self:Choices, Self:TempChoices );
   ClearList( Self:VarList );
   GetSlotList( owner, owner:Value_List );
   EnumList( owner:Value_List, Value,
      {
      If Member?( owner:ValidVars, Value )
       Then {
            AppendToList( Self:VarList, Value );
            AppendToList( Self:ChoiceNames, Value );
            AppendToList( Self:Choices, Value );
            };
      } );
   } );

/************ METHOD: Vars  ************/
MakeMethod( ItemEdit2, Vars, [owner window choice ],
   {
   Self:Function = PopupMenu( Choice );
   If ( Self:Function #= Edit )
    Then SendMessage( Self, EditVar, owner, window, choice );
   If ( Self:Function #= Delete )
    Then SendMessage( Self, DeleteVar, owner, window, choice );
   If ( Self:Function #= Rename )
    Then SendMessage( Self, RenameVar, owner, window, choice );
   } );

/************ METHOD: RenameVar  ************/
MakeMethod( ItemEdit2, RenameVar, [owner window choice ],

   Self:Check = TRUE;
   If ( GetOwner( owner, choice ) #= owner )
    Then {
        While  (Self:Check)
            {
            Self:name = choice;
            PostInputForm( " Rename Variable " # owner, Self:name,
               "New name" );
            If ( Not( Slot?( owner, Self:name ) )
                  And Not( Null?( Self:name ) ) )
             Then {
                 If Not( Class?( Self:name ) )
                  Then {
                      MakeClass( Self:name, Variables );
                      If Not( Member?( Items:ValidVars,
                              Self:name ) )
                       Then AppendToList( Items:ValidVars,
                              Self:name );
```

```
                    };
                RenameSlot( owner, choice, Self:name );
                RenameSlot( owner, N # choice,
                   N # Self:name );
                MoveInstance( GetValue( owner, N # Self:name ),
                   Self:name );
                Self:Check = FALSE;
                If ( CountInstances( choice )
                      == 0 )
                 Then {
                    DeleteClass( choice );
                    RemoveFromList( Items:ValidVars,
                       choice );
                    };
                SetValue( owner, N # Self:name,
                   RenameInstance( GetValue( owner, N
                                             #
                                             Self:name ),
                      wner # _ # Self:name ) );
                };
            If ( Null?( Self:name ) Or Self:name #= owner )
             Then Self:Check = FALSE;
            If Self:Check
             Then PostMessage( "ERROR - Variable already exists" );
            };
        }
      Else {
        Self:Check = FALSE;
        PostMessage( "ERROR - Variable is inherited" );
        };
  } );

/************** METHOD: DeleteVar  **************/
MakeMethod( ItemEdit2, DeleteVar, [owner window choice ],
  {
  If ( Self:NVar #= owner # _ # choice )
   Then {
      Self:name = PostMenu( "
Delete variable " # choice #
                          " ?", YES, NO );
      If ( Self:name #= YES )
       Then {
          DeleteInstance( GetValue( owner, N # choice ) );
          DeleteSlot( owner, choice );
          DeleteSlot( owner, N # choice );
          If ( CountInstances( choice ) == 0 )
           Then {
              DeleteClass( choice );
              RemoveFromList( Items:ValidVars, choice );
              };
          };
      }
```

```
    Else PostMessage( choice # " Not Local" );
  } );

/************** METHOD: ItemNum  **************/
MakeMethod( ItemEdit2, ItemNum, [owner window choice ],
   {
   PostInputForm( " Item Number for " # owner, owner, ItemNo, "Number:" );
   } );
ItemEdit2:Y = 160;
SetValue( ItemEdit2:Choices, Input, Output, -, AddVar, -, EditDevice, ItemNum, -, Level );
SetValue( ItemEdit2:ChoiceNames, "Edit &Inputs", "Edit &Outputs", -, "Add &Variable", -, "&Edit
Device", "&Item Number", -, Level );
MakeSlot( ItemEdit2:Check );
SetSlotOption( ItemEdit2:Check, INHERIT, FALSE );
SetSlotOption( ItemEdit2:Check, VALUE_TYPE, BOOLEAN );
ItemEdit2:Check = FALSE;
SetSlotOption( ItemEdit2:Check, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:Check, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:Check, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit2:Check, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:VarList );
SetSlotOption( ItemEdit2:VarList, INHERIT, FALSE );
SetSlotOption( ItemEdit2:VarList, NO_AUTO_ASK, TRUE );
SetSlotOption( ItemEdit2:VarList, MULTIPLE );
SetValue( ItemEdit2:VarList, Level );
SetSlotOption( ItemEdit2:VarList, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:VarList, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:VarList, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit2:VarList, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:TempChoiceNames );
SetSlotOption( ItemEdit2:TempChoiceNames, INHERIT, FALSE );
SetSlotOption( ItemEdit2:TempChoiceNames, MULTIPLE );
SetValue( ItemEdit2:TempChoiceNames, "Edit &Inputs", "Edit &Outputs", -, "Add &Variable", -,
"&Edit Device", "&Item Number", - );
SetSlotOption( ItemEdit2:TempChoiceNames, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:TempChoiceNames, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:TempChoiceNames, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit2:TempChoiceNames, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:TempChoices );
SetSlotOption( ItemEdit2:TempChoices, INHERIT, FALSE );
SetSlotOption( ItemEdit2:TempChoices, MULTIPLE );
SetValue( ItemEdit2:TempChoices, Input, Output, -, AddVar, -, EditDevice, ItemNum, - );
SetSlotOption( ItemEdit2:TempChoices, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:TempChoices, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:TempChoices, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit2:TempChoices, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:NVar );
SetSlotOption( ItemEdit2:NVar, INHERIT, FALSE );
ItemEdit2:NVar = TKP01_Level;
SetSlotOption( ItemEdit2:NVar, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:NVar, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:NVar, BEFORE_CHANGE, NULL );
```

```
SetSlotOption( ItemEdit2:NVar, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:name );
ItemEdit2:name = Level;
MakeSlot( ItemEdit2:Error );
ItemEdit2:Error = 2;
MakeSlot( ItemEdit2:Max );
SetSlotOption( ItemEdit2:Max, INHERIT, FALSE );
SetSlotOption( ItemEdit2:Max, VALUE_TYPE, NUMBER );
ItemEdit2:Max = 100;
SetSlotOption( ItemEdit2:Max, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:Max, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:Max, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit2:Max, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:Min );
SetSlotOption( ItemEdit2:Min, INHERIT, FALSE );
SetSlotOption( ItemEdit2:Min, VALUE_TYPE, NUMBER );
ItemEdit2:Min = 0;
SetSlotOption( ItemEdit2:Min, IF_NEEDED, NULL );
SetSlotOption( ItemEdit2:Min, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit2:Min, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit2:Min, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit2:Function );
ItemEdit2:Function = Rename;
ItemEdit2:DefaultMethod = Vars;
ItemEdit2:X = 43;

    /*************************************
     **** INSTANCE: ItemEdit3
     *************************************/
MakeInstance( ItemEdit3, Menu );

/************ METHOD: Instance  ************/
MakeMethod( ItemEdit3, Instance, [owner window choice ],
  {
  Self:Check = TRUE;
  While (Self:Check)
      {
      Self:Instance = NULL;
      PostInputForm( " New Instance - " # owner, Self:Instance,
         "Instance name" );
      If ( Not( Null?( Self:Instance ) ) And ( Not( Instance?( Self:Instance ) )
                                          And MakeInstance( Self:Instance,
                                              owner ) ) )
       Then {
          Self:Check = FALSE;
          Global:ItemNo += 1;
          SetValue( {
                  Self:Instance;
                  }, ItemNo, Global:ItemNo );
          };
      If Null?( Self:Instance )
       Then Self:Check = FALSE;
```

```
        If Self:Check
          Then PostMessage( "ERROR - Instance already exists" );
        };
    } );

/************** METHOD: SubClass  **************/
MakeMethod( ItemEdit3, SubClass, [owner window choice ],
    {
    Self:Check = TRUE;
    While (Self:Check)
        {
        Self:SubClass = NULL;
        PostInputForm( " New SubClass " # owner, Self:SubClass,
          "SubClass name" );
        If ( Not( Null?( Self:SubClass ) ) And ( Not( Class?( Self:SubClass ) )
                                          And MakeClass( Self:SubClass,
                                              owner ) ) )
          Then Self:Check = FALSE;
        If Null?( Self:SubClass )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR - Class already exists" );
        };
    } );

/************** METHOD: AddVar  **************/
MakeMethod( ItemEdit3, AddVar, [owner ],
    {
    Self:Check = TRUE;
    While (Self:Check)
        {
        Self:Check = FALSE;
        Self:name = NULL;
        PostInputForm( " Add Variable - " # owner, Self:name, "Variable Name" );
        EnumList( Self:VarList, V,
          {
          If ( Not( Null?( Self:name ) ) And ( V #= Self:name ) )
            Then {
                Self:Check = TRUE;
                Self:Error = 1;
                };
          } );
        If Member?( owner, ValidSlots,
              {
              Self:name;
              } )
          Then {
              Self:Check = TRUE;
              Self:Error = 2;
              };
        If ( Not( Null?( Self:name ) ) And Self:Check #= FALSE )
          Then {
```

```
        If Not( Class?( Self:name ) )
         Then {
             MakeClass( Self:name, Variables );
             Global:VarNo += 1;
             SetValue( {
                     Self:name;
                     }, Number, Global:VarNo );
             If Not( Member?( Items:ValidVars, Self:name ) )
              Then AppendToList( Items:ValidVars, Self:name );
             };
        If IsAKindOf?( Self:name, Variables )
         Then {
             If Not( Instance?( owner # _ # Self:name ) )
              Then {
                  MakeInstance( owner # _ # Self:name,
                     Self:name );
                  };
             If IsAKindOf?( owner # _ # Self:name, Self:name )
              Then {
                  MakeSlot( owner, Self:name );
                  SetSlotOption( owner, Self:name,
                     VALUE_TYPE, NUMBER );
                  SetSlotOption( owner, Self:name,
                     WHEN_ACCESS, GetVar );
                  MakeSlot( owner, N # Self:name );
                  SetValue( owner, N # Self:name,
                     owner # _ # Self:name );
                  SetValue( owner # _ # Self:name,
                     Owner, owner );
                  SetValue( GetValue( owner, N #
                                          {
                                          Self:name;
                                          } ),
                     Type, Analog );
                  SendMessage( Self, EditVar, owner,
                     window, Self:name );
                  }
              Else {
                  Self:Check = TRUE;
                  Self:Error = 2;
                  };
             }
         Else {
             Self:Check = TRUE;
             Self:Error = 2;
             };
        };
    If Null?( Self:name )
     Then Self:Check = FALSE;
    If Self:Check
     Then {
        If ( Self:Error == 1 )
```

```
            Then PostMessage( "ERROR - Variable already exists " );
          If ( Self:Error == 2 )
            Then PostMessage( "ERROR - Reserved name" );
          };
      };
  } );

/************** METHOD: Delete  **************/
MakeMethod( ItemEdit3, Delete, [owner window choice ],
  {
  If ( CountInstances( owner ) == 0 )
    Then {
      If ( PostMenu( " Delete " # owner # " ?", YES, NO )
          #= YES )
        Then {
          EnumList( owner:ValidVars, VD,
            {
            If Slot?( owner, N # VD )
              Then {
                If ( Not( Null?( GetValue( owner,
                                  N # VD ) ) )
                      And ( GetOwner( owner, N
                                      #
                                      VD )
                            #= owner ) )
                  Then {
                    If Instance?( owner # _ #
                                  VD )
                      Then DeleteInstance( owner
                                           #
                                           _
                                           #
                                           VD );
                    If ( CountInstances( VD )
                         == 0 )
                      Then {
                        DeleteClass( VD );
                        RemoveFromList( Items:ValidVars,
                          VD );
                        };
                    };
                };
            } );
          DeleteClass( owner );
          };
      }
    Else PostMessage( "ERROR - Erase instances of class first" );
  } );

/************** METHOD: Rename  **************/
MakeMethod( ItemEdit3, Rename, [owner window choice ],
  {
```

```
    Self:Check = TRUE;
    While (Self:Check)
        {
        Self:name = owner;
        PostInputForm( " Rename Class " # owner, Self:name, "New name" );
        If ( Not( Class?( Self:name ) ) And Not( Null?( Self:name ) )
                And RenameClass( owner, Self:name ) )
          Then Self:Check = FALSE;
        If ( Null?( Self:name ) Or Self:name #= owner )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR -  Instance already exists" );
        };
    };

/************** METHOD: InitMenu  ***************/
MakeMethod( ItemEdit3, InitMenu, [owner ],
    {

    If ( owner #= Items )
      Then AppendToList( Self:Disabled, Delete, Rename, Move, Edit )
      Else ClearList( Self:Disabled );
    ClearList( Self:ChoiceNames );
    ClearList( Self:Choices );
    AppendToList( Self:ChoiceNames, Self:TempChoiceNames );
    AppendToList( Self:Choices, Self:TempChoices );
    ClearList( Self:VarList );
    GetSlotList( owner, owner:Value_List );
    ForAllList( owner:Value_List, Value,
      {
      If Member?( owner:ValidVars, Value )
        Then {
            AppendToList( Self:VarList, Value );
            AppendToList( Self:ChoiceNames, Value );
            AppendToList( Self:Choices, Value );
            };
      } );
    } );

/************** METHOD: EditVar  ***************/
MakeMethod( ItemEdit3, EditVar, [owner window choice ],
    {
    Self:NVar = GetValue( owner, N # choice );
    If ( Self:NVar #= owner # _ # choice )
      Then {
          SendMessage( ItemVars, Init, owner, choice, Self:NVar );
          }
      Else {
          If ( PostMenu( " Make Local?", YES, NO ) #= YES )
            Then {
                If Not( Instance?( owner # _ # choice ) )
                  Then {
                      MakeInstance( owner # _ # choice, choice );
```

```
                    };
                MakeSlot( owner, choice );
                SetSlotOption( owner, choice, VALUE_TYPE, NUMBER );
                SetSlotOption( owner, choice, WHEN_ACCESS, GetVar );
                MakeSlot( owner, N # choice );
                If Not( Member?( Items:ValidVars, Self:name ) )
                  Then AppendToList( Items:ValidVars, choice );
                SetSlotOption( owner, choice, VALUE_TYPE, NUMBER );
                SetValue( owner, N # choice, owner # _ # choice );
                PostMessage( "  Making local" );
                SetValue( GetValue( owner, N # (
                                        choice;
                                        ) ), Type, Analog );
                SendMessage( Self, EditVar, owner, window, choice );
                };
            };
    } );

/************** METHOD: Vars  **************/
MakeMethod( ItemEdit3, Vars, [owner window choice ],
    {
    Self:Function = PopupMenu( Choice );
    If ( Self:Function #= Edit )
      Then SendMessage( Self, EditVar, owner, window, choice );
    If ( Self:Function #= Delete )
      Then SendMessage( Self, DeleteVar, owner, window, choice );
    If ( Self:Function #= Rename )
      Then SendMessage( Self, RenameVar, owner, window, choice );
    } );

/************** METHOD: RenameVar  **************/
MakeMethod( ItemEdit3, RenameVar, [owner window choice ],
    {
    Self:Check = TRUE;
    If ( GetOwner( owner, choice ) #= owner )
      Then {
          While  (Self:Check)
              {
              Self:name = choice;
              PostInputForm( " Rename Variable " # owner, Self:name,
                 "New name" );
              If ( Not( Slot?( owner, Self:name ) )
                     And Not( Null?( Self:name ) ) )
               Then {
                   If Not( Class?( Self:name ) )
                    Then {
                        MakeClass( Self:name, Variables );
                        If Not( Member?( Items:ValidVars,
                                Self:name ) )
                          Then AppendToList( Items:ValidVars,
                                Self:name );
                        };
```

```
                    RenameSlot( owner, choice, Self:name );
                    RenameS          ner, N # choice,
                       N # S.          ) );
                    MoveInstance( GetValue( owner, N # Self:name ),
                       Self:name );
                    Self:Check = FALSE;
                    If ( CountInstances( choice )
                          == 0 )
                     Then {
                        DeleteClass( choice );
                        RemoveFromList( Items:ValidVars,
                           choice );
                        };
                    Set  lue( owner, N # Self:name,
                       .enameInstance( GetValue( owner, N
                                                  #
                                                  Self:name ),
                           owner # _ # Self:name ) );
                    };
                If ( Null?( Self:name ) Or Self:name #= owner )
                 Then Self:Check = FALSE;
                If Self:Check
                 Then PostMessage( "ERROR -  Variable already exists" );
                };
            }
         Else {
            Self:Check = FALSE;
            PostMessage( "ERROR - Variable is inherited" );
            };
      } );

/************** METHOD: DeleteVar  **************/
MakeMethod( ItemEdit3, DeleteVar, [owner window choice ],
   {
   If ( Self:NVar #= owner # _ # choice )
    Then {
        Self:name = PostMenu( "
Delete variable " # choice #
                              " ?", YES, NO );
        If ( Self:name #= YES )
         Then {
            DeleteInstance( GetValue( owner, N # choice ) );
            DeleteSlot( owner, choice );
            DeleteSlot( owner, N # choice );
            If ( CountInstances( choice ) == 0 )
             Then {
                DeleteClass( choice );
                RemoveFromList( Items:ValidVars, choice );
                };
            };
        }
    Else PostMessage( choice # " Not Local !" );
```

```
  } ),
ItemEdit3:X = 166;
ItemEdit3:Y = 128;
SetValue( ItemEdit3:Choices, Instance, SubClass, -, AddVar, EditDevice, -, Rename, Delete, - );
SetSlotOption( ItemEdit3:ChoiceNames, NO_AUTO_ASK, FALSE );
SetValue( ItemEdit3:ChoiceNames, "Add &Instance", "Add &Subclass", -, "Add &Variable", "Edit
&Device", -, &Rename, &Delete, - );
ClearList( ItemEdit3:Disabled );
ItemEdit3:DefaultMethod = Vars;
MakeSlot( ItemEdit3:Check );
SetSlotOption( ItemEdit3:Check, INHERIT, FALSE );
SetSlotOption( ItemEdit3:Check, NO_AUTO_ASK, TRUE );
SetSlotOption( ItemEdit3:Check, VALUE_TYPE, BOOLEAN );
ItemEdit3:Check = FALSE;
SetSlotOption( ItemEdit3:Check, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:Check, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:Check, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:Check, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:Instance );
SetSlotOption( ItemEdit3:Instance, INHERIT, FALSE );
ItemEdit3:Instance = P02;
SetSlotOption( ItemEdit3:Instance, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:Instance, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:Instance, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:Instance, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:SubClass );
ItemEdit3:SubClass = Stream;
MakeSlot( ItemEdit3:name );
ItemEdit3:name = Leach;
MakeSlot( ItemEdit3:VarList );
SetSlotOption( ItemEdit3:VarList, INHERIT, FALSE );
SetSlotOption( ItemEdit3:VarList, MULTIPLE );
ClearList( ItemEdit3:VarList );
SetSlotOption( ItemEdit3:VarList, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:VarList, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:VarList, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:VarList, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:Error );
ItemEdit3:Error = 1;
MakeSlot( ItemEdit3:TempChoices );
SetSlotOption( ItemEdit3:TempChoices, INHERIT, FALSE );
SetSlotOption( ItemEdit3:TempChoices, MULTIPLE );
SetValue( ItemEdit3:TempChoices, Instance, SubClass, -, AddVar, EditDevice, -, Rename, Delete, - );
SetSlotOption( ItemEdit3:TempChoices, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:TempChoices, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:TempChoices, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:TempChoices, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:TempChoiceNames );
SetSlotOption( ItemEdit3:TempChoiceNames, INHERIT, FALSE );
SetSlotOption( ItemEdit3:TempChoiceNames, MULTIPLE );
SetValue( ItemEdit3:TempChoiceNames, "Add &Instance", "Add &Subclass", -, "Add &Variable",
"Edit &Device", -, &Rename, &Delete, - );
```

```
SetSlotOption( ItemEdit3:TempChoiceNames, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:TempChoiceNames, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:TempChoiceNames, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:TempChoiceNames, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:NVar );
ItemEdit3:NVar = HalLeach_Level;
MakeSlot( ItemEdit3:Min );
SetSlotOption( ItemEdit3:Min, INHERIT, FALSE );
SetSlotOption( ItemEdit3:Min, VALUE_TYPE, NUMBER );
ItemEdit3:Min = 0;
SetSlotOption( ItemEdit3:Min, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:Min, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:Min, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:Min, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:Max );
SetSlotOption( ItemEdit3:Max, INHERIT, FALSE );
SetSlotOption( ItemEdit3:Max, VALUE_TYPE, NUMBER );
ItemEdit3:Max = 100;
SetSlotOption( ItemEdit3:Max, IF_NEEDED, NULL );
SetSlotOption( ItemEdit3:Max, WHEN_ACCESS, NULL );
SetSlotOption( ItemEdit3:Max, BEFORE_CHANGE, NULL );
SetSlotOption( ItemEdit3:Max, AFTER_CHANGE, NULL );
MakeSlot( ItemEdit3:Function );
ItemEdit3:Function = Edit;

    /*****************************************
     **** INSTANCE: ActionEdit
     *****************************************/
MakeInstance( ActionEdit, Menu );

/************** METHOD: Edit  **************/
MakeMethod( ActionEdit, Edit, [owner window choice ],
  {
  owner:Item = Global:Item;
  PostInputForm( "Edit Action " # owner, owner, Variable, "Variable:",
    owner, ActionMesg, " Action Message:", owner, Tag, "Tag Name:",
    owner, ActionNo, "Action Number", owner, AcVariation, "Variation:",
    owner, Mode, "Variation type", owner, TimeLSP, "Time Lag(sec):",
    owner, InitCost, "Initial Cost:", owner, Delay, "Repeat Delay(sec):",
    owner, ActionType, "MANUAL/AUTO" );
  PostInputForm( " Edit Action " # owner # " continued", owner, VariableType,
    "Analog/Discrete", owner, Failures, Failures, owner, Successes,
    "Successes:" );
  } );

/************** METHOD: Delete  **************/
MakeMethod( ActionEdit, Delete, [owner window choice ],
  {
  If ( PostMenu( "Delete " # owner # " ?", YES, NO ) #= YES )
   Then DeleteInstance( owner );
  } );
```

```
/************** METHOD: Rename  **************/
MakeMethod( ActionEdit, Rename, [owner window choice ],
   {
   Self:Check = TRUE;
   While (Self:Check)

      Self:name = owner;
      PostInputForm( " Rename Instance " # owner, Self:name,
         "New name" );
      If ( Not( Instance?( Self:name ) ) And Not( Null?( Self:name ) )
            And RenameInstance( owner, Self:name ) )
        Then Self:Check = FALSE;
      If ( Null?( Self:name ) Or Self:name #= owner )
        Then Self:Check = FALSE;
      If Self:Check
        Then PostMessage( "ERROR -  Instance already exists" );
      };
   } );

/************** METHOD: Move  **************/
MakeMethod( ActionEdit, Move, [owner window choice ],
   {
   Self:Check = TRUE;
   While (Self:Check)
      {
      PostInputForm( "Move " # owner, Self:name, "New parent" );
      If ( Class?( Self:name ) And Not( Null?( Self:name ) )
            And MoveInstance( owner, Self:name ) )
        Then Self:Check = FALSE;
      If ( Null?( Self:name ) Or Self:name #= GetParent( owner ) )
        Then Self:Check = FALSE;
      If Self:Check
        Then PostMessage( "ERROR -  Class does not exists" );
      };
   } );
ActionEdit:X = 160;
ActionEdit:Y = 158;
SetValue( ActionEdit:Choices, Edit, Delete, Rename );
SetValue( ActionEdit:ChoiceNames, &Edit, &Delete, &Rename );
MakeSlot( ActionEdit:Check );
SetSlotOption( ActionEdit:Check, INHERIT, FALSE );
SetSlotOption( ActionEdit:Check, VALUE_TYPE, BOOLEAN );
ActionEdit:Check = FALSE;
SetSlotOption( ActionEdit:Check, IF_NEEDED, NULL );
SetSlotOption( ActionEdit:Check, WHEN_ACCESS, NULL );
SetSlotOption( ActionEdit:Check, BEFORE_CHANGE, NULL );
SetSlotOption( ActionEdit:Check, AFTER_CHANGE, NULL );
MakeSlot( ActionEdit:name );
ActionEdit:name = Unchoke;

   /*************************************
    **** INSTANCE: ActionEdit2
```

```
/*********** METHOD: Instance ***************/
MakeMethod( ActionEdit2, Instance, [owner window choice ],
  {
  Self:Check = TRUE;
  While (Self:Check)
    {
    Self:Instance = NULL;
    PostInputForm( " New Instance " # owner, Self:Instance,
      "Instance name" );
    If Not( Null?( Self:Instance ) )
     Then {
       If ( Not( Instance?( Self:Instance ) )
            And MakeInstance( Self:Instance, owner ) )
         Then ( Self:Check = FALSE )
         Else PostMessage( Self:Instance # " already exists!" );
       }
     Else Self:Check = FALSE;
    };
  } );

/*********** METHOD: SubClass ***************/
MakeMethod( ActionEdit2, SubClass, [owner window choice ],
  {
  Self:Check = TRUE;
  While (Self:Check)
    {
    PostInputForm( " New SubClass " # owner, Self:SubClass,
      "SubClass name" );
    If ( Not( Class?( Self:SubClass ) ) And MakeClass( Self:SubClass,
                                        owner ) )
     Then Self:Check = FALSE;
    If Null?( Self:SubClass )
     Then Self:Check = FALSE;
    };
  } );

/*********** METHOD: Delete ***************/
MakeMethod( ActionEdit2, Delete, [owner window choice ],
  {
  If ( CountInstances( owner ) == 0 )
   Then {
     If ( PostMenu( " Delete " # owner # " ?", YES, NO )
          #= YES )
      Then DeleteClass( owner );
     }
   Else PostMessage( "ERROR - Erase instances of class first" );
  } );

/*********** METHOD: Rename ***************/
```

```
MakeMethod( ActionEdit2, Rename, [owner window choice ],
   {
   Self:Check = TRUE;
   While  (Self:Check)
        {
        Self:name = owner;
        PostInputForm( " Rename Class " # owner, Self:name, "New Class name" );
        If ( Not( Class?( Self:name ) ) And Not( Null?( Self:name ) )
              And RenameClass( owner, Self:name ) )
          Then Self:Check = FALSE;
        If ( Null?( Self:name ) Or Self:name #= owner )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR -  Instance already exists" );
        };
   } );

/************** METHOD: InitMenu  **************/
MakeMethod( ActionEdit2, InitMenu, [owner ],
   {
   If ( owner #= Actions )
     Then AppendToList( Self:Disabled, Rename, Delete, Edit, Move )
     Else ClearList( Self:Disabled );
   } ),
ActionEdit2:X = 59;
ActionEdit2:Y = 125;
SetValue( ActionEdit2:Choices, Instance );
SetSlotOption( ActionEdit2:ChoiceNames, NO_AUTO_ASK, FALSE );
SetValue( ActionEdit2:ChoiceNames, "Add &Instance" );
ClearList( ActionEdit2:Disabled );
MakeSlot( ActionEdit2:Check );
SetSlotOption( ActionEdit2:Check, INHERIT, FALSE );
SetSlotOption( ActionEdit2:Check, NO_AUTO_ASK, TRUE );
SetSlotOption( ActionEdit2:Check, VALUE_TYPE, BOOLEAN );
ActionEdit2:Check = FALSE;
SetSlotOption( ActionEdit2:Check, IF_NEEDED, NULL );
SetSlotOption( ActionEdit2:Check, WHEN_ACCESS, NULL );
SetSlotOption( ActionEdit2:Check, BEFORE_CHANGE, NULL );
SetSlotOption( ActionEdit2:Check, AFTER_CHANGE, NULL );
MakeSlot( ActionEdit2:Instance );
SetSlotOption( ActionEdit2:Instance, INHERIT, FALSE );
ActionEdit2:Instance = P02_D01;
SetSlotOption( ActionEdit2:Instance, IF_NEEDED, NULL );
SetSlotOption( ActionEdit2:Instance, WHEN_ACCESS, NULL );
SetSlotOption( ActionEdit2:Instance, BEFORE_CHANGE, NULL );
SetSlotOption( ActionEdit2:Instance, AFTER_CHANGE, NULL );
MakeSlot( ActionEdit2:SubClass );
ActionEdit2:SubClass = Hello;
MakeSlot( ActionEdit2:name );
ActionEdit2:name = A_TKP01;

    /*************************************
```

```
        **** INSTANCE: LinkEdit
        ***************************************/
MakeInstance( LinkEdit, Menu );

/************ METHOD: Edit  ************/
MakeMethod( LinkEdit, Edit, [owner window choice ],
   {
   PostInputForm( "Edit Link " # owner # " - Creator: " # owner:Creator,
      owner, SVariable, " Secondary Variable:", owner, PItem, "Primary Item:",
      owner, PVariable, "Primary Variable", owner, MaxVar, "Max relation range",
      owner, MinVar, "Min relation randge", owner, TRelation, "Time Relation:",
      owner, VSRelation, " Secondary Variation Relation", owner,
      VPRelation, " Primary Variation Relation:", owner, EstimatedCost,
      "Estimated Cost:" );
   PostInputForm( "Edit Link " # owner # " - Creator: " # owner:Creator,
      owner, Failures, "Failures:", owner, Successes, "Successes:",
      owner, Wait, "Wait:", owner, Hold_Time, "Hold Time:" );
   } );

/************ METHOD: Delete  ************/
MakeMethod( LinkEdit, Delete, [owner window choice ],
   {
   If ( PostMenu( "Delete " # owner # " ?", YES, NO ) #= YES )
    Then DeleteInstance( owner );
   } );

/************ METHOD: Rename  ************/
MakeMethod( LinkEdit, Rename, [owner window choice ],
   {
   Self:Check = TRUE;
   While (Self:Check)
        {
        Self:name = owner;
        PostInputForm( " Rename Instance " # owner, Self:name,
           "New name" );
        If ( Not( Instance?( Self:name ) ) And Not( Null?( Self:name ) )
              And RenameInstance( owner, Self:name ) )
          Then Self:Check = FALSE;
        If ( Null?( Self:name ) Or Self:name #= owner )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR -  Instance already exists" );
        };
   } );

/************ METHOD: Move  ************/
MakeMethod( LinkEdit, Move, [owner window choice ],
   {
   Self:Check = TRUE;
   While (Self:Check)
        {
        PostInputForm( "Move " # owner, Self:name, "New parent" );
```

```
        If ( Class?( Self:name ) And Not( Null?( Self:name ) )
                And MoveInstance( owner, Self:name ) )
          Then Self:Check = FALSE;
        If ( Null?( Self:name ) Or Self:name #= GetParent( owner ) )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR -  Class does not exists" );
        };
   } );
LinkEdit:X = 178;
LinkEdit:Y = 104;
SetValue( LinkEdit:Choices, Edit, Delete, Rename );
SetValue( LinkEdit:ChoiceNames, &Edit, &Delete, &Rename );
MakeSlot( L      it:Check );
SetSlotOption( LinkEdit:Check, INHERIT, FALSE );
SetSlotOption( LinkEdit:Check, VALUE_TYPE, BOOLEAN );
LinkEdit:Check = FALSE;
SetSlotOption( LinkEdit:Check, IF_NEEDED, NULL );
SetSlotOption( LinkEdit:Check, WHEN_ACCESS, NULL );
SetSlotOption( LinkEdit:Check, BEFORE_CHANGE, NULL );
SetSlotOption( LinkEdit:Check, AFTER_CHANGE, NULL );
MakeSlot( LinkEdit:name );
LinkEdit:name = A_TKP04;

      /*************************************
       **** INSTANCE: Session1
       *************************************/
MakeInstance( Session1, KSession );
Session1:X = 89;
Session1:Y = 142;
Session1:Title = Interfaces;
Session1:SessionNumber = 1;
Session1:Width = 464;
Session1:Height = 251;
Session1:Menu = FALSE;
Session1:Visible = FALSE;
Session1:State = HIDDEN;
SetValue( Session1:BackgroundColor, 192, 192, 192 );
Session1:Titlebar = TRUE;
Session1:Sizebox = TRUE;
 ResetWindow ( Session1 );

      /*************************************
       **** INSTANCE: Explanation
       *************************************/
MakeInstance( Explanation, Root );

/************ METHOD: Add  *********/
MakeMethod( Explanation, Add, [text ],
   {
   If Self:Explain
     Then {
```

```
        DisplayText( HalEx, FormatValue( "\n  " # Time( ) #
                                         "   " # Date( )
                                         # "\n  " ) );
        DisplayText( HalEx, FormatValue( "  " # text # NULL ) );
        };
    } );

/************** METHOD: Display  **************/
MakeMethod( Explanation, Display, [text ],
    {
    If Self:Explain
      Then DisplayText( HalEx, FormatValue( "\n   " # text ) );
    } );

/************** METHOD: Clear  **************/
MakeMethod( Explanation, Clear, [],
    {
    ClearTranscriptImage( HalEx );
    ClearTranscriptImage( Error );
    } );

/************** METHOD: Error  **************/
MakeMethod( Explanation, Error, [text ],
    {
    If Self:Error
      Then DisplayText( Error, FormatValue( "\n " # Time( ) # "  "
                                         # text ) );
    } );

/************** METHOD: Action  **************/
MakeMethod( Explanation, Action, [Action Root ],
    {
    If Self:Explain
      Then {
        DisplayText( HalEx, FormatValue( "\n  " # Time( ) #
                                         "   " # Date( )
                                         # "  ACTION: " #
                                         Action # "\n    " ) );
        If Instance?( Root )
          Then DisplayText( HalEx, Root:Message );
        DisplayText( HalEx, FormatValue( "\n    " ) );
        DisplayText( HalEx, Action:ActionMesg );
        };
    } );
MakeSlot( Explanation:Explain );
SetSlotOption( Explanation:Explain, INHERIT, FALSE );
SetSlotOption( Explanation:Explain, VALUE_TYPE, BOOLEAN );
Explanation:Explain = TRUE;
SetSlotOption( Explanation:Explain, IF_NEEDED, NULL );
SetSlotOption( Explanation:Explain, WHEN_ACCESS, NULL );
SetSlotOption( Explanation:Explain, BEFORE_CHANGE, NULL );
SetSlotOption( Explanation:Explain, AFTER_CHANGE, NULL );
```

```
SetSlotOption( Explanation:Explain, IMAGE, CheckBox3 );
MakeSlot( Explanation:Error );
SetSlotOption( Explanation:Error, INHERIT, FALSE );
SetSlotOption( Explanation:Error, VALUE_TYPE, BOOLEAN );
Explanation:Error = TRUE;
SetSlotOption( Explanation:Error, IF_NEEDED, NULL );
SetSlotOption( Explanation:Error, WHEN_ACCESS, NULL );
SetSlotOption( Explanation:Error, BEFORE_CHANGE, NULL );
SetSlotOption( Explanation:Error, AFTER_CHANGE, NULL );
SetSlotOption( Explanation:Error, IMAGE, CheckBox2 );

    /*************************************
     **** INSTANCE: Files
     *************************************/
MakeInstance( Files, Menu );

/************ METHOD: Load  ************/
MakeMethod( Files, Load, [],
  {
  SendMessage( File, Load );
  Global:Runstate = NULL;
  } );

/************ METHOD: Save  ************/
MakeMethod( Files, Save, [],
  SendMessage( File, Save ) );

/************ METHOD: New  ************/
MakeMethod( Files, New, [],
  {
  If Global:Change
   Then {
       PostMessage( " Changes have not been Saved!" );
       If ( PostMenu( "Changes not Saved! Erase Knowledge Base?",
             Yes, No ) #= Yes )
        Then {
            Clear_KB( );
            };
       }
   Else {
       If ( PostMenu( "Erase Knowledge Base?", Yes, No )
            #= Yes )
        Then {
            Clear_KB( );
            };
       };
  File:BackUpFile = "C:\KAPPA\HAL\BACKUP.KNB";
  } );

/************ METHOD: Quit  ************/
MakeMethod( Files, Quit, [],
  {
```

```
   If Global:Change
    Then {
        PostMessage( " Changes have not been Saved!" );
        If ( PostMenu( "Changes not Saved! QUIT?", Yes, No )
              #= Yes )
         Then {
            Clear_KB( );
            Exit( NOPROMPT );
            };
        }
    Else {
        If ( PostMenu( QUIT?, Yes, No ) #= Yes )
         Then {
            Clear_KB( );
            Exit( NOPROMPT );
            };
        };
   } );

/************** METHOD: Settings  **************/
MakeMethod( Files, Settings, [],
   {
   PostInputForm( " HAL EXPERT SETTINGS", Global, ActionSpacing,
      "Action Release Rate [sec]:", Global, StdDelay, "Standard Delay Time [sec]:",
      Global, LinkChai  "Link Rules", Global, StationID, "HB Station ID number",
      Global, BackU  " Back up rate [Min]:", File, BackUpFile,
      " Backup File N  " );
   } );

/************** METHOD: Directory  **************/
MakeMethod( Files, Directory, [],
   {
   PostInputForm( "Set Directory", File, Path, "Path:" );
   } );
SetSlotOption( Files:ChoiceNames, NO_AUTO_ASK, TRUE );
SetValue( Files:ChoiceNames, &New, -, "&Load Knowledge", "&Save Knowledge", -, &Directory,
"&Settings ...", -, &Quit );
SetValue( Files:Choices, New, -, Load, Save, -, Directory, Settings, -, Quit );

    /*************************************
     **** INSTANCE: Times
     *************************************/
MakeInstance( Times, Root );

/************** METHOD: Date  **************/
MakeMethod( Times, Date, [],
   {
   Self:Month = NULL;
   Self:Day = NULL;
   Self:Date = Date( );
   Self:TLen = StringLength( Self:Date );
   For x From ( Self:TLen - 4 ) To ( Self:TLen - 3 )
```

```
        Do {
        Self:Char = SubString( Self:Date, x, x );
        If Number?( Self:Char )
          Then Self:Day = Self:Day # Self:Char;
        };
    For x From 1 To 2
        Do {
        Self:Char = SubString( Self:Date, x, x );
        If Number?( Self:Char )
          Then Self:Month = Self:Month # Self:Char;
        };
    If ( StringLength( Self:Month ) == 1 )
      Then Self:Month = 0 # Self:Month;
    If ( StringLength( Self:Day ) == 1 )
      Then Self:Day = 0 # Self:Day;
    } );
MakeSlot( Times:TimeNum );
SetSlotOption( Times:TimeNum, INHERIT, FALSE );
SetSlotOption( Times:TimeNum, VALUE_TYPE, NUMBER );
Times:TimeNum = 0;
SetSlotOption( Times:TimeNum, IF_NEEDED, NULL );
SetSlotOption( Times:TimeNum, WHEN_ACCESS, NULL );
SetSlotOption( Times:TimeNum, BEFORE_CHANGE, NULL );
SetSlotOption( Times:TimeNum, AFTER_CHANGE, NULL );
MakeSlot( Times:Char );
Times:Char = /;
MakeSlot( Times:Time );
SetSlotOption( Times:Time, INHERIT, FALSE );
SetSlotOption( Times:Time, VALUE_TYPE, NUMBER );
SetSlotOption( Times:Time, MINIMUM_VALUE, 0 );
Times:Time = 301.208791;
SetSlotOption( Times:Time, IF_NEEDED, NULL );
SetSlotOption( Times:Time, WHEN_ACCESS, NULL );
SetSlotOption( Times:Time, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Time, AFTER_CHANGE, NULL );
MakeSlot( Times:Date );
SetSlotOption( Times:Date, INHERIT, FALSE );
Times:Date = "4/4/93";
SetSlotOption( Times:Date, IF_NEEDED, NULL );
SetSlotOption( Times:Date, WHEN_ACCESS, NULL );
SetSlotOption( Times:Date, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Date, AFTER_CHANGE, NULL );
MakeSlot( Times:Hr );
SetSlotOption( Times:Hr, INHERIT, FALSE );
SetSlotOption( Times:Hr, VALUE_TYPE, NUMBER );
Times:Hr = 09;
SetSlotOption( Times:Hr, IF_NEEDED, NULL );
SetSlotOption( Times:Hr, WHEN_ACCESS, NULL );
SetSlotOption( Times:Hr, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Hr, AFTER_CHANGE, NULL );
MakeSlot( Times:Day );
SetSlotOption( Times:Day, INHERIT, FALSE );
```

```
SetSlotOption( Times:Day, VALUE_TYPE, NUMBER );
Times:Day = 04;
SetSlotOption( Times:Day, IF_NEEDED, NULL );
SetSlotOption( Times:Day, WHEN_ACCESS, NULL );
SetSlotOption( Times:Day, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Day, AFTER_CHANGE, NULL );
MakeSlot( Times:TLen );
SetSlotOption( Times:TLen, INHERIT, FALSE );
SetSlotOption( Times:TLen, VALUE_TYPE, NUMBER );
Times:TLen = 6;
SetSlotOption( Times:TLen, IF_NEEDED, NULL );
SetSlotOption( Times:TLen, WHEN_ACCESS, NULL );
SetSlotOption( Times:TLen, BEFORE_CHANGE, NULL );
SetSlotOption( Times:TLen, AFTER_CHANGE, NULL );
MakeSlot( Times:Sec );
SetSlotOption( Times:Sec, INHERIT, FALSE );
SetSlotOption( Times:Sec, VALUE_TYPE, NUMBER );
Times:Sec = 04;
SetSlotOption( Times:Sec, IF_NEEDED, NULL );
SetSlotOption( Times:Sec, WHEN_ACCESS, NULL );
SetSlotOption( Times:Sec, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Sec, AFTER_CHANGE, NULL );
MakeSlot( Times:Min );
SetSlotOption( Times:Min, INHERIT, FALSE );
SetSlotOption( Times:Min, VALUE_TYPE, NUMBER );
Times:Min = 48;
SetSlotOption( Times:Min, IF_NEEDED, NULL );
SetSlotOption( Times:Min, WHEN_ACCESS, NULL );
SetSlotOption( Times:Min, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Min, AFTER_CHANGE, NULL );
MakeSlot( Times:Month );
SetSlotOption( Times:Month, INHERIT, FALSE );
Times:Month = 04;
SetSlotOption( Times:Month, IF_NEEDED, NULL );
SetSlotOption( Times:Month, WHEN_ACCESS, NULL );
SetSlotOption( Times:Month, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Month, AFTER_CHANGE, NULL );
MakeSlot( Times:TCNo );
SetSlotOption( Times:TCNo, INHERIT, FALSE );
SetSlotOption( Times:TCNo, VALUE_TYPE, NUMBER );
Times:TCNo = 74;
SetSlotOption( Times:TCNo, IF_NEEDED, NULL );
SetSlotOption( Times:TCNo, WHEN_ACCESS, NULL );
SetSlotOption( Times:TCNo, BEFORE_CHANGE, NULL );
SetSlotOption( Times:TCNo, AFTER_CHANGE, NULL );
MakeSlot( Times:CycleNo );
SetSlotOption( Times:CycleNo, INHERIT, FALSE );
SetSlotOption( Times:CycleNo, VALUE_TYPE, NUMBER );
Times:CycleNo = 95;
SetSlotOption( Times:CycleNo, IF_NEEDED, NULL );
SetSlotOption( Times:CycleNo, WHEN_ACCESS, NULL );
SetSlotOption( Times:CycleNo, BEFORE_CHANGE, NULL );
```

```
SetSlotOption( Times:CycleNo, AFTER_CHANGE, NULL );
MakeSlot( Times:Max );
SetSlotOption( Times:Max, INHERIT, FALSE );
SetSlotOption( Times:Max, VALUE_TYPE, NUMBER );
Times:Max = 336;
SetSlotOption( Times:Max, IF_NEEDED, NULL );
SetSlotOption( Times:Max, WHEN_ACCESS, NULL );
SetSlotOption( Times:Max, BEFORE_CHANGE, NULL );
SetSlotOption( Times:Max, AFTER_CHANGE, NULL );
MakeSlot( Times:CyclePeriod );
SetSlotOption( Times:CyclePeriod, INHERIT, FALSE );
SetSlotOption( Times:CyclePeriod, VALUE_TYPE, NUMBER );
SetSlotOption( Times:CyclePeriod, MINIMUM_VALUE, 0 );
Times:CyclePeriod = 3600.1;
SetSlotOption( Times:CyclePeriod, IF_NEEDED, NULL );
SetSlotOption( Times:CyclePeriod, WHEN_ACCESS, NULL );
SetSlotOption( Times:CyclePeriod, BEFORE_CHANGE, NULL );
SetSlotOption( Times:CyclePeriod, AFTER_CHANGE, NULL );

    /*************************************
     **** INSTANCE: Explain
     *************************************/
MakeInstance( Explain, Menu );

/************** METHOD: Clear  **************/
MakeMethod( Explain, Clear, [],
  {
  SendMessage( Explanation, Clear );
  } );

/************** METHOD: Load  **************/
MakeMethod( Explain, Load, [],
  {
  File:EFileName = SelectFile( " Load Explanation File",
                   File:Path # "*.Exp" );
  If Not( Null?( File:EFileName ) )
   Then {
       PostBusy( ON, "Loading ..." );
       CatchError( DisplayFile( HalEx, File:EFileName ),
          {
          Error( 5, NULL, NULL );
          } );
       PostBusy( OFF );
       };
  } );

/************** METHOD: Save  **************/
MakeMethod( Explain, Save, [],
  {
  File:EFileName = SelectFile( "Save Explanation Window",
                   File:Path # "*.Exp" );
  If Not( Null?( File:EFileName ) )
```

```
    Then {
        If CatchError( OpenWriteFile( File:EFileName ), FALSE )
          Then {
              PostBusy( ON, "Saving ..." );
              SaveTranscriptImage( HalEx );
              CloseWriteFile( );
              PostBusy( OFF );
              }
          Else {
              Error( 6, NULL, NULL );
              };
        };
   } );

/************** METHOD: Exp  **************/
MakeMethod( Explain, Exp, [owner window choice ],
   {
   If ( choice #= Enable )
    Then {
        Explanation:Explain = TRUE;
        AppendToList( Self:Disabled, Enable );
        If Member?( Self:Disabled, Disable )
         Then RemoveFromList( Self:Disabled, Disable );
        };
   If ( choice #= Disable )
    Then {
        Explanation:Explain = FALSE;
        AppendToList( Self:Disabled, Disable );
        If Member?( Self:Disabled, Enable )
         Then RemoveFromList( Self:Disabled, Enable );
        };
   } );
SetValue( Explain:Choices, Save, Load, -, Clear );
SetValue( Explain:ChoiceNames, &Save, &Load, -, &Clear );
SetValue( Explain:Disabled, Disable, Disable, Disable, Enable );
Explain:DefaultMethod = Exp;

     /*************************************
      **** INSTANCE: InterfaceMenu
      *************************************/
MakeInstance( InterfaceMenu, Menu );

/************** METHOD: Close  **************/
MakeMethod( InterfaceMenu, Close, [],
   {
   HideWindow( Session1 );
   } );
SetValue( InterfaceMenu:Choices, Close );
SetValue( InterfaceMenu:ChoiceNames, &Close );

     /*************************************
      **** INSTANCE: Inter
```

```
        *********(**********************/
MakeInstance( Inter, Root );

/************ METHOD: Update  ************/
MakeMethod( Inter, Update, [],
   {
   ResetImage( ComboBox5 );
   } );

/************** METHOD: Type  **************/
MakeMethod( Inter, Type, [],
   {
   If ( Self:VariableType #= Analog )
    Then {
        ShowImage( Slider1 );
        HideImage( Edit2 );
        };
   If ( Self:VariableType #= Discrete )
    Then {
        ShowImage( Edit2 );
        HideImage( Slider1 );
        };
   } );

/************** METHOD: Change  **************/
MakeMethod( Inter, Change, [],
   {
   If ( Not( Null?( Self:Item ) ) And Not( Null?( Self:Variable ) ) )
    Then {
        If Instance?( Self:Item )
         Then {
            Self:VariableType = SendMessage( Self:Item, GetVal,
                                Self:Variable, TYPE );
            Self:TagName = SendMessage( Self:Item, GetVal,
                           Self:Variable, TAG );
            Self:Value = GetValue( GetValue( {
                                        Self:Item;
                                        },
                                    {
                                    N # Self:Variable;
                                    } ), Variable );
            If ( Self:Value #= ERROR )
             Then Beep( );
            }
         Else {
            Beep( );
            };
        );
   } );
MakeSlot( Inter:Value );
SetSlotOption( Inter:Value, INHERIT, FALSE );
SetSlotOption( Inter:Value, NO_AUTO_ASK, TRUE );
```

```
Inter:Value = 51.3;
SetSlotOption( Inter:Value, IF_NEEDED, NULL );
SetSlotOption( Inter:Value, WHEN_ACCESS, NULL );
SetSlotOption( Inter:Value, BEFORE_CHANGE, NULL );
SetSlotOption( Inter:Value, AFTER_CHANGE, NULL );
SetSlotOption( Inter:Value, IMAGE, ComboBox2, Slider1, Edit2 );
MakeSlot( Inter:TagName );
SetSlotOption( Inter:TagName, INHERIT, FALSE );
Inter:TagName = FIP022;
SetSlotOption( Inter:TagName, IF_NEEDED, NULL );
SetSlotOption( Inter:TagName, WHEN_ACCESS, NULL );
SetSlotOption( Inter:TagName, BEFORE_CHANGE, NULL );
SetSlotOption( Inter:TagName, AFTER_CHANGE, NULL );
SetSlotOption( Inter:TagName, IMAGE, ComboBox7, Edit1 );
MakeSlot( Inter:VariableType );
SetSlotOption( Inter:VariableType, INHERIT, FALSE );
SetSlotOption( Inter:VariableType, ALLOWABLE_VALUES, Analog, Discrete );
Inter:VariableType = Analog;
SetSlotOption( Inter:VariableType, IF_NEEDED, NULL );
SetSlotOption( Inter:VariableType, WHEN_ACCESS, NULL );
SetSlotOption( Inter:VariableType, BEFORE_CHANGE, NULL );
SetSlotOption( Inter:VariableType, AFTER_CHANGE, Type );
SetSlotOption( Inter:VariableType, IMAGE, RadioButtonGroup1 );
MakeSlot( Inter:Item );
SetSlotOption( Inter:Item, INHERIT, FALSE );
Inter:Item = P22;
SetSlotOption( Inter:Item, IF_NEEDED, NULL );
SetSlotOption( Inter:Item, WHEN_ACCESS, NULL );
SetSlotOption( Inter:Item, BEFORE_CHANGE, NULL );
SetSlotOption( Inter:Item, AFTER_CHANGE, Change );
SetSlotOption( Inter:Item, IMAGE, ComboBox4 );
MakeSlot( Inter:Variable );
SetSlotOption( Inter:Variable, INHERIT, FALSE );
Inter:Variable = Flow;
SetSlotOption( Inter:Variable, IF_NEEDED, NULL );
SetSlotOption( Inter:Variable, WHEN_ACCESS, NULL );
SetSlotOption( Inter:Variable, BEFORE_CHANGE, NULL );
SetSlotOption( Inter:Variable, AFTER_CHANGE, Change );
SetSlotOption( Inter:Variable, IMAGE, ComboBox5, ComboBox2 );

    /*****************************************
     **** INSTANCE: File
     *****************************************/
MakeInstance( File, Root );

/************ METHOD: Save  ************/
MakeMethod( File, Save, [],
  {
  Self:FileName = SelectFile( "Save Knowledge Base", Self:FileName );
  If Not( Null?( Self:FileName ) )
   Then {
      PostBusy( ON, "Saving Knowledge ..." );
```

```
        SendMessage( Self, Write, Self:FileName );
        Global:Change = FALSE;
        PostBusy( OFF );
        }
     Else {
        Error( 2, NULL, NULL );
        };
  } );

/************** METHOD: Load  ***************/
MakeMethod( File, Load, [],
   {
   Self:BackUpFile = "C:\KAPPA\HAL\BACKUP.KNB";
   Self:FileName = SelectFile( " Load Knowledge Base",
                  Self:Path # "*.KNB" );
   If Not( Null?( Self:FileName ) )
    Then {
        PostBusy( ON, "Loading ..." );
        Clear_KB( );
        CatchError( {
                  InterpretFile( Self:FileName );
                  Self:BackUpFile = Self:FileName;
                  },
           {
           Error( 3, NULL, NULL );
           } );
        PostBusy( OFF );
        };
  } );

/************** METHOD: Write  **************/
MakeMethod( File, Write, [FileName ],
   {
   If CatchError( OpenWriteFile( FileName ), Error( 24, NULL,
                                    FileName ) )
    Then {
        EnumSubClasses( Items, Item,
           {
           WriteClass( Item );
           GetInstanceList( Item, Self:InstanceList );
           EnumList( Self:InstanceList, Inst,
              {
              WriteInstance( Inst );
              } );
           } );
        EnumSubClasses( Actions, Action,
           {
           If Not( Action #= Default )
            Then WriteClass( Action );
           GetInstanceList( Action, Self:InstanceList );
           EnumList( Self:InstanceList, Inst,
              {
```

```
            WriteInstance( Inst );
            } );
         } );
      EnumSubClasses( Links, Link,
         {
         WriteClass( Link );
         GetInstanceList( Link, Self:InstanceList );
         EnumList( Self:InstanceList, Inst,
            {
            WriteInstance( Inst );
            } );
         } );
      EnumSubClasses( Variables, Var,
         {
         WriteClass( Var );
         GetInstanceList( Var, Self:InstanceList );
         EnumList( Self:InstanceList, Inst,
            {
            WriteInstance( Inst );
            } );
         } );
      WriteAllRules( );
      WriteInstance( Rule );
      CloseWriteFile( );
      PostBusy( OFF );
      };
   } );

/************ METHOD: BackUp  ************/
MakeMethod( File, BackUp, [],
   {
   PostBusy( ON, "Timed backup ..." );
   SendMessage( Self, Write, Self:BackUpFile );
   PostBusy( OFF );
   } );
MakeSlot( File:FileName );
SetSlotOption( File:FileName, INHERIT, FALSE );
File:FileName = "C:\KAPPA\HAL\TEST1.KNB";
MakeSlot( File:InstanceList );
SetSlotOption( File:InstanceList, INHERIT, FALSE );
SetSlotOption( File:InstanceList, MULTIPLE );
SetValue( File:InstanceList, TKP01_Level );
SetSlotOption( File:InstanceList, IF_NEEDED, NULL );
SetSlotOption( File:InstanceList, WHEN_ACCESS, NULL );
SetSlotOption( File:InstanceList, BEFORE_CHANGE, NULL );
SetSlotOption( File:InstanceList, AFTER_CHANGE, NULL );
MakeSlot( File:EFileName );
File:EFileName = "C:\KAPPAPC\HAL\Hal0404.Exp";
MakeSlot( File:Path );
SetSlotOption( File:Path, INHERIT, FALSE );
File:Path = FormatValue ( "C:\\KAPPAPC\\HAL\\" );
SetSlotOption( File:Path, IF_NEEDED, NULL );
```

```
SetSlotOption( File:Path, WHEN_ACCESS, NULL );
SetSlotOption( File:Path, BEFORE_CHANGE, NULL );
SetSlotOption( File:Path, AFTER_CHANGE, NULL );
MakeSlot( File:BackUpFile );
SetSlotOption( File:BackUpFile, INHERIT, FALSE );
File:BackUpFile = "C:\KAPPA\HAL\BACKUP.KNB";

/*****************************************
 **** INSTANCE: BrowserMenu
 *****************************************/
MakeInstance( BrowserMenu, Menu );

/************** METHOD: Close  **************/
MakeMethod( BrowserMenu, Close, [],
  HideWindow( BROWSER ) );

/************** METHOD: Return  **************/
MakeMethod( BrowserMenu, Return, [],
  {
  HideWindow( BROWSER );
  If Function?( ShowInstanceMenu )
    Then DeleteFunction( ShowInstanceMenu );
  MakeFunction( ShowInstanceMenu, [ obj ], PopupMenu( ItemEdit,
                                    obj, BROWSER ) );
  ShowInstanceMenu;
  If Function?( ShowClassMenu )
    Then DeleteFunction( ShowClassMenu );
  MakeFunction( ShowClassMenu, [ obj ], PopupMenu( ItemEdit3,
                                   obj, BROWSER ) );
  ShowClassMenu;
  PositionWindow( BROWSER, 70, 70, 300, 300 );
  SetMenuBar( BROWSER, BrowserMenu );
  SetBrowserFocus( Items );
  SetWindowTitle( BROWSER, "Item Browser" );
  ShowWindow( BROWSER );
  } );
SetValue( BrowserMenu:Choices, Close, Return );
SetValue( BrowserMenu:ChoiceNames, &Close, &Return );

/*****************************************
 **** INSTANCE: EditDevice
 *****************************************/
MakeInstance( EditDevice, Menu );

/************** METHOD: InitMenu  **************/
MakeMethod( EditDevice, InitMenu, [owner ],
  {
  ClearList( Self:ChoiceNames );
  ClearList( Self:Choices );
  AppendToList( Self:ChoiceNames, Self:TempChoiceNames );
  AppendToList( Self:Choices, Self:TempChoices );
  ClearList( Self:VarList );
```

```
    ClearList( Self:SlotList );
    GetSlotList( owner, Self:SlotList );
    EnumList( Self:SlotList, Value,
       {
       If Member?( Items:ValidSlots, Value )
        Then {
            AppendToList( Self:VarList, Value );
            AppendToList( Self:ChoiceNames, Value );
            AppendToList( Self:Choices, Value );
            };
       } );
    } );

/************** METHOD: Rename  **************/
MakeMethod( EditDevice, Rename, [owner window choice ],
    {
    Self:Check = TRUE;
    If ( GetOwner( owner, choice ) #= owner )
     Then {
         While  (Self:Check)
              {
              Self:name = owner;
              PostInputForm( " Rename Variable " # owner, Self:name,
                 "New name" );
              If ( Not( Slot?( owner, Self:name ) )
                      And Not( Null?( Self:name ) ) )
               Then {
                   If Not( Member?( owner:ValidSlots, Self:name ) )
                     Then AppendToList( owner:ValidSlots,
                            Self:name );
                   RenameSlot( owner, choice, Self:name );
                   RemoveFromList( owner:ValidSlots, choice );
                   Self:Check = FALSE;
                   };
              If ( Null?( Self:name ) Or Self:name #= owner )
               Then Self:Check = FALSE;
              If Self:Check
               Then PostMessage( "ERROR -  Variable already exists" );
              };
         }
     Else {
         Self:Check = FALSE;
         PostMessage( "ERROR - Variable is inherited" );
         };
    } );

/************** METHOD: Edit  **************/
MakeMethod( EditDevice, Edit, [owner window choice ],
    {
    PostInputForm( "Edit Slot", owner:choice, " Slot" );
    } );
```

```
/************** METHOD: Devices  **************/
MakeMethod( EditDevice, Devices, [owner window choice ],
   {
   Self:Function = PopupMenu( Choice );
   If ( Self:Function #= Edit )
     Then SendMessage( Self, Edit, owner, window, choice );
   If ( Self:Function #= Delete )
     Then SendMessage( Self, Delete, owner, window, choice );
   If ( Self:Function #= Rename )
     Then SendMessage( Self, Rename, owner, window, choice );
   } );

/************** METHOD: Delete  **************/
MakeMethod( EditDevice, Delete, [owner window choice ],
   {
   Self:name = PostMenu( "
Delete variable " # choice # " ?", YES,
                NO );
   If ( Self:name #= YES )
     Then {
          DeleteSlot( owner, choice );
          If Member?( Items:ValidSlots, choice )
            Then RemoveFromList( Items:ValidSlots, choice );
          };
   } );

/************** METHOD: AddDevice  **************/
MakeMethod( EditDevice, AddDevice, [owner window choice ],
   {
   Self:Check = TRUE;
   While  (Self:Check)
        {
        Self:Check = FALSE;
        Self:name = NULL;
        PostInputForm( " Add Variable - " # owner, Self:name, "Variable Name" );
        EnumList( Self:VarList, V,
           {
           If ( Not( Null?( Self:name ) ) And ( V #= Self:name ) )
             Then {
                  Self:Check = TRUE;
                  Self:Error = 1;
                  };
           } );
        If Member?( Items:ValidVars,
             {
             Self:name;
             } )
          Then {
             Self:Check = TRUE;
             Self:Error = 2;
             };
        If ( Not( Null?( Self:name ) ) And Self:Check #= FALSE )
```

```
        Th n {
            If Not( Slot?( owner, Self:name ) )
              Then {
                  MakeSlot( owner, Self:name );
                  };
            If Not( Member?( Items:ValidSlots, Self:name ) )
              Then AppendToList( Items:ValidSlots, Self:name );
            SendMessage( Self, Edit, owner, window, Self:name );
            }
          Else {
            Self:Error = 2;
            };
        If Null?( Self:n me )
          Then Self:Check = FALSE;
        If Self:Check
          Then {
            If ( Self:Error == 1 )
              Then PostMessage( "ERROR - Device already exists " );
            If ( Self:Error == 2 )
              Then PostMessage( "ERROR - Reserved name" );
            };
        };
    } );
SetValue( EditDevice:Choices, AddDevice, - );
SetValue( EditDevice:ChoiceNames, "&Add Device", - );
EditDevice:DefaultMethod = Devices;
MakeSlot( EditDevice:SlotList );
SetSlotOption( EditDevice:SlotList, INHERIT, FALSE );
SetSlotOption( EditDevice:SlotList, NO_AUTO_ASK, TRUE );
SetSlotOption( EditDevice:SlotList, MULTIPLE );
SetValue( EditDevice:SlotList, Inputs, Outputs, ValidVars, Value_List, NodeName, Counter, ValidSlots,
ItemNo, Level, NLevel );
SetSlotOption( EditDevice:SlotList, IF_NEEDED, NULL );
SetSlotOption( EditDevice:SlotList, WHEN_ACCESS, NULL );
SetSlotOption( EditDevice:SlotList, BEFORE_CHANGE, NULL );
SetSlotOption( EditDevice:SlotList, AFTER_CHANGE, NULL );
MakeSlot( EditDevice:VarList );
SetSlotOption( EditDevice:VarList, INHERIT, FALSE );
SetSlotOption( EditDevice:VarList, MULTIPLE );
ClearList( EditDevice:VarList );
MakeSlot( EditDevice:TempChoiceNames );
SetSlotOption( EditDevice:TempChoiceNames, INHERIT, FALSE );
SetSlotOption( EditDevice:TempChoiceNames, MULTIPLE );
SetValue( EditDevice:TempChoiceNames, "&Add Device", - );
SetSlotOption( EditDevice:TempChoiceNames, IF_NEEDED, NULL );
SetSlotOption( EditDevice:TempChoiceNames, WHEN_ACCESS, NULL );
SetSlotOption( EditDevice:TempChoiceNames, BEFORE_CHANGE, NULL );
SetSlotOption( EditDevice:TempChoiceNames, AFTER_CHANGE, NULL );
MakeSlot( EditDevice:TempChoices );
SetSlotOption( EditDevice:TempChoices, INHERIT, FALSE );
SetSlotOption( EditDevice:TempChoices, MULTIPLE );
SetValue( EditDevice:TempChoices, AddDevice, - );
```

```
SetSlotOption( EditDevice:TempChoices, IF_NEEDED, NULL );
SetSlotOption( EditDevice:TempChoices, WHEN_ACCESS, NULL );
SetSlotOption( EditDevice:TempChoices, BEFORE_CHANGE, NULL );
SetSlotOption( EditDevice:TempChoices, AFTER_CHANGE, NULL );
MakeSlot( EditDevice:Check );
SetSlotOption( EditDevice:Check, INHERIT, FALSE );
SetSlotOption( EditDevice:Check, VALUE_TYPE. BOOLEAN );
EditDevice:Check = FALSE;
SetSlotOption( EditDevice:Check, IF_NEEDED, NULL );
SetSlotOption( EditDevice:Check, WHEN_ACCESS, NULL );
SetSlotOption( EditDevice:Check, BEFORE_CHANGE, NULL );
SetSlotOption( EditDevice:Check, AFTER_CHANGE, NULL );
MakeSlot( EditDevice:Error );
SetSlotOption( EditDevice:Error, INHERIT, FALSE );
SetSlotOption( EditDevice:Error, VALUE_TYPE, NUMBER );
EditDevice:Error = 2;
SetSlotOption( EditDevice:Error, IF_NEEDED, NULL );
SetSlotOption( EditDevice:Error, WHEN_ACCESS, NULL );
SetSlotOption( EditDevice:Error, BEFORE_CHANGE, NULL );
SetSlotOption( EditDevice:Error, AFTER_CHANGE, NULL );
MakeSlot( EditDevice:name );
SetSlotOption( EditDevice:name, INHERIT, FALSE );
SetSlotOption( EditDevice:name, IF_NEEDED, NULL );
SetSlotOption( EditDevice:name, WHEN_ACCESS, NULL );
SetSlotOption( EditDevice:name, BEFORE_CHANGE, NULL );
SetSlotOption( EditDevice:name, AFTER_CHANGE, NULL );
MakeSlot( EditDevice:Function );
EditDevice:Function = Delete;

    /*****************************************
     **** INSTANCE: ItemVars
     *****************************************/
MakeInstance( ItemVars, Root );

/************ METHOD: Init  ************/
MakeMethod( ItemVars, Init, [owner choice VariableName ],
   {
   Text5:Title = "Variable " # owner # " - " # choice;
   Self:Owner = owner;
   Self:Choice = choice;
   Self:VarName = VariableName;
   Self:Ok = TRUE;
   Self:UpperLimit = GetValue( {
                        Self:VarName;
                        }, UpperLevel );
   Self:LowerLimit = GetValue( {
                        Self:VarName;
                        }, LowerLevel );
   Self:SetPoint = GetValue( {
                       Self:VarName;
                       }, SetPoint );
   Self:LongText = GetValue( {
```

```
                        Self:VarName;
                        }, LongText );
    Self:ShortText = GetValue( {
                        Self:VarName;
                        }, ShortText );
    Self:PlantArea = GetValue( {
                        Self:VarName;
                        }, PlantArea );
    Self:Dimension = GetValue( {
                        Self:VarName;
                        }, Units );
    Self:Type = GetValue( {
                      Self:VarName;
                      }, MesType );
    Self:Value = GetValue( {
                       Self:VarName;
                       }, Variable );
    Self:VarNo = GetValue( {
                       Self:VarName;
                       }, Number );
    If ( Self:Type #= ANALOG )
     Then {
         SendMessage( Self, Analog );
         }
     Else {
         If ( Self:Type #= BINARY )
          Then {
              SendMessage( Self, Binary );
              }
          Else SendMessage( Self:Counter );
         };
    Self:Tag = GetValue( {
                     Self:VarName;
                     }, Tag );
    ShowWindow( Session2 );
    } );

/************** METHOD: GetData  **************/
MakeMethod( ItemVars, GetData, {},
    {
    If Not( Self:Tag #= NONE )
     Then {
         If Not( CatchError( {
                         HBRead( Self:Tag, 1 );
                         }, HBError( ) ) #= 40 )
          Then {
              Self:Dimension = CatchError( {
                                     HBRead( Self:Tag,
                                       2 );
                                     },
                                   {
                                   SendMessage( Self, Error );
```

```
                    } );
Self:PlantArea = CatchError( {
                        HBRead( Self:Tag,
                           5 );
                        },
                    {
                    SendMessage( Self, Error );
                    } );
Self:ShortText = CatchError( {
                        HBRead( Self:Tag,
                           6 );
                        },
                    {
                    SendMessage( Self, Error );
                    } );
Self:LongText = CatchError( {
                        HBRead( Self:Tag,
                           7 );
                        },
                    {
                    SendMessage( Self, Error );
                    } );
Self:Type = HBType( );
Self:Condition = CatchError( HBCond( ), NULL );
If ( Self:Type #= COUNTER )
  Then {}
  Else If ( Self:Type #= BINARY )
       Then {
           SendMessage( Self, Binary );
           Self:State1 = CatchError( {
                                HBRead( Self:Tag,
                                   2 );
                                },
                           {
                           SendMessage( Self,
                              Error );
                           } );
           Self:State0 = CatchError( {
                                HBRead( Self:Tag,
                                   3 );
                                },
                           {
                           SendMessage( Self,
                              Error );
                           } );
           }
       Else {
           SendMessage( Self, Analog );
           Self:Min = CatchError( {
                                HBRead( Self:Tag,
                                   4 );
                                },
```

```
                                        {
                                        SendMessage( Self,
                                           Error );
                                        } );
                                Self:Max = CatchError( {
                                              HBRead( Self:Tag,
                                                 3 );
                                              },
                                        {
                                        SendMessage( Self,
                                           Error );
                                        } );
                                };
                        };
                    };
                };

/************** METHOD: Error  **************/
MakeMethod( ItemVars, Error, [],
    {
    Beep( );
    SendMessage( Explanation, Error, HBErrorMsg( ) );
    SendMessage( Explanation, Error, Self:Tag );
    CatchError( HBClose( ), FALSE );
    CatchError( HBOpen( Global:StationID ), FALSE );
    Self:Ok = FALSE;
    NULL;
    } );

/************** METHOD: Analog  **************/
MakeMethod( ItemVars, Analog, [],
    {
    ShowImage( Slider2 );
    ShowImage( Slider3 );
    ShowImage( Slider4 );
    HideImage( Edit3 );
    HideImage( Edit5 );
    HideImage( Text11 );
    HideImage( Text13 );
    HideImage( Text16 );
    ShowImage( Edit6 );
    ShowImage( Edit7 );
    ShowImage( Edit4 );
    ShowImage( Text4 );
    ShowImage( Text6 );
    ShowImage( Text7 );
    If ( Not( Null?( Self:Max ) ) And Not( Null?( Self:Min ) ) )
      Then {
          SetSlotOption( Self, SetPoint, MAXIMUM_VALUE, Self:Max );
          SetSlotOption( Self, SetPoint, MINIMUM_VALUE, Self:Min );
          SetSlotOption( Self, UpperLimit, MAXIMUM_VALUE, Self:Max );
          SetSlotOption( Self, UpperLimit, MINIMUM_VALUE, Self:Min );
```

```
        SetSlotOption( Self, LowerLimit, MAXIMUM_VALUE, Self:Max );
        SetSlotOption( Self, LowerLimit, MINIMUM_VALUE, Self:Min );
        };
    } );

/************** METHOD: Binary  **************/
MakeMethod( ItemVars, Binary, [],
    {
    HideImage( Text4 );
    HideImage( Text6 );
    HideImage( Text7 );
    HideImage( Edit4 );
    HideImage( Edit6 );
    HideImage( Edit7 );
    ShowImage( Edit3 );
    ShowImage( Edit5 );
    ShowImage( Text11 );
    ShowImage( Text13 );
    ShowImage( Text16 );
    HideImage( Slider2 );
    HideImage( Slider3 );
    HideImage( Slider4 );
    } );

/************** METHOD: Update  **************/
MakeMethod( ItemVars, Update, [],
    {
    If ( Ok And Instance?( Self:VarName ) )
     Then {
        Self:VarName:Tag = Self:Tag;
        Self:VarName:SetPoint = Self:SetPoint;
        Self:VarName:UpperLevel = Self:UpperLimit;
        Self:VarName:LowerLevel = Self:LowerLimit;
        Self:VarName:Units = Self:Dimension;
        Self:VarName:PlantArea = Self:PlantArea;
        Self:VarName:LongText = Self:LongText;
        Self:VarName:ShortText = Self:ShortText;
        Self:VarName:MesType = Self:Type;
        Self:VarName:Variable = Self:Value;
        Self:VarName:Number = Self:VarNo;
        If ( Self:Type #= ANALOG )
         Then {
            Self:VarName:Type = Analog;
            };
        If ( Self:Type #= COUNTER )
         Then {
            Self:VarName:Type = Analog;
            };
        If ( Self:Type #= BINARY )
         Then {
            Self:VarName:Type = Discrete;
            ClearList( Self:VarName:StateList );
```

```
            ClearList( Self:VarName:StateValues );
            AppendToList( Self:VarName:StateList, Self:State1 );
            AppendToList( Self:VarName:StateValues, 1 );
            AppendToList( Self:VarName:StateList, Self:State0 );
            AppendToList( Self:VarName:StateValues, 0 );
            };
        If ( Not( Null?( Self:Max ) ) And Not( Null?( Self:Min ) ) )
          Then {
            SetSlotOption( {
                        Self:Owner;
                        },
              {
              Self:Choice;
              }, MAXIMUM_VALUE, Self:Max );
            SetSlotOption( {
                        Self:Owner;
                        },
              {
              Self:Choice;
              }, MINIMUM_VALUE, Self:Min );
            SetSlotOption( GetValue( Self:VarName ), SetPoint,
              MAXIMUM_VALUE, Self:Max );
            SetSlotOption( GetValue( Self:VarName ), SetPoint,
              MINIMUM_VALUE, Self:Min );
            SetSlotOption( GetValue( Self:VarName ), LowerLevel,
              MAXIMUM_VALUE, Self:Max );
            SetSlotOption( GetValue( Self:VarName ), LowerLevel,
              MINIMUM_VALUE, Self:Min );
            SetSlotOption( GetValue( Self:VarName ), UpperLevel,
              MAXIMUM_VALUE, Self:Max );
            SetSlotOption( GetValue( Self:VarName ), UpperLevel,
              MINIMUM_VALUE, Self:Min );
            };
        };
    HideWindow( Session2 );
    ShowWindow( BROWSER );
    } );

/************** METHOD: Counter  **************/
MakeMethod( ItemVars, Counter, [],
    {
    ShowImage( Slider2 );
    ShowImage( Slider3 );
    ShowImage( Slider4 );
    HideImage( Edit3 );
    HideImage( Edit5 );
    ShowImage( Edit6 );
    ShowImage( Edit7 );
    ShowImage( Edit4 );
    ShowImage( Text4 );
    ShowImage( Text6 );
    ShowImage( Text7 );
```

```
    SetSlotOption( Self, SetPoint, MAXIMUM_VALUE, Self:Max );
    SetSlotOption( Self, SetPoint, MINIMUM_VALUE, Self:Min );
    SetSlotOption( Self, UpperLimit, MAXIMUM_VALUE, Self:Max );
    SetSlotOption( Self, UpperLimit, MINIMUM_VALUE, Self:Min );
    SetSlotOption( Self, LowerLimit, MAXIMUM_VALUE, Self:Max );
    SetSlotOption( Self, LowerLimit, MINIMUM_VALUE, Self:Min );
    } );

/************** METHOD: Type  **************/
MakeMethod( ItemVars, Type, [],
    {
    If ( Self:Type #= ANALOG )
      Then SendMessage( Self, Analog );
    If ( Self:Type #= COUNTER )
      Then SendMessage( Self, Counter );
    If ( Self:Type #= BINARY )
      Then SendMessage( Self, Binary );
    } );
MakeSlot( ItemVars:Max );
SetSlotOption( ItemVars:Max, INHERIT, FALSE );
SetSlotOption( ItemVars:Max, VALUE_TYPE, NUMBER );
ItemVars:Max = 100;
SetSlotOption( ItemVars:Max, IF_NEEDED, NULL );
SetSlotOption( ItemVars:Max, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:Max, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:Max, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:Max, IMAGE, Edit6 );
MakeSlot( ItemVars:Min );
SetSlotOption( ItemVars:Min, INHERIT, FALSE );
SetSlotOption( ItemVars:Min, VALUE_TYPE, NUMBER );
ItemVars:Min = 0;
SetSlotOption( ItemVars:Min, IF_NEEDED, NULL );
SetSlotOption( ItemVars:Min, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:Min, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:Min, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:Min, IMAGE, Edit7 );
MakeSlot( ItemVars:Condition );
ItemVars:Condition = NULL;
MakeSlot( ItemVars:Tag );
SetSlotOption( ItemVars:Tag, INHERIT, FALSE );
ItemVars:Tag = LIP002;
SetSlotOption( ItemVars:Tag, AFTER_CHANGE, GetData );
SetSlotOption( ItemVars:Tag, IMAGE, Edit12 );
MakeSlot( ItemVars:VarName );
SetSlotOption( ItemVars:VarName, INHERIT, FALSE );
ItemVars:VarName = TKP01_Level;
SetSlotOption( ItemVars:VarName, IF_NEEDED, NULL );
SetSlotOption( ItemVars:VarName, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:VarName, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:VarName, AFTER_CHANGE, NULL );
MakeSlot( ItemVars:ShortText );
ItemVars:ShortText = Level;
```

```
SetSlotOption( ItemVars:ShortText, IMAGE, Edit8 );
MakeSlot( ItemVars:LongText );
SetSlotOption( ItemVars:LongText, INHERIT, FALSE );
ItemVars:LongText = NULL;
SetSlotOption( ItemVars:LongText, IF_NEEDED, NULL );
SetSlotOption( ItemVars:LongText, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:LongText, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:LongText, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:LongText, IMAGE, Edit9 );
MakeSlot( ItemVars:Type );
SetSlotOption( ItemVars:Type, INHERIT, FALSE );
ItemVars:Type = ANALOG;
SetSlotOption( ItemVars:Type, IF_NEEDED, NULL );
SetSlotOption( ItemVars:Type, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:Type, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:Type, AFTER_CHANGE, Type );
SetSlotOption( ItemVars:Type, IMAGE, RadioButtonGroup3 );
MakeSlot( ItemVars:Dimension );
SetSlotOption( ItemVars:Dimension, INHERIT, FALSE );
ItemVars:Dimension = FormatValue ( "\%" );
SetSlotOption( ItemVars:Dimension, IF_NEEDED, NULL );
SetSlotOption( ItemVars:Dimension, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:Dimension, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:Dimension, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:Dimension, IMAGE, Edit4 );
MakeSlot( ItemVars:PlantArea );
SetSlotOption( ItemVars:PlantArea, INHERIT, FALSE );
ItemVars:PlantArea = P;
SetSlotOption( ItemVars:PlantArea, IF_NEEDED, NULL );
SetSlotOption( ItemVars:PlantArea, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:PlantArea, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:PlantArea, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:PlantArea, IMAGE, Edit11 );
MakeSlot( ItemVars:State1 );
ItemVars:State1 = ON;
SetSlotOption( ItemVars:State1, IMAGE, Edit3 );
MakeSlot( ItemVars:State0 );
SetSlotOption( ItemVars:State0, INHERIT, FALSE );
ItemVars:State0 = OFF;
SetSlotOption( ItemVars:State0, IF_NEEDED, NULL );
SetSlotOption( ItemVars:State0, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:State0, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:State0, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:State0, IMAGE, Edit5 );
MakeSlot( ItemVars:UpperLimit );
SetSlotOption( ItemVars:UpperLimit, INHERIT, FALSE );
SetSlotOption( ItemVars:UpperLimit, VALUE_TYPE, NUMBER );
SetSlotOption( ItemVars:UpperLimit, MINIMUM_VALUE, 0 );
SetSlotOption( ItemVars:UpperLimit, MAXIMUM_VALUE, 20 );
ItemVars:UpperLimit = 100;
SetSlotOption( ItemVars:UpperLimit, IF_NEEDED, NULL );
SetSlotOption( ItemVars:UpperLimit, WHEN_ACCESS, NULL );
```

```
SetSlotOption( ItemVars:UpperLimit, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:UpperLimit, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:UpperLimit, IMAGE, Slider4 );
MakeSlot( ItemVars:SetPoint );
SetSlotOption( ItemVars:SetPoint, INHERIT, FALSE );
SetSlotOption( ItemVars:SetPoint, VALUE_TYPE, NUMBER );
SetSlotOption( ItemVars:SetPoint, MINIMUM_VALUE, 0 );
SetSlotOption( ItemVars:SetPoint, MAXIMUM_VALUE, 20 );
ItemVars:SetPoint = 50;
SetSlotOption( ItemVars:SetPoint, IF_NEEDED, NULL );
SetSlotOption( ItemVars:SetPoint, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:SetPoint, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:SetPoint, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:SetPoint, IMAGE, Slider3 );
MakeSlot( ItemVars:LowerLimit );
SetSlotOption( ItemVars:LowerLimit, INHERIT, FALSE );
SetSlotOption( ItemVars:LowerLimit, VALUE_TYPE, NUMBER );
SetSlotOption( ItemVars:LowerLimit, MINIMUM_VALUE, 0 );
SetSlotOption( ItemVars:LowerLimit, MAXIMUM_VALUE, 20 );
ItemVars:LowerLimit = 0;
SetSlotOption( ItemVars:LowerLimit, IF_NEEDED, NULL );
SetSlotOption( ItemVars:LowerLimit, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:LowerLimit, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:LowerLimit, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:LowerLimit, IMAGE, Slider2 );
  Slot( ItemVars:Ok );
  Option( ItemVars:Ok, INHERIT, FALSE );
SetSlotOption( ItemVars:Ok, VALUE_TYPE, BOOLEAN );
ItemVars:Ok = TRUE;
SetSlotOption( ItemVars:Ok, IF_NEEDED, NULL );
SetSlotOption( ItemVars:Ok, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:Ok, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:Ok, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:Ok, IMAGE, CheckBox4 );
MakeSlot( ItemVars:Owner );
ItemVars:Owner = TKP01;
MakeSlot( ItemVars:Choice );
ItemVars:Choice = Level;
MakeSlot( ItemVars:Value );
SetSlotOption( ItemVars:Value, INHERIT, FALSE );
ItemVars:Value = 0;
SetSlotOption( ItemVars:Value, IF_NEEDED, NULL );
SetSlotOption( ItemVars:Value, WHEN_ACCESS, NULL );
SetSlotOption( ItemVars:Value, BEFORE_CHANGE, NULL );
SetSlotOption( ItemVars:Value, AFTER_CHANGE, NULL );
SetSlotOption( ItemVars:Value, IMAGE, Edit14 );
MakeSlot( ItemVars:VType );
MakeSlot( ItemVars:VarNo );
ItemVars:VarNo = 3;
SetSlotOption( ItemVars:VarNo, IMAGE, Edit15 );

    /*************************************
```

```
      **** INSTANCE: ActionDisp
      ************************************/
MakeInstance( ActionDisp, Root );

/************** METHOD: Display  **************/
MakeMethod( ActionDisp, Display, [],
   {
   If Not( Null?( Self:CurrentChoice ) )
    Then {
        ClearTranscriptImage( Action );
        DisplayText( Action, FormatValue( "\n  " # Time( ) #
                                          "   " # Date( )
                                          # "          ACTION "
                                          # Self:CurrentChoice
                                          # "\n\n" ) );
        DisplayText( Action, FormatValue( "\n    " ) );
        DisplayText( Action, Self:CurrentChoice:Root:Message );
        DisplayText( Action, FormatValue( "\n   It is recommended that you - \n    " ) );
        DisplayText( Action, FormatValue( " \n    " ) );
        DisplayText( Action, Self:CurrentChoice:ActionMesg );
        DisplayText( Action, FormatValue( "\n" ) );
        If GetValue( {
                  Self:CurrentChoice;
                  }, Acknowledge )
         Then ( Self:Acknowledged = ACKNOWLEDGED )
         Else Self:Acknowledged = NULL;
        };
   } );

/************** METHOD: Init  **************/
MakeMethod( ActionDisp, Init, [],
   {
   ClearTranscriptImage( Action );
   ClearList( SingleListBox1, AllowableValues );
   ClearList( Self:Actions );
   Self:Acknowledged = NULL;
   Self:CurrentChoice = NULL;
   Self:ChoicePos = 0;
   } );

/************** METHOD: Reset  **************/
MakeMethod( ActionDisp, Reset, [],
   {
   If ( LengthList( Self:Actions ) == 0 )
    Then {
        Self:CurrentChoice = NULL;
        ClearTranscriptImage( Action );
        };
   } );
MakeSlot( ActionDisp:CurrentChoice );
SetSlotOption( ActionDisp:CurrentChoice, INHERIT, FALSE );
ActionDisp:CurrentChoice = NULL;
```

```
SetSlotOption( ActionDisp:CurrentChoice, IF_NEEDED, NULL );
SetSlotOption( ActionDisp:CurrentChoice, WHEN_ACCESS, NULL );
SetSlotOption( ActionDisp:CurrentChoice, BEFORE_CHANGE, NULL );
SetSlotOption( ActionDisp:CurrentChoice, AFTER_CHANGE, Display );
SetSlotOption( ActionDisp:CurrentChoice, IMAGE, SingleListBox1 );
MakeSlot( ActionDisp:Acknowledged );
SetSlotOption( ActionDisp:Acknowledged, INHERIT, FALSE );
ActionDisp:Acknowledged = NULL;
SetSlotOption( ActionDisp:Acknowledged, IF_NEEDED, NULL );
SetSlotOption( ActionDisp:Acknowledged, WHEN_ACCESS, NULL );
SetSlotOption( ActionDisp:Acknowledged, BEFORE_CHANGE, NULL );
SetSlotOption( ActionDisp:Acknowledged, AFTER_CHANGE, NULL );
SetSlotOption( ActionDisp:Acknowledged, IMAGE, StateBox1 );
MakeSlot( ActionDisp:Actions );
SetSlotOption( ActionDisp:Actions, INHERIT, FALSE );
SetSlotOption( ActionDisp:Actions, MULTIPLE );
SetSlotOption( ActionDisp:Actions, VALUE_TYPE, OBJECT );
SetSlotOption( ActionDisp:Actions, ALLOWABLE_CLASSES, Actions );
ClearList( ActionDisp:Actions );
SetSlotOption( ActionDisp:Actions, IF_NEEDED, NULL );
SetSlotOption( ActionDisp:Actions, WHEN_ACCESS, NULL );
SetSlotOption( ActionDisp:Actions, BEFORE_CHANGE, NULL );
SetSlotOption( ActionDisp:Actions, AFTER_CHANGE, Reset );
MakeSlot( ActionDisp:ChoicePos );
SetSlotOption( ActionDisp:ChoicePos, INHERIT, FALSE );
SetSlotOption( ActionDisp:ChoicePos, VALUE_TYPE, NUMBER );
ActionDisp:ChoicePos = 0;
SetSlotOption( ActionDisp:ChoicePos, IF_NEEDED, NULL );
SetSlotOption( ActionDisp:ChoicePos, WHEN_ACCESS, NULL );
SetSlotOption( ActionDisp:ChoicePos, BEFORE_CHANGE, NULL );
SetSlotOption( ActionDisp:ChoicePos, AFTER_CHANGE, NULL );
SetSlotOption( ActionDisp:ChoicePos, IMAGE, Edit10 );

    /*************************************
     **** INSTANCE: LinkEdit2
     *************************************/
MakeInstance( LinkEdit2, Menu );

/************ METHOD: Instance  ************/
MakeMethod( LinkEdit2, Instance, [owner window choice ],
  {
  Self:Check = TRUE;
  While  (Self:Check)
      {
      PostInputForm( " Ne . Instance " # owner, Self:Instance,
         "Instance name" );
      If ( Not( Null?( Self:Instance ) ) And ( Not( Instance?( Self:Instance ) )
                                          And MakeInstance( Self:Instance,
                                              owner ) ) )
        Then ( Self:Check = FALSE )
        Else If Not( Null?( Self:Instance ) )
             Then PostMessage( Self:Instance # " already exists!" );
```

```
        If Null?( Self:Instance )
          Then Self:Check = FALSE;
        };
    } );

/************** METHOD: SubClass  **************/
MakeMethod( LinkEdit2, SubClass, [owner window choice ],
    {
    Self:Check = TRUE;
    While  (Self:Check)
        {
        PostInputForm( " New SubClass " # owner, Self:SubClass,
            "SubClass name" );
        If ( Not( Class?( Self:SubClass ) ) And MakeClass( Self:SubClass,
                                                owner ) )
          Then Self:Check = FALSE;
        If Null?( Self:SubClass )
          Then Self:Check = FALSE;
        };
    } );

/************** METHOD: Delete  **************/
MakeMethod( LinkEdit2, Delete, [owner window choice ],
    {
    If ( CountInstances( owner ) == 0 )
     Then {
          If ( PostMenu( " Delete " # owner # " ?", YES, NO )
                #= YES )
           Then DeleteClass( owner );
          }
     Else PostMessage( "ERROR - Erase instances of class first" );
    } );

/************** METHOD: Rename  **************/
MakeMethod( LinkEdit2, Rename, [owner window choice ],
    {
    Self:Check = TRUE;
    While  (Self:Check)
        {
        Self:name = owner;
        PostInputForm( " Rename Class " # owner, Self:name, "New Class name" );
        If ( Not( Class?( Self:name ) ) And Not( Null?( Self:name ) )
              And RenameClass( owner, Self:name ) )
          Then Self:Check = FALSE;
        If ( Null?( Self:name ) Or Self:name #= owner )
          Then Self:Check = FALSE;
        If Self:Check
          Then PostMessage( "ERROR -  Instance already exists" );
        };
    } );

/************** METHOD: InitMenu  **************/
```

```
MakeMethod( LinkEdit2, InitMenu, [owner ],
   {
   If ( owner #= Links )
     Then AppendToList( Self:Disabled, Rename, Delete, Edit, Move )
     Else ClearList( Self:Disabled );
   } );
LinkEdit2:X = 46;
LinkEdit2:Y = 120;
SetValue( LinkEdit2:Choices, Instance );
SetSlotOption( LinkEdit2:ChoiceNames, NO_AUTO_ASK, FALSE );
SetValue( LinkEdit2:ChoiceNames, "Add &Instance" );
ClearList( LinkEdit2:Disabled );
MakeSlot( LinkEdit2:Check );
SetSlotOption( LinkEdit2:Check, INHERIT, FALSE );
SetSlotOption( LinkEdit2:Check, NO_AUTO_ASK, TRUE );
SetSlotOption( LinkEdit2:Check, VALUE_TYPE, BOOLEAN );
LinkEdit2:Check = FALSE;
SetSlotOption( LinkEdit2:Check, IF_NEEDED, NULL );
SetSlotOption( LinkEdit2:Check, WHEN_ACCESS, NULL );
SetSlotOption( LinkEdit2:Check, BEFORE_CHANGE, NULL );
SetSlotOption( LinkEdit2:Check, AFTER_CHANGE, NULL );
MakeSlot( LinkEdit2:Instance );
SetSlotOption( LinkEdit2:Instance, INHERIT, FALSE );
SetSlotOption( LinkEdit2:Instance, IF_NEEDED, NULL );
SetSlotOption( LinkEdit2:Instance, WHEN_ACCESS, NULL );
SetSlotOption( LinkEdit2:Instance, BEFORE_CHANGE, NULL );
SetSlotOption( LinkEdit2:Instance, AFTER_CHANGE, NULL );
MakeSlot( LinkEdit2:SubClass );
LinkEdit2:SubClass = Hello;
MakeSlot( LinkEdit2:name );
LinkEdit2:name = A_TKP01;

     /*************************************
      **** INSTANCE: Simulation
      *************************************/
MakeInstance( Simulation, Root );

/************** METHOD: Evaluate  **************/
MakeMethod( Simulation, Evaluate, [],
   {
   Self:D_Time = GetTime( ) - Self:Last_Time;
   Self:Last_Time = GetTime( );
   EnumList( Self:Evaluate, X, SetValue( Self, X, Self:X ) );
   } );

/************** METHOD: LIP002  **************/
MakeMethod( Simulation, LIP002, [Slot Level ],
   {
   ( Level / 100 * 12000 + ( Self:D_Time * ( P01:Flow - P02:Flow ) ) )
     / 120;
   } );
MakeSlot( Simulation:Evaluate );
```

```
SetSlotOption( Simulation:Evaluate, INHERIT, FALSE );
SetSlotOption( Simulation:Evaluate, MULTIPLE );
SetValue( Simulation:Evaluate, LIP002 );
SetSlotOption( Simulation:Evaluate, IF_NEEDED, NULL );
SetSlotOption( Simulation:Evaluate, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:Evaluate, BEFORE_CHANGE, NULL );
SetSlotOption( Simulation:Evaluate, AFTER_CHANGE, NULL );
MakeSlot( Simulation:LIP002 );
SetSlotOption( Simulation:LIP002, INHERIT, FALSE );
SetSlotOption( Simulation:LIP002, VALUE_TYPE, NUMBER );
Simulation:LIP002 = 13.7306694674992;
SetSlotOption( Simulation:LIP002, IF_NEEDED, NULL );
SetSlotOption( Simulation:LIP002, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:LIP002, BEFORE_CHANGE, LIP002 );
SetSlotOption( Simulation:LIP002, AFTER_CHANGE, NULL );
MakeSlot( Simulation:Last_Time );
SetSlotOption( Simulation:Last_Time, INHERIT, FALSE );
SetSlotOption( Simulation:Last_Time, VALUE_TYPE, NUMBER );
Simulation:Last_Time = 338710.608791;
SetSlotOption( Simulation:Last_Time, IF_NEEDED, NULL );
SetSlotOption( Simulation:Last_Time, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:Last_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Simulation:Last_Time, AFTER_CHANGE, NULL );
MakeSlot( Simulation:D_Time );
SetSlotOption( Simulation:D_Time, INHERIT, FALSE );
SetSlotOption( Simulation:D_Time, VALUE_TYPE, NUMBER );
Simulation:D_Time = 60;
SetSlotOption( Simulation:D_Time, IF_NEEDED, NULL );
SetSlotOption( Simulation:D_Time, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:D_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Simulation:D_Time, AFTER_CHANGE, NULL );

    /*****************************************
     *** INSTANCE: HalEx
     *****************************************/
MakeInstance( HalEx, Transcript );
HalEx:SessionNumber = 0;
HalEx:Title = "HAL Expert - Explanation Window";
SetValue( HalEx:ForegroundColor, 0, 0, 0 );
SetValue( HalEx:BackgroundColor, 0, 255, 255 );
SetValue( HalEx:ForegroundColor2, 0, 0, 0 );
SetValue( HalEx:BackgroundColor2, 255, 255, 255 );
HalEx:Width = 462;
HalEx:Height = 247;
HalEx:Visible = TRUE;
HalEx:X = 70;
HalEx:Y = 11;
HalEx:VertScroll = TRUE;
HalEx:HorzScroll = TRUE;
HalEx:Font = Helv;
HalEx:TextSize = 8;
HalEx:Bold = FALSE;
```

```
SetSlotOption( Simulation:Evaluate, INHERIT, FALSE );
SetSlotOption( Simulation:Evaluate, MULTIPLE );
SetValue( Simulation::Evaluate, LIP002 );
SetSlotOption( Simulation:Evaluate, IF_NEEDED, NULL );
SetSlotOption( Simulation:Evaluate, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:Evaluate, BEFORE_CHANGE, NULL );
SetSlotOption( Simulation:Evaluate, AFTER_CHANGE, NULL );
MakeSlot( Simulation.LIP002 );
SetSlotOption( Simulation:LIP002, INHERIT, FALSE );
SetSlotOption( Simulation:LIP002, VALUE_TYPE, NUMBER );
Simulation:LIP002 = 13.7306694674992;
SetSlotOption( Simulation:LIP002, IF_NEEDED, NULL );
SetSlotOption( Simulation:LIP002, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:LIP002, BEFORE_CHANGE, LIP002 );
SetSlotOption( Simulation:LIP002, AFTER_CHANGE, NULL ):
MakeSlot( Simulation:Last_Time );
SetSlotOption( Simulation:Last_Time, INHERIT, FALSE );
SetSlotOption( Simulation:Last_Time, VALUE_TYPE, NUMBER );
Simulation:Last_Time = 338710.608791;
SetSlotOption( Simulation:Last_Time, IF_NEEDED, NULL );
SetSlotOption( Simulation:Last_Time, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:Last_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Simulation:Last_Time, AFTER_CHANGE, NULL );
MakeSlot( Simulation:D_Time );
SetSlotOption( Simulation:D_Time, INHERIT, FALSE );
SetSlotOption( Simulation:D_Time, VALUE_TYPE, NUMBER );
Simulation:D_Time = 60;
SetSlotOption( Simulation:D_Time, IF_NEEDED, NULL );
SetSlotOption( Simulation:D_Time, WHEN_ACCESS, NULL );
SetSlotOption( Simulation:D_Time, BEFORE_CHANGE, NULL );
SetSlotOption( Simulation:D_Time, AFTER_CHANGE, NULL );

    /*********************************
     **** INSTANCE: HalEx
     *********************************/
MakeInstance( HalEx, Transcript );
HalEx:SessionNumber = 0;
HalEx:Title = "HAL Expert - Explanation Window";
SetValue( HalEx:ForegroundColor, 0, 0, 0 );
SetValue( HalEx:BackgroundColor, 0, 255, 255 );
SetValue( HalEx:ForegroundColor2, 0, 0, 0 );
SetValue( HalEx:BackgroundColor2, 255, 255, 255 );
HalEx:Width = 462;
HalEx:Height = 247;
HalEx:Visible = TRUE;
HalEx:X = 70;
HalEx:Y = 11;
HalEx:VertScroll = TRUE;
HalEx:HorzScroll = TRUE;
HalEx:Font = Helv;
HalEx:TextSize = 8;
HalEx:Bold = FALSE;
```

```
HalEx:Underline = FALSE;
HalEx:Italic = FALSE;
HalEx:StrikeOut = FALSE;
HalEx:VectorFont = CVP02_ScrewCon;
HalEx:Font2 = Helv;
HalEx:TextSize2 = 8;
HalEx:Bold2 = TRUE;
HalEx:Underline2 = FALSE;
HalEx:Italic2 = FALSE;
HalEx:StrikeOut2 = FALSE;
HalEx:TextVector2 = CVP02_ScrewCon;
 ResetImage ( HalEx );

      /*************************************
       **** INSTANCE: Error
       *************************************/
MakeInstance( Error, Transcript );
Error:SessionNumber = 0;
Error:Title = ERRORS;
SetValue( Error:ForegroundColor, 0, 0, 0 );
SetValue( Error:BackgroundColor, 255, 0, 0 );
SetValue( Error:ForegroundColor2, 0, 0, 0 );
SetValue( Error:BackgroundColor2, 255, 255, 255 );
Error:Width = 431;
Error:Height = 94;
Error:Visible = TRUE;
Error:X = 96;
Error:Y = 336;
Error:Font = Helv;
Error:TextSize = 8;
Error:Bold = TRUE;
Error:Underline = FALSE;
Error:Italic = FALSE;
Error:StrikeOut = FALSE;
Error:VectorFont = CVP02_ScrewCon;
Error:VertScroll = AUTO;
Error:HorzScroll = AUTO;
Error:Font2 = Helv;
Error:TextSize2 = 8;
Error:Bold2 = TRUE;
Error:Underline2 = FALSE;
Error:Italic2 = FALSE;
Error:StrikeOut2 = FALSE;
Error:TextVector2 = CVP02_ScrewCon;
 ResetImage ( Error );

      /*************************************
       **** INSTANCE: Action
       *************************************/
MakeInstance( Action, Transcript );
Action:SessionNumber = 3;
Action:Title = "Action description";
```