AN ABSTRACT OF THE THESIS OF

Thomas M. Scott for the degree of Master of Science in Industrial Engineering presented on March 14, 1991. Title: Design of an Object-Oriented Paradigm for Model Generation: An Application in Timber Harvesting

Abstract approved: Redacted for Privacy

[illegible]

The potential gains that could be realized from optimizing timber harvesting are significant. To a great extent it is these initial functions that dictate the quality of future manufacturing steps. Timber harvesting systems are defined by the operations they contain and the equipment that perform them. For any given harvesting situation there are a large number of unique systems, performing under a variety of cost, production, and environmental conditions. The objective of optimized timber harvesting is to reduce cost while simultaneously increasing production and minimizing environmental impact. One way to accomplish this objective is through mechanization and automation.

Mechanization increases the production output,

efficiency, and product quality. However selecting an appropriate level of mechanization to avoid under utilization of expensive resources is a critical decision. The decision requires that the product mix, environmental and user specified constraints be matched against the available equipment technology, and the required performance criteria.

This research describes a computer based system which queries a user on the timber stand specifics and a set of harvesting objectives. The system then matches these user's needs to a level of mechanization that would maximize the efficiency of the production system. The computer accomplishes this by searching a set of databases containing information on the available technology and its impact on production, efficiency, economics and the environment. The level of mechanization is determined by specific combinations of existing equipment. Individual pieces of equipment that are compatible with one another are balanced together to form a viable productive unit.

Design of an Object-Oriented Paradigm for Model Generation: An Application in Timber Harvesting

by

Thomas M. Scott

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 14, 1991

Commencement June 1991

APPROVED:

Redacted for Privacy

Professor of Industrial and Manufacturing Engineering in charge of Major

Redacted for Privacy

Head of department of Industrial and Manufacturing Engineering

Redacted for Privacy

Dean of Graduate School

Date Thesis is presented March 14, 1991

Typed by Thomas M. Scott for Thomas M. Scott

ACKNOWLEDGEMENTS

I would like to thank my major professor Dr. Sabah Randhawa. His continued support and encouragement were necessary for the project's success. I would like to thank Dr. Ken Funck and Dr. Eldon Olsen for their input and support during the project. I would also like to thank Tony Chou for helping with some of the difficult points in the software development.

I would especially like to thank my family who were always there when I needed them. My two children Jeffery and Dana showered me with love and understanding throughout this project. My good friend Rudy Hansen was always ready with words of encouragement and I thank him for that.

My deepest gratitude however goes to my Lord Jesus Christ who never let me lose sight of my priorities. From Luke 9:25 " For what is a man profited if he gains the whole world and loses or forfeits himself? ".

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Design of an Object-Oriented Paradigm for Model Generation: An Application in Timber Harvesting

CHAPTER 1
INTRODUCTION

Timber harvesting is an area of considerable importance, particularly in the Pacific Northwest. In recent years, significant changes have occurred in the log manufacturing and harvesting environment. There have been two important reasons for these changes. First, public awareness concerning forest issues has increased dramatically in recent years. Protection of the environment and enhancement of the scenic values are factors that must be considered when making harvesting related decisions.

Second and perhaps more important, there has been an increase in the proportion of merchantable second growth forests now being harvested. This has greatly affected equipment selection as well as the overall logging practices. Typically the smaller stem sizes found in second growth forests have increased the opportunity for the use of material handling systems and automatic processors. The choice of equipment for any phase of the timber harvesting process is a function of multiple factors including tract geometry, topography, plot size and concentration, tree stand characteristics including species distribution and log size distribution.

Modeling and evaluation of timber harvesting systems can be done using analytical modeling. However, mathematical analysis of such a complex process is extremely difficult to conduct. Simulation is the most common technique used for analyzing complex systems such as timber harvesting. Scenario or Model formulation is recognized as the most difficult phase in building a simulation model, yet all simulation techniques start with the premise that this phase is correct and complete. Typical simulation analysis efforts consist of three cyclic phases (Law 1982), beginning with defining the system to be modeled. From this definition the system is translated into a simulation model program. The program is fed data, output is then generated and analyzed. Current simulation applications primarily focus on the second of these three phases, that is, given an alternative to be simulated, translate it into a simulation model. Some effort has been made in developing output model analyzers. However little if any effort has gone into the first phase, that of model or scenario generation. Paralleling the trends in simulation technology, there have been a number of simulation models that have been developed for analyzing different facets of log harvesting, but very little effort has been expending on timber harvesting equipment selection itself. The system described in this research is specifically designed to generate modeling scenarios for timber harvesting. The scenarios generated represent the sequence of operations required for a given harvesting environment. Each operation is represented by a piece of equipment designated to perform that operation from a database of available technology. The harvesting scenarios give specific information about equipment compatibility and overall costs. The systems identified as feasible harvesting alternatives can then be analyzed using a simulation model for equipment balancing and other detailed performance measures. One such model is LOGSIM (Randhawa and Olsen; 1990). LOGSIM is a computer-based simulation system for modeling the entire harvesting system from the felling operation until the log arrives at a conversion facility.

Several researchers have suggested the potential for the use of artificial intelligence (AI) techniques in simulation and decision support. These include Oren, 1977; Oren and Ziegler, 1979: Goldin and Klahr, 1981; Gory and Krumland, 1983; Shannon, 1984; Calu, Wael, Esmeijer, Kerckhoffs and Vansteenkiste, 1984; Shannon, Mayer and Adelsberger, 1985; Ruiz-Meir, Talvage and Ben-Arieh, 1985 and Kerchkhoffs and Vansteenkiste, 1986. The research in this area has progressed in two directions : provide model development capabilities that would allow the user to easily develop and modify representations of systems to be simulated, and to increase the understanding of the system being studied through better interpretation of simulation output.

The focus of this research is to develop a methodology for the automatic generation of simulation models or scenarios. This is accomplished by taking user input used to describe a particular harvesting environment, and mapping the user environment to a database of available equipment technology (Figure 1.1). The mapping process employs an artificial intelligence technique known as a best first search algorithm which identifies the feasible harvesting alternatives.

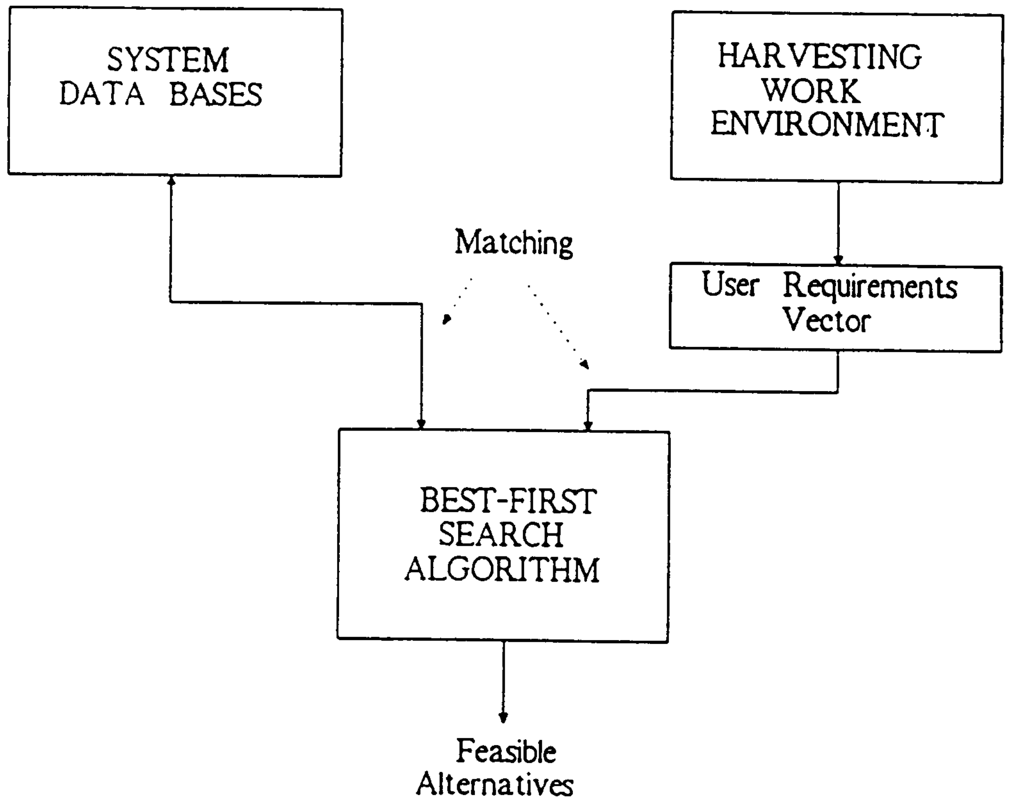


Figure 1.1 The Mapping Procedure

Problem Domain

Selection of the appropriate harvesting equipment is determined by the interaction of a multitude of factors. The primary factors may be categorized into two major categories. The first category is the specific harvesting environment and the second category is the database of available equipment. The database of available equipment must include well defined attributes used to determine site and equipment compatibility. Once these two sets of data are assembled one needs to use a systematic procedure to determine the combinations of equipment that satisfy all harvesting requirements while producing the required finished product. The actual problem domain is slightly more complicated since the compatibility between pieces of equipment must also be considered.

General Approach

A work environment is created by the user to produce an N-dimension vector space representing the timber stand environment. The equipment database is created by a two step process, initiated with the creation of a harvesting operation and then the creation of appropriate equipment for the given operation. The search process uses a best first search algorithm driven by harvesting cost.

The initial state of the system is standing timber with the first operation always defined as the start operation.

Therefore the search process begins by mapping the specific user's environment to the equipment filed under the start operation. The start operation contains one machine which is a null machine whose only purpose is to define the user selected starting operations. For example a user might want the operations felling, felling plus bunching and felling plus delimbing to be the three starting operations, so the null machine would contain this information. Basically the start operation and the null machine define the initial state of the system. Each starting operation is explored for machines that pass the requirement vector, those that pass are put in a queue and designated as unexpanded systems.

Each piece of equipment contains a variable which indicates the operations that can follow the current operation. The search algorithm will select the lowest cost equipment from the starting operations. This lowest cost equipment becomes a starting system. The search algorithm and the mapping procedures expand the system by selecting pieces of equipment in the starting system's feasible successor operations to come up with feasible extended systems. These feasible extended systems are then added to the queue of unexpanded systems. This queue of unexpanded systems is then sorted to identify the lowest cost system to expand next. This search process is continued until a suitable system is identified based on the following criteria.

1.) The system's last machine produces the goal product.

2.) The system's last machine produces the goal location.

3.) The system becomes the lowest cost system in queue.

In words the search process is searching for the system that produces the user defined product at the user defined location. When a given state is expanded the new expanded systems are put into the queue and the queue is sorted by cost. Therefore a system may be expanded to produce the goal product at the goal location, but due to cost remain buried in the queue of unexpanded systems. The first system that meets both goal states and becomes the lowest cost system is identified as a suitable system.

CHAPTER 2

METHODOLOGY

Timber harvesting is traditionally a sequential process where a stand of timber is cut and processed to deliver a specific product to a mill for further processing. A harvesting system is comprised of components that are interrelated and jointly contribute to a common objective. The sequential nature of timber harvesting suggests that certain steps must be performed in a given order to achieve the objective. However, the order of these operations can vary greatly from system to system (Conway 1982).

No matter what harvesting system is being considered they all have certain characteristics in common, These include:

1. Cutting the tree, usually a felling operation.
2. Skidding or Yarding, usually referred to as primary transportation.
3. Loading the trees on vehicles, referred to as the loading operation.
4. Log or tree transportation, usually referred to secondary transportation.

These four components represent a macro starting point from which the more sophisticated harvesting systems are built. In today's highly mechanized industry a single machine often performs several operations. Harvesting systems vary depending on a large number of external factors such as stand geometry, terrain and the form and location of the primary product objective. In this research, these external factors are represented in the user's vector.

Environment Creation

The TIMBER HARVESTER is a completely menu driven system that allows a user to easily create, edit and display the current harvesting environment. The users environment is broken down into three components, Site Variables , Stand Variables and Harvesting Requirements.

Site Variables

__GroundFirmness__

The firmness of the ground, defined by using a discrete 1-5 scale; 1 defining soft ground and 5 defining a hard ground firmness.

__GroundRoughness__

The roughness of the ground, defined on a discrete scale 1-5; 1 defining mild ground roughness and 5 defining a severe ground roughness.

__Slope__

The average slope on the given site.

__HaulDistance__

The average transportation distance from the woods to loading location.

Stand Variables

<u>Acres</u>

The size of the given timber stand in (acres).

<u>Dbh</u>

The average diameter at breast height of the major species in the stand.

<u>Merchantable Trees</u>

The number of merchantable trees per acre.

<u>Species</u>

The major species on the given site.

<u>Tree Height</u>

The average tree height (in feet) for a stand.

<u>Unmerchantable Trees</u>

The number of unmerchantable trees per acre.

Requirement Variables

<u>Shear Felling</u>

This variable determines if a shear attachment may be used during felling.

<u>Suspension</u>

This variable determines if one end of the log or tree must be suspended during transport.

<u>Product</u>

The required finished product.

Once the users environment is created it defines an N - dimension vector space representing the particular timber stand. The user vector variables are continually accessed during program execution to map the equipment to the specific harvesting environment.

Database Creation and Selection

The database of available equipment is selected from a larger database of equipment technology. The larger database contains state of the art harvesting systems as well as the older systems. The reason for the two separate databases is so a given user may select only the equipment that are pertinent to their operation. Both databases have database functions for addition, deletion and display. The difference is that additions to the database of available equipment are selected from the database of equipment technology. This means that only those pieces of equipment contained in the database of equipment technology can be added to the database of available equipment.

Both databases are arranged by harvesting operations, where equipment designated to perform a given operation is filed under that operation. The TIMBER HARVESTER allows for the creation and deletion of harvesting operations at any time. This data representation scheme requires a concise definition of an operation.

DEFINITION: An operation is a procedure that takes an input material form at a particular location and transforms the material into an output form at a particular location. The output form must be consistent with the operation for which a given piece of equipment is assigned.

For a felling operation the input form would be standing timber and the output form would be a complete tree. It would obviously be inconsistent to have a piece of equipment categorized for the felling operation performing delimbing, because the output form would be a delimbed tree. Therefore all delimbing must be performed by equipment assigned to that operation. Some equipment will have an output form that is the same as the input form and the only difference will be the change in location (eg skidding).

Once an operation is created, equipment can then be added under that operation to produce that operation's output form at a location. After the equipment is given a name and added to the database of equipment technology, its attributes must be assigned that define the operating specifics about a given piece of equipment. Assigning attributes to a given piece of equipment is nothing more that giving it a unique identity. These attributes are accessed during program execution to calculate production costs and to determine suitability of the equipment with the current user environment. The equipment attributes in the system databases include:

<u>Accumulation</u>

This attribute is described in two parts. The input accumulation refers to the accumulation state that a given piece of equipment can accept. Given a particular type of input accumulation a given machine will produce an output accumulation state which may or may not be the same as the input accumulation. For example a feller buncher's accumulation state is bunched or decked while manual felling results in a random accumulation state.

<u>Attachment</u>

The attachment attribute refers to a prominent attachment such as a shear attachment for a feller buncher or a grapple attachment on a skidder. This attribute variable is referenced during program execution to calculate production costs.

<u>CostPMH</u>

The cost of the operation per machine hour is an entered value that is independent of the user environment. This value must be known or estimated by the user.

<u>Dbh</u>

The dbh is described in two parts. The lowerbound value represents a piece of equipment's lower bound operating range. The upperBound value represents the upper bound operating range.

<u>Ground</u>

This attribute is described in two parts. The ground roughness refers to general topography and rocks etc. The ground firmness refers to soil texture. A scaling factor from 1 to 5 is used to define the two ranges. A factor of one being mild or soft while a factor of 5 being severe or hard.

<u>HPClass</u>

The horse power class is a user entered value describing the horse power class of a given piece of equipment.

Product

The product is described in two parts. The first part is termed the input and refers to an input product. This input product refers to the output product from a piece of equipment that performed the previous operation. Given a particular input form a given piece of equipment will produce an output product.

Location

The location attribute refers to the location where the output product is located after processing. This may or may not be the same location as before processing. An example would be a bucking operation where the output location is the same after the bucking operation as before. With a skidding operation the location changes but the product state remains constant.

MaintCost

The maintenance cost per machine hour is a user defined value. The value is entered in a prompter window during attribute assignment.

Operations

This attribute is one of the more important attributes, and can have significant impact on the solution space. A piece of equipment assigned to performing a given operation needs to know what possible operations can follow the operation it performs. An example would be manual felling, at the completion of the felling operation there most probably will be more than one possible way to proceed with harvesting.

Case 1: We might bunch the trees before skidding.

Case 2: We might skid whole trees by themselves.

Case 3: We might delimb the trees.

The following operations would then be included in the attribute operations.

Case 1: bunching.

Case 2: skidding.

Case 3: delimbing.

These operations are all considered valid moves from the operation felling. It is important to realize that this attribute determines how systems can be constructed. It is equally important to realize that the user is in complete control of this attribute. This feature gives the user the flexibility to add operations and equipment freely.

ProdCost

This attribute is a default variable that is entered as a constant when no regression/production equation is defined. It is a constant value that is not affected by the user environment. This feature allows the incorporation of equipment for which sufficient data is not available.

ProdEquation

This attribute defines the name of a production equation. The production equation name is the name of a method in the class machine. The machine class may have many production methods so prodEquation identifies the specific equation to be used by a given piece of equipment. During program execution this equation will be utilized to provide an environmentally matched production cost for a given piece of equipment.

Slope

The slope attribute is a two part variable. The lowerBound defines the lowest slope for which a given piece of equipment can operate. The upperBound defines the largest positive slope for which the equipment can operate. Therefore the slope attribute defines the operating range for slope.

Suspension

This attribute is used to reflect a piece of equipment's ability to suspend one end during transport. This is a binary variable.

Search Process

The TIMBER HARVESTER uses an artificial intelligence technique known as a best first search (Figure 2.1). A best first search is a way to combine the advantages of both depth first and breadth first search into a single method.

At each step of the best first search process we select the most promising system generated thus far. This most promising system is then expanded to include the feasible moves from the current state of the system.

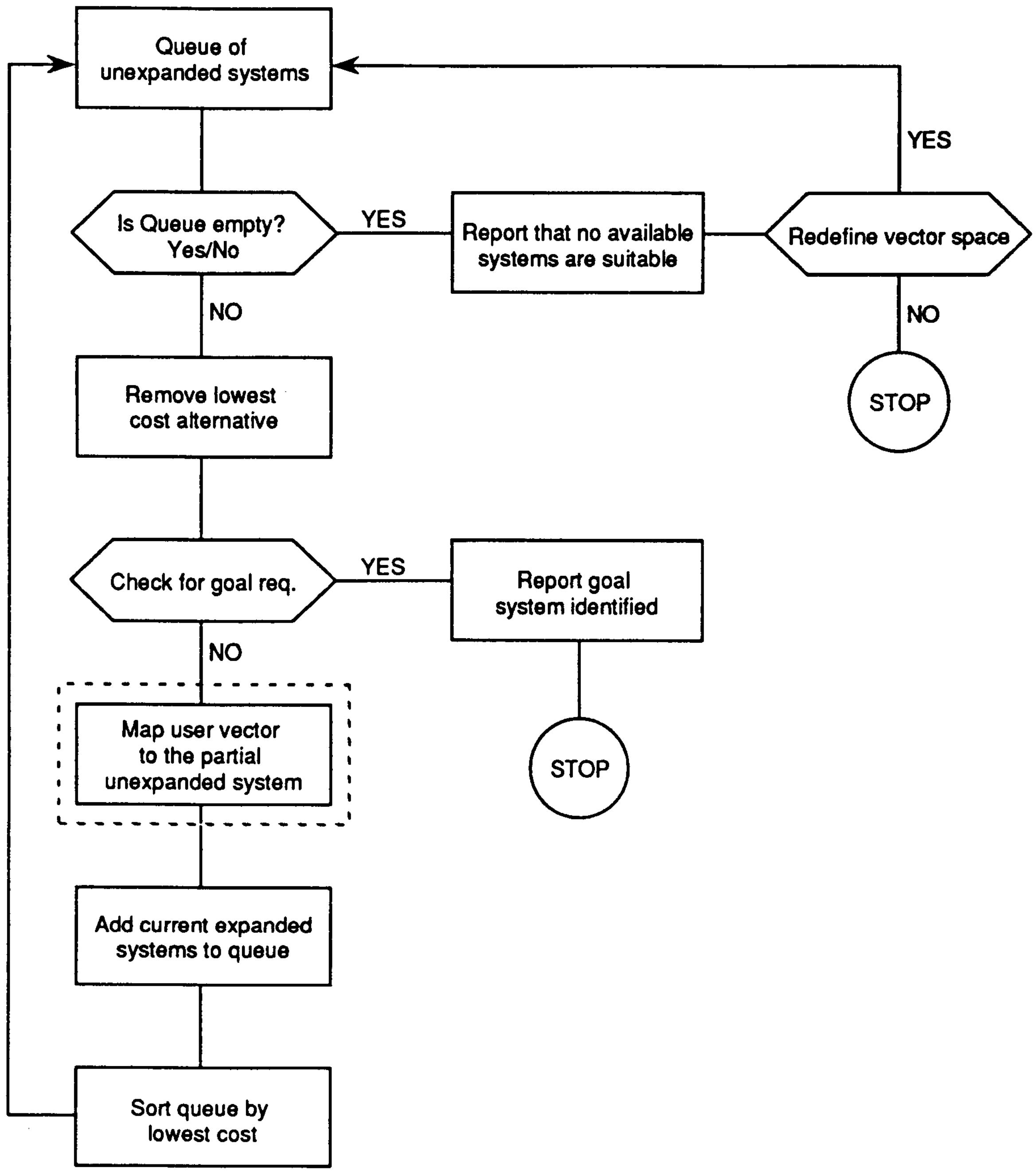

Queue of unexpanded systems
Is Queue empty? Yes/No
YES
Report that no available systems are suitable
Redefine vector space
YES
NO
STOP
NO
Remove lowest cost alternative
Check for goal req.
YES
Report goal system identified
STOP
NO
Map user vector to the partial unexpanded system
Add current expanded systems to queue
Sort queue by lowest cost

Figure 2.1 Search Procedure.

If any of the feasible moves generate a solution and still remains the most promising system, it is declared a suitable harvesting system for the current user's environment. The feasible moves from any state of the system is determined from a combination of factors during the mapping procedures employed by the search algorithm.

The system's initial state is defined as standing timber and all harvesting operations begin with the start operation. At the beginning of the search routine a null piece of equipment defines the user defined starting operations. The equipment in each starting operation are then mapped to the user's environment and the feasible pieces of equipment become the starting set of unexpanded systems.

Equipment that is assigned to a particular operation needs to know what operations can follow the one it performs. This key piece of information focuses the mapping routines to map only a subset of the database, thereby greatly improving program execution time. Once we know what operations can follow the current state of the system, we map each piece of equipment categorized in those selected operations and generate a collection of feasible moves from the current state.

Mapping Procedures

The mapping procedures are embedded in the search process and are used to generate the feasible moves from a partial state (Figure 2.2).

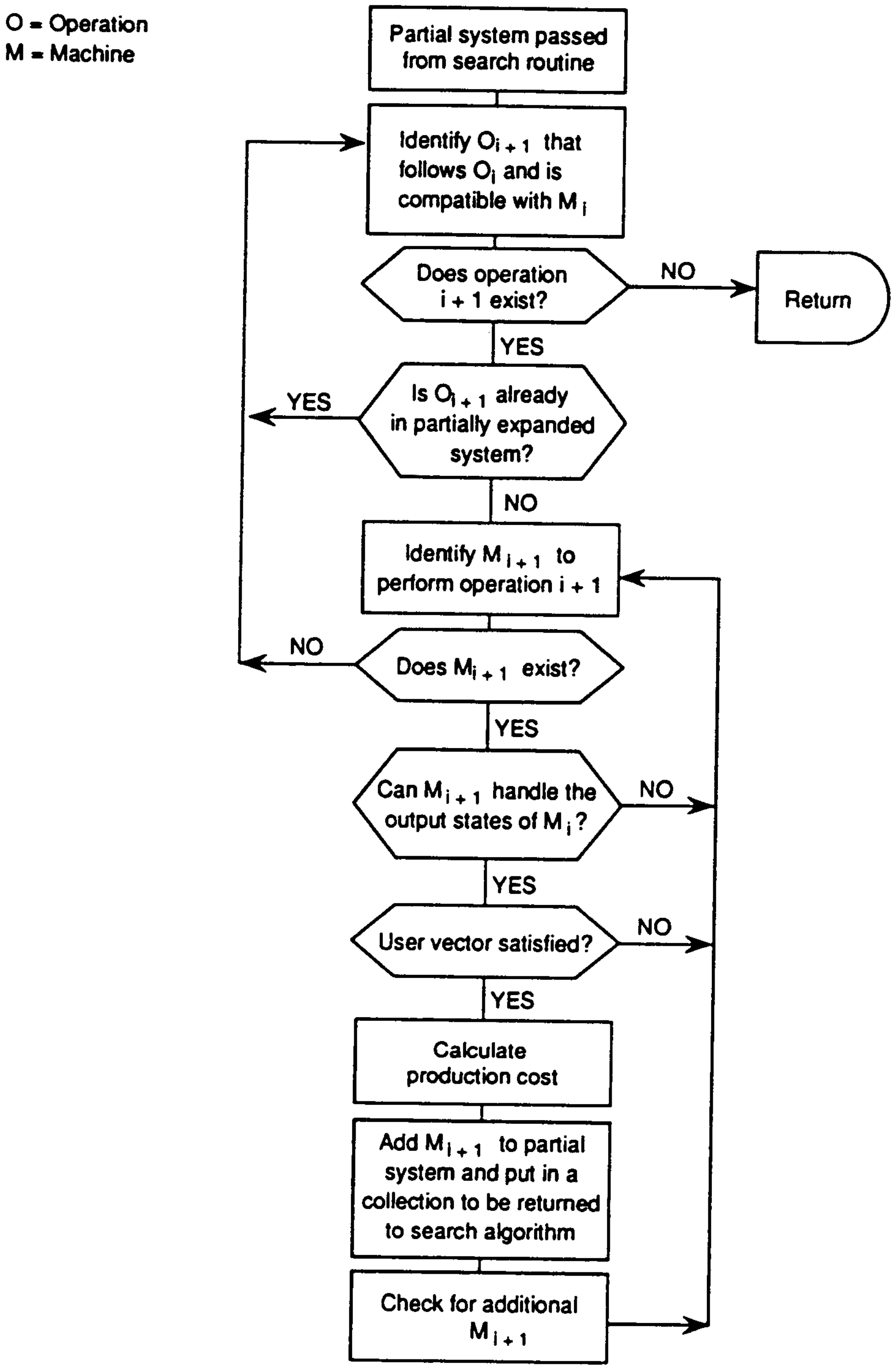

O = Operation
M = Machine
Partial system passed from search routine
Identify O_{i+1} that follows O_i and is compatible with M_i
Does operation i + 1 exist?
NO
Return
YES
Is O_{i+1} already in partially expanded system?
YES
NO
Identify M_{i+1} to perform operation i + 1
Does M_{i+1} exist?
NO
YES
Can M_{i+1} handle the output states of M_i?
NO
YES
User vector satisfied?
NO
YES
Calculate production cost
Add M_{i+1} to partial system and put in a collection to be returned to search algorithm
Check for additional M_{i+1}

Figure 2.2 Mapping Procedure.

There are three critical maps that must be satisfied for a piece of equipment to be selected as a feasible extension of the current system state. The following is a list of the critical maps.

1. Does the partial system already contain the candidate operation.
2. Can the candidate equipment for the next operation process the output from the previous stage.
3. Can the candidate equipment operate on the current user's environment.

Each one of these critical maps will now be discussed.

Does the partial system already contain the candidate operation.

The Timber Harvester is designed to be very flexible, allowing the user to define systems with any order of operations. For example trees can be bucked into logs prior to skidding or after the skidding operation. Regardless of when the trees are bucked the bucking operation is performed only once. Therefore it is necessary to map the candidate equipment to the current system to determine if the system has already performed that operation. If the system does contain the operation for which the candidate equipment is designated to perform, then the candidate equipment is removed from consideration.

Can candidate equipment process the current material state.

Each piece of equipment knows what material state it can accept and given a specific input material form at a specific location, each piece of equipment will generate a specific output material form. The three variables that define the material state are accumulation, product and location. These variables are all collections of input/output form. That is for a given input the equipment will generate a given output. Timber harvesting systems are by nature a sequential process where a predecessor equipment's output product will become the successor equipments input product.

If any one of these three variables cannot map the predecessors output form to the successors (candidate) equipments input forms then the equipment is discarded as incompatible equipment. This process of mapping the three material form variables defines the equipment compatibility.

Can candidate equipment operate on the current user's environment.

Each piece of equipment has its operating ranges which define its operational limitations. These operating ranges and other critical machine attributes must be mapped to the user's environment. If the specific harvesting environment falls within the equipment's operational limitations the equipment remains a candidate for system expansion.

If all three critical maps are satisfied the equipment production cost is calculated by using a production regression equation. The production equation provides an accurate assessment of the production cost for a given piece of equipment, on the user specified site. If a production equation is not available an estimated production cost will be used. Each successful candidate equipment is then added to the partial system.

The resulting systems are then added to the collection of unexpanded systems. The collection or queue of unexpanded systems is then sorted by least cost and the process is repeated. The system stops when the lowest cost system ends at the required location with the required product.

CHAPTER 3

COMPUTER MODEL

To achieve the desired degree of abstraction and reusability, the object-oriented paradigm was selected. The object-orientated framework provided a viable representation for the problem domain, by allowing the problem to be decomposed into logical modules (objects), containing both data and procedures. The objects represent real world entities, but more specifically represent partitioned areas of computer memory allocated to a specific object's data and procedures (Tello, 1989). Therefore, these objects can act as encapsulated entities by interacting with other objects in the system by passing messages between one another.

Object-oriented programming is defined in its purest sense as programming implemented by sending messages to objects (Pinson and Wiener 1988). Smalltalk/V, (SMALLTALK/V Tutorial and Programming Handbook, 1986) was the specific software tool selected, because of its interactive programming environment and its strict compliance to the four basic object properties of data abstraction, data encapsulation, inheritance and polymorphism. These properties will now be discussed in terms of their relationship to this research.

Data Abstraction

Object-oriented code by its very definition links data with procedures. In procedural programming parameters (data) is sent to procedures that act on the data in a predetermined way. In procedural programming the data is passive and the procedures are active. In the object-oriented paradigm the data and the procedures are linked together producing a form of active data (Tello 1989). The significant result of this linkage between data and procedures is data abstraction. In the most general sense, an abstraction is a concise representation for a more complicated idea or object. Details of the implementation of an abstraction are not essential to an understanding of its purpose and functionality. Thus using and understanding an abstraction enables higher-level human functioning than is possible if one has always to focus on the details (Pinson and Wiener 1988). The behavior of an abstract data object is defined by abstract operations which are called methods.

Methods are to object-orientated programming as procedures are to procedural programming. The real relevance of data abstraction is that it relieves the user of an object from having to know the details of invoking the object's methods. This release allows a programmer to become the user of an object instead of the programmer of an object.

This concept is the key to the reusablilty of object-oriented code. This attribute is exploited during database creation by using one object class to represent many different kinds of harvesting equipment. Therefore once this object is created a programmer has only to know the names of the messages to be able to use the object in it's full functionality.

Data Encapsulation

Data encapsulation restricts the affects of change by restricting all access to an objects data to messages that invoke that particular object's methods. The result of encapsulation is the minimization of inadvertent changes due to interdependencies between objects. This attribute of the object-oriented paradigm assures the programmer that no surprise program errors should occur, if the methods or data within an object are modified.

The abstract machine class used in this research must be able to attain the machine characteristics of any kind of machine. To do this, production equations must reside as methods in the machine class object. However it is impossible to foresee all the types of machines a given user may require, so they must be dynamically created when needed. To accomplish this the machine class object must be modified to include the production equation required by a new type of machine. Data encapsulation almost guarantees that no surprise affects will occur due to the addition of the new method.

Inheritance

Objects in an object oriented system are arranged in hierarchies. Classes of objects located higher in the hierarchy are usually more generalized and the classes lower in the hierarchy are more specialized. Classes higher in the class hierarchy are usually called super classes while classes lower in the hierarchy are called subclasses. In Smalltalk all objects are a subclass of the class object.

A fundamental concept in object-orientated programming is that of a class. The class hierarchy defines the properties of each subclass created. A newly created subclass inherits the method protocol of any of it's superclasses higher in the class hierarchy. This facility for inheritance greatly reduces the amount of redundant coding. A class is an object itself, but its main purpose is for creating copies of itself. These copies of class objects are called instances of the class. When a class produces instances of itself we say it is instantiated. Since an instance is a copy of a class object it also inherits all the descriptive protocol from the class hierarchy.

When developing an object-oriented system a programmer must decide whether it is beneficial to make a subclass to an existing class or not. Smalltalk comes with a rich library of object classes of which the class Collection defines the protocol for many useful data structures. This research created many useful subclasses of the class Collection which inherited all the protocol of classes higher in the class hierarchy. The primary result of inheritance is greatly reduced development time.

Polymorphism

This object-oriented concept allows the same message to be used on different objects with the same or very different results. The ability to use the same message for similar operations on different kinds of objects allows for completely new classes of objects to be used in existing applications. An additional benefit is that it facilitates the use of consistently logical message names, thereby increasing program clarity.

System Layout

Developing an object-oriented system, begins with defining the problem domain and a solution domain. With an understanding of both domains the programmer identifies the objects required to reach a solution. The object-oriented system of this research was designed around the following steps.

1. Definition of the problem domain.
   (as explained in chapter 1).

2. Object definition.

    a) a subclass of existing object class ?
    b) create new subclass ?

3. Object interaction.

    a) Identify the messages that each object should respond to.

    b) Create methods to answer required messages.

    c) Establish the required sequence of messages between interacting objects that will lead to a problem solution.

Object Definition

System objects are arranged in a hierarchical manner with all objects being a subclass of the class Object. The system hierarchy is arranged in the following manner with an asterisk indicating objects unique to the Timber Harvester.

```
         Object
          Collection
           IndexedCollection
            OrderedCollection
*            System
            Dictionary
*            DictionaryDictionary
*            DictionaryOrderedCollection
*         Environment
*         Machine
*         SystemFinder
*          BestFirstSelection
*         TimberSimPane
```

For a definition of the Smalltalk system objects see (SMALLTALK/V Tutorial and Programming Handbook, 1986). A brief description of the objects unique to the TIMBER HARVEST is given below.

<u>System</u>

This object class holds a state which is added or removed from the queue during the search procedure. Since it is a subclass of OrderedCollection it inherits all the properties of that class.

<u>DictionaryDictionary</u>

This object class is a subclass of the class Dictionary which is instantiated to create data structures to store and retrieve objects. As the name suggests this data structure is a dictionary within a dictionary. This object Inherits all the protocol of the class Dictionary.

<u>DictionaryOrderedCollection</u>

This object class is a subclass of the class Dictionary which is instantiated to create data structures to store and retrieve objects. As the name suggests this data structure is an orderedCollection within a dictionary. This object Inherits all the protocol of the class Dictionary.

<u>Environment</u>

This object's main function is to provide instance variables and the pooled Dictionary (EnvironmentDictionary) to other system objects. The pooled Dictionary houses the key/value pairs for the user vector.

<u>Machine</u>

The Machine class object is an abstract class that contains the instance variables and methods that comprise a generic machine. When instantiated the machine object can take on specific attributes of a particular machine.

<u>SystemFinder</u>

The SystemFinder class provides the instance variables and messages that are needed by search routines, including the BestFirstSelection used by the Timber Harvester. The SystemFinder class is a subclass of the class Object and has the sole purpose of providing instance variables and generic methods for its subclasses. BestFirstSelection is a subclass of SystemFinder and so by the inheritance property inherits the instance variables and generic methods. The purpose for this decoupling is so more than one search algorithm can easily be explored. To incorporate another search algorithm one would only have to create another subclass and define the object protocol.

<u>BestFirstSelection</u>

This object along with its superclass SystemFinder performs the best first search. The BestFirstSelection has no instance variables of its own, only those of its super class which are inherited. The BestFirstSelection contains the instance methods that are specific to a best first search.

<u>TimberSimPane</u>

The TimberSimPane is the application frame for the Timber Harvester, which means it defines all the panes and controls all program flow. This object is by necessity large with many methods and instance variables.

The system organization is simplified by defining objects as belonging to one of the following five class categories.

1. Data storage class objects
   - System
   - OrderedCollection
   - Dictionary
   - DictionaryDictionary
   - DictionaryOrderedCollection

2. User's Environment class objects
   - Environment

3. Equipment class objects
   - Machine

4. Search class objects
   - SystemFinder
   - BestFirstSelection

5. User interface class objects
   - TimberSimPane

However it must be remembered that many objects perform data storage that are not categorized under that heading. Objects that fall under the data storage heading are used almost exclusively for that purpose. There are some important global objects that are used to contain data that must be accessible to all other system objects. The two equipment databases are global objects and are instances of the class DictionaryDictionary whose names are Operations and OperationsAvailable. This data structure allows us to store equipment by operation name and by the specific equipment name.

The OperationsAvailable database as mentioned previously is a subset of the Operations database. Two other important databases have global access, WoodDensity and SkidderSpeeds. WoodDensity is an instance of the class Dictionary and contains key value pairs relating a species symbol to a numeric density value. The SkidderSpeeds database contains skidder performance measures by horse power class, attachment class and terrain slope. The SkidderSpeeds database is an instance of the class DictionaryDictionary.

The bulk of the computer system is comprised of the Environment, Equipment, and Search class objects. These objects interact by sending messages to one another until a problem solution is attained. Program operation can best be discussed by dissecting the search objects and discussing their methods and messages.

Object Interaction

Assuming the necessary environment and equipment data has been input into the system, the user invokes the search algorithm by selecting the " Match Equipment " option during program execution. This selection sends a message called startSystem to an instance of BestFirstSelection called EquipmentSelector. The corresponding method residing in the object is invoked resulting in an initial queue. The queue contains all types of equipment for the start operation that passed the mapping procedures. This queue is the starting set of all unexpanded systems, where one or more starting points may eventually become feasible harvesting systems. The main search algorithm is now invoked by a message (called search) which begins by expanding the lowest cost equipment.

SEARCH METHOD (refer to Figure 2.1).

The search process begins by checking to see if the queue is empty. If the queue is empty the program reports that a suitable system is not possible. The user may redefine the user vector and run the search again or exit the system. If there are system objects in the queue the lowest cost system object is removed and stored it in a local variable called aState. The partial system represented by the variable aState is then checked to see if the goal states are satisfied. If both the goal location and the goal product states match the user specified states the program will report that a suitable system has been identified.

If the goal requirements were not met, the partial system is sent as a parameter with the message expandState. The expandState message is a BestFirstSelection instance method that is responsible for expanding the partial system with successor states (Figure 3.1). To do the actual identification, the method sends the message feasibleMachinesFrom, again with the partial system as a parameter.

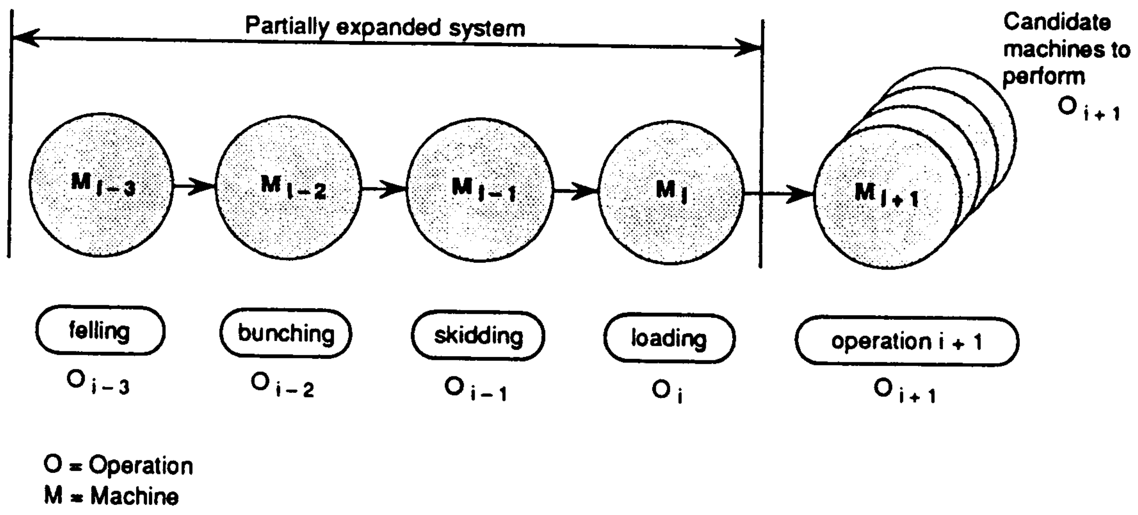


Figure 3.1 System Representation.

The feasibleMachinesFrom method's main function is to return an OrderedCollection of pieces of equipment to the expandState method. The orderedCollection contains the equipment that the partial system object could feasibly move to from the last piece of equipment in the partial system. The feasibleMachinesFrom method performs the three critical maps previously discussed.

The feasibleMachines method proceeds by iterating through the feasible operations (Figure 3.1). The method first checks whether the current system already contains the candidate operation. If the system does not contain the operation in question, then the process continues, and if it does contain the operation , it is discarded and an alternative operation, if one exists is selected for evaluation. The next step is to identify the equipment available to perform the new operation. The equipment filed under the new operation are evaluated separately. The evaluation consists of mapping the machine attributes previously discussed to the users environment vector.

The message processMaterial is sent to the candidate equipment with the parameters outputProductState, locationOut and accumulationStateOut. The parameters represent the current state of the material.

The processMaterial method directly defines the equipment compatibility. An equipment that cannot process the current material state, is by definition an incompatible piece of equipment for expanding the current system. This process of passing the output variables equipment as input variables to a candidate piece of equipment, bears a strong analogy to passing a baton in a rely race.

If the equipment can process the current material state, the equipment is then mapped to the user environment vector. The equipment is sent the message compCumVector, whose name is an abbreviation for Compute Cumulative Vector. This equipment vector is created when the equipment attributes are mapped to the user environment vector. The vector is a set of binary values that describe a successful attribute map with a 1 or an unsuccessful attribute map with a 0. If the size of the Cumulative Vector is equal to the number of occurrences of the value 1 a 'pass' is stored in the equipment's instance variable mappingFactor, otherwise 'fail' is recorded.

If a 'pass' is recorded the piece of equipment is defined as compatible with the user's specified harvesting environment and therefore defined as a feasible successor equipment. Before an equipment can be added to a copy of the partial unexpanded system the production cost must be calculated. If no production equation is defined the user defined scaler value is used. If a production equation is defined a sequence of three steps are followed to evaluate costs.

1. Determine the name of the production equation.
2. Perform the production equation.
3. Store the result in the instance variable prodCost.

The production equation provides the ability to model an equipment's operating cost as a function of the specific stand environment. Once the production equation is calculated the equipment is added to a collection of feasible expansion equipments called feasibleMachines. When the operations list is exhausted the collection called feasibleMachines is returned to the expandState message which was the calling method for the feasibleMachineFrom method.

The expandState method then adds each equipment contained in the feasibleMachines collection to a copy of the partial system to be expanded, producing successor states of the partial system. Each new expanded system represents a new state that has been created. The number of created states is echoed back to the user during program execution. The newly created states are added to a collection called successors and returned from the expandState method to the calling method search. The search method adds the successor collection to the queue of unexpanded systems and then sorts the queue where the whole search process begins again.

All programming in Smalltalk is accomplished by sending messages and from the previous discussion it should be apparent that the nesting of messages and sequences of messages can become fairly complex. It should also be apparent that objects can send messages to themselves quite frequently during method execution.

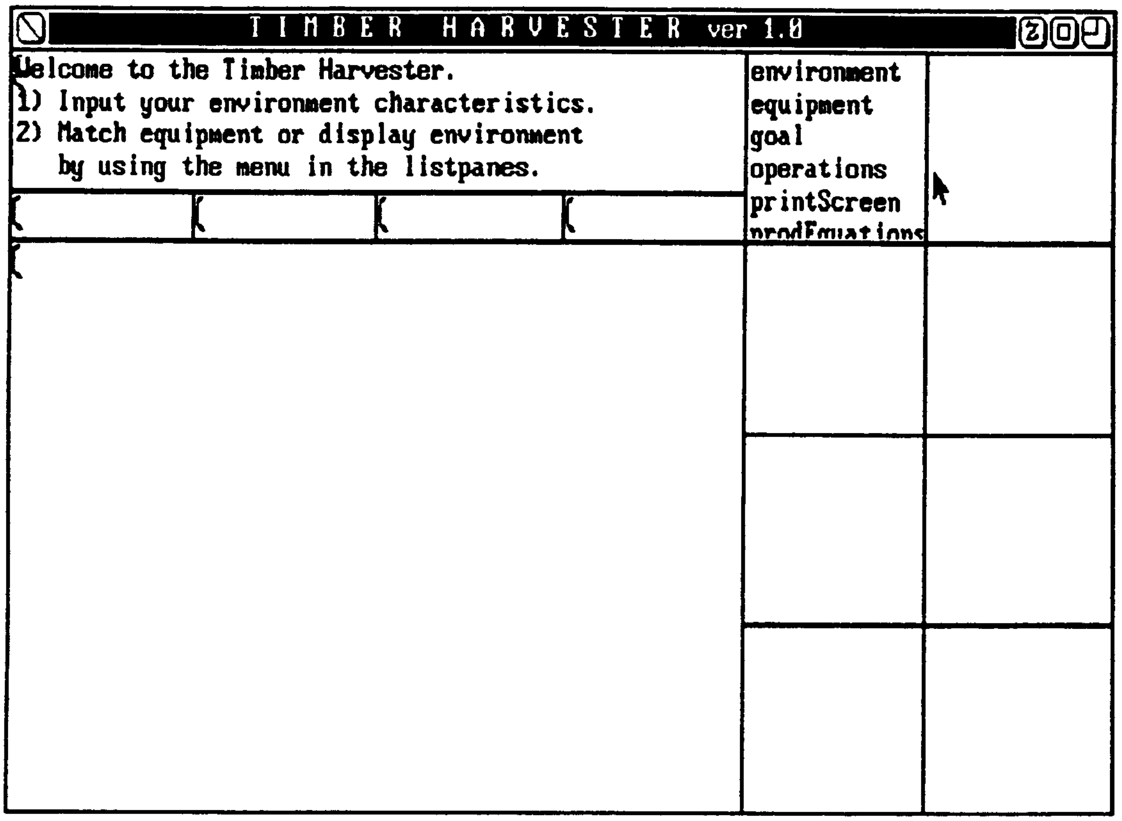
CHAPTER 4

APPLICATION AND IMPLEMENTATION

Timber Harvester Organization

The Timber Harvester is laid out in a system of rectangular windows. There are two types of windows, list pane windows and text pane windows. There are two text panes and 8 list panes (Figure 4.1). The window pane feature of the Timber Harvester makes it very user friendly. This feature also allows a user with no knowledge of Smalltalk to interact with the system successfully.

To operate the system a user is required to make an appropriate selection from a menu and click on the mouse. The system uses a networked-menu system, that is, a choice on a menu leads to a sub menu and so on. At each step a "text window" explains the menu options and parameters to be specified. This hierarchial representation keeps a user supplied with up to date information on the state of the system.



Welcome to the Timber Harvester.
1) Input your environment characteristics.
2) Match equipment or display environment
by using the menu in the listpanes.

Figure 4.1 Timber Harvester User Interface.

In addition to executing the equipment search algorithm, the Timber Harvester provides complete editing capabilities for the equipment and support data bases, and the user environment. Specific functions provided in the Timber Harvester include:

1. Creation, modification and removal of harvesting operations.
2. Creation, modification and removal of harvesting equipment for specific operations.
3. Assignment and modification of equipment attributes.
4. Creation and modification of the contents of the support data bases.
5. Defining harvesting requirements (site specifics, stand specifics, etc).
6. Printing hard copies of text panes and screen dumps.

The alternatives selected by Timber Harvester depend on the information provided in the equipment and support data bases. There is a wide range of mechanized equipment available for the many different tasks in a harvesting operation. To reduce the task of finding production equations for every piece of equipment to manageable size, similar function machines were grouped together as a generic equipment type and further broken down by subcategories such as horse power class. Suitable published production equations were used for the production rates for the generic equipment types in the system. Where production equations were unavailable or inappropriate, individual equipment studies were grouped together and production rates transformed through regression analysis to develop production equations for the generic equipment class. If no production data was found, "ball park" estimates were created to enable the program to operate. The user has the option of overriding the production figures if so desired.

Production rates for individual equipment were obtained from published production studies form organizations such as : Forest Engineering Research Institute of Canada (FERIC), Logging Industry Research Association (LIRA), Canadian Pulp and Paper Association (CPPA), manufacturers handbooks and articles published in trade magazines.

Timber Harvester Application Example

The Timber Harvester has been verified for accuracy and validated using a timber harvesting problem domain. To illustrate the application of Timber Harvester, consider the subsystem of a larger problem domain shown in Figure 4.2. Each node shows the product state after the foregoing operation is complete. The system shown in Figure 4.2 starts with standing timber to be harvested, therefore the start operation will have one equipment called begin. The name of the equipment is strictly arbitrary. The begin equipment describes the initial harvesting product state through it's output product states. These output product

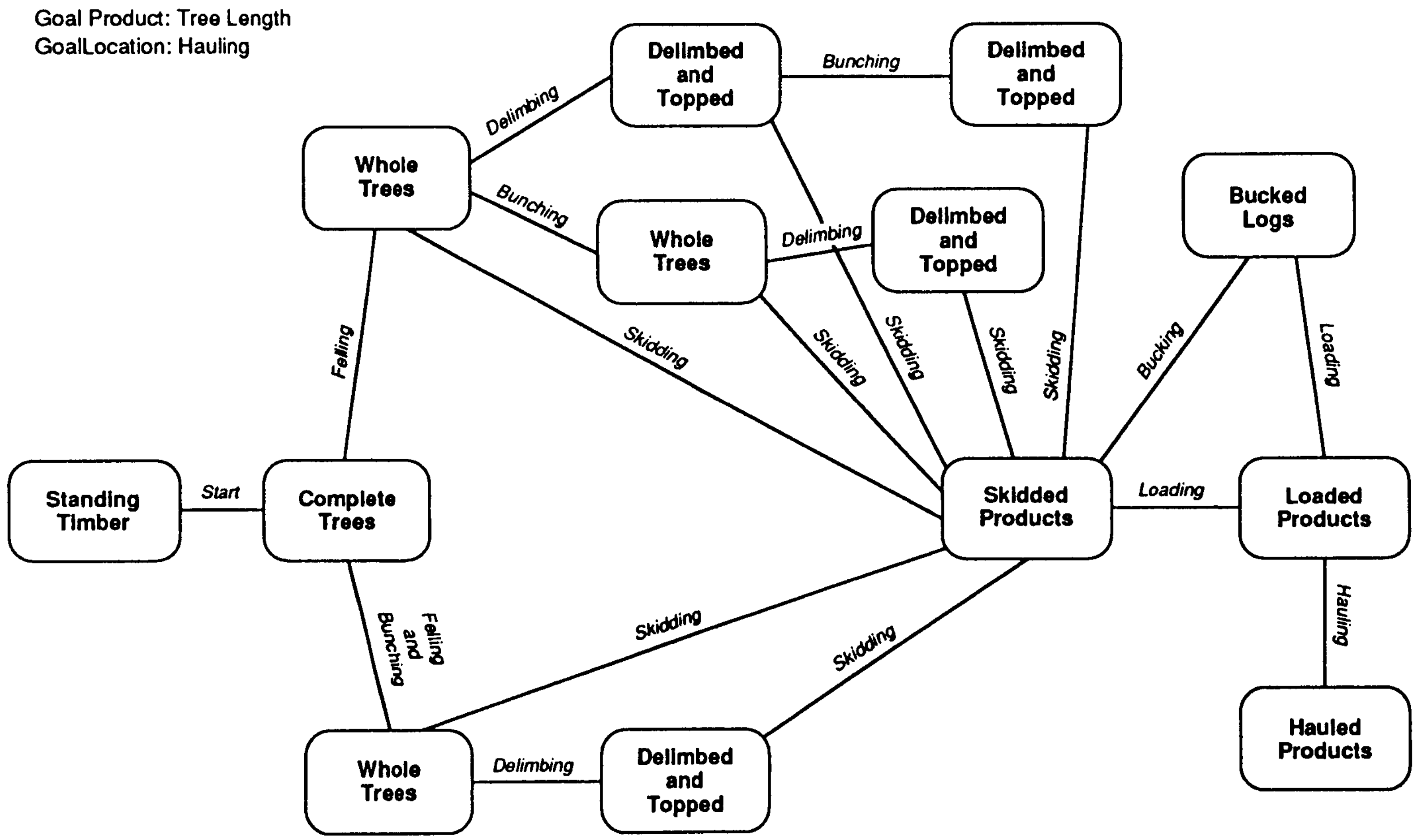


Figure 4.2 System Application Example

states are important because they define what equipment can start the harvesting operation. Only those equipments that can process the initial product states become feasible equipment to start the harvesting operation. The initial product state is as follows:

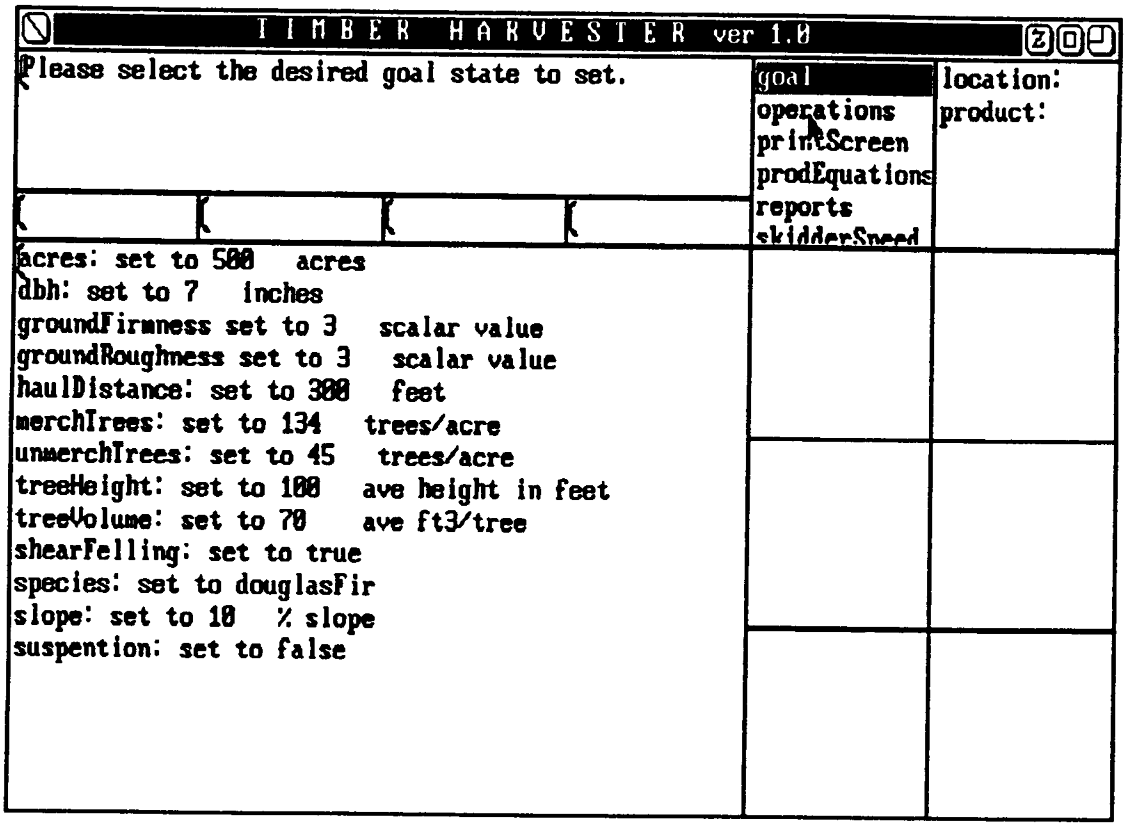
1. location = standing.
2. accumulation = random.
3. product state = complete tree.

This is the starting variable assignment for most timber harvesting situations that are encountered. The start operation and the begin equipment are usually preprogrammed and their inclusion in this section is only for completeness. The user is required only to describe their specific harvesting environment and then initiate the search algorithm. Table 1 summarizes the set of attributes for the harvesting environment and (Figure 4.3) shows the screen display. Table 2 summarizes the equipment built in the database that may be used by the processes in the application system. The (Figure 4.4) shows a screen display of the equipment attribute assignment for a given piece of equipment.

As previously discussed the completion of each operation gives rise to three important variables: product state, accumulation state and location. The three variables define the equipment compatibility. For example, many grapple skidders require the material to be bunched prior to skidding. Therefore any system that does not yield a bunched accumulation state prior to skidding will not be able to employ grapple skidders. Therefore grapple skidders are sensitive to the accumulation state of the material.

Table 1. User Environment Summary

Variable	Scale	Range of Measurement
SITE VARIABLES		
groundFirmness	Discrete Scale	1-5 1:Soft 5:Hard
groundRoughness	Discrete Scale	1-5 1:mild 5:Severe
Slope	Percent Slope	-30 to 30
Haul Distance	Distance (feet)	N/A
STAND VARIABLES		
acres	Stand size (acres)	N/A
dbh	Avg. diameter (in)	1 to 40 (in)
merchantable Trees	Merch. trees/acre	N/A
species	Species name	N/A
tree height	Avg. height (ft)	N/A
tree Volume	Volume (cubic ft)	N/A
unmerch Trees	Unmerch. trees/acre	N/A
PRODUCT REQUIREMENTS		
shear felling	Binary discrete scale	N/A
suspension	Binary discrete scale	N/A



Please select the desired goal state to set.

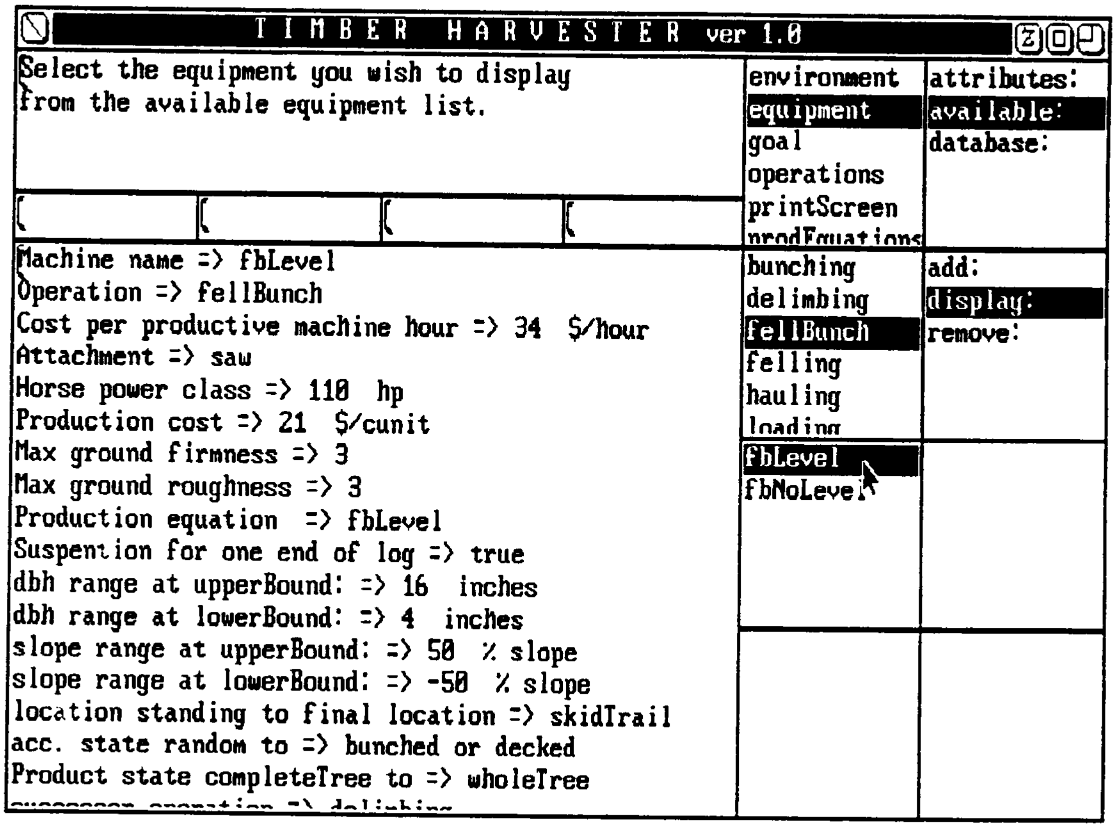
```
acres: set to 500   acres
dbh: set to 7   inches
groundFirmness set to 3   scalar value
groundRoughness set to 3   scalar value
haulDistance: set to 300   feet
merchTrees: set to 134   trees/acre
unmerchTrees: set to 45   trees/acre
treeHeight: set to 100   ave height in feet
treeVolume: set to 70   ave ft3/tree
shearFelling: set to true
species: set to douglasFir
slope: set to 10   % slope
suspention: set to false
```

Figure 4.3 The User's Environment

Table 2. Available Equipment Used in System Application Example

Operation: felling	Equipment
	Manual (manual)
Operation: fellBunch	Equipment
	Feller buncher no level
(fbNoLevel)	
	Feller buncher level (fbLevel)
Operation: bunching	Equipment
	Wheeled Grapple (wheelGrp)
Operation: delimbing	Equipment
	Manual (manual)
	Processor (processor)
Operation: loading	Equipment
	Stroke boom loader (StrokeBoom)
Operation: Skidding	Equipment
	Caterpiller with Cable (catCable)
	Caterpiller with Grapple (catGrapple)

( ) indicates shortened computer name.



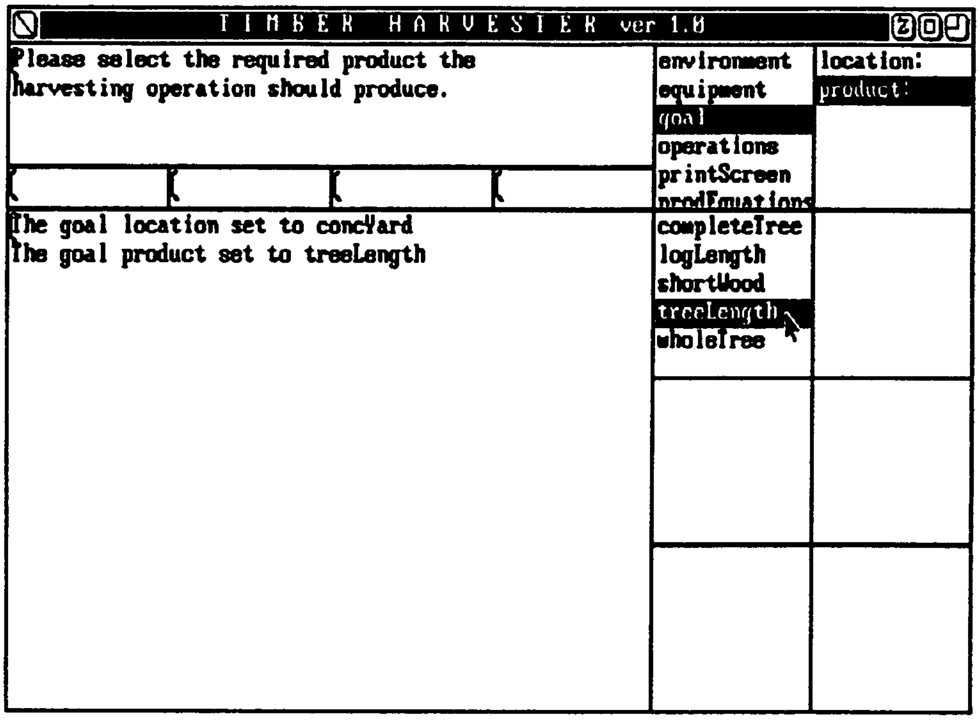
Select the equipment you wish to display from the available equipment list.

```
Machine name => fbLevel
Operation => fellBunch
Cost per productive machine hour => 34  $/hour
Attachment => saw
Horse power class => 110  hp
Production cost => 21  $/cunit
Max ground firmness => 3
Max ground roughness => 3
Production equation  => fbLevel
Suspention for one end of log => true
dbh range at upperBound: => 16  inches
dbh range at lowerBound: => 4  inches
slope range at upperBound: => 50  % slope
slope range at lowerBound: => -50  % slope
location standing to final location => skidTrail
acc. state random to => bunched or decked
Product state completeTree to => wholeTree
```

Figure 4.4 Equipment Attributes

It should be noted that output variables of one operation are input variables to the successor operation.

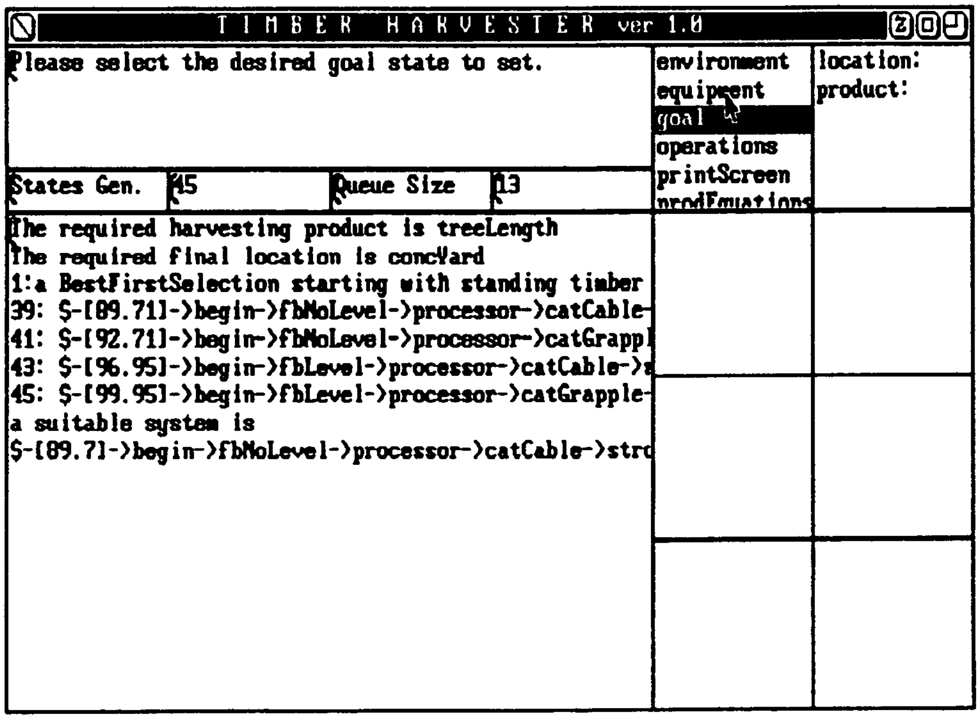
The objective is to generate feasible harvesting systems and identify the most cost effective alternative. The user is required to select the goal location and the goal product desired and then initiate the search process (Figure 4.5). During execution the Timber Harvester displays each alternative that meets both the goal location and the goal product (Figure 4.6). Referring to (Figure 4.6) the number on the far left correlates to the state number the search algorithm assigned to the system; it serves as the identification number for the report. The next number in square brackets is the logging cost per mbf. Referring to (Figure 4.7) the list of equipment identifies a suitable harvesting system. The user can display (or print a hard copy) of the system. The computer screen can scroll from right to left and top to bottom in any window, the (Figure 4.8) shows the complete suitable harvesting system selected. Furthermore, the user may query the Timber Harvester for details on specific equipment of the system identified.



Please select the required product the harvesting operation should produce.

```
The goal location set to concYard
The goal product set to treeLength
```

Figure 4.5 Select Goals

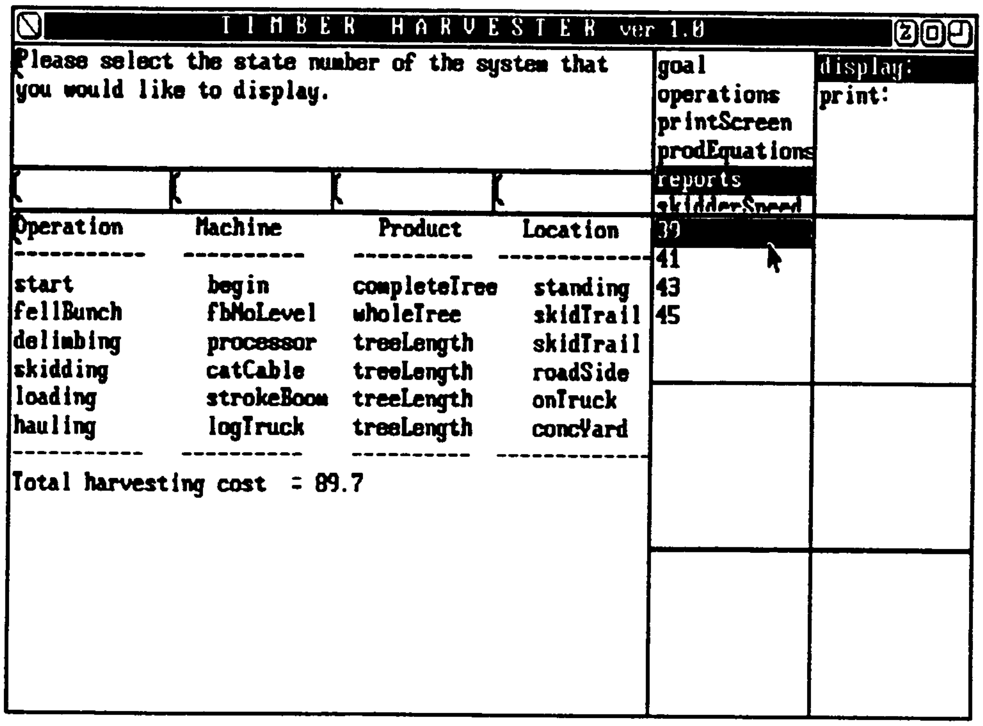


Please select the desired goal state to set.

States Gen. 45 Queue Size 13

```
The required harvesting product is treeLength
The required final location is concYard
1:a BestFirstSelection starting with standing timber
39: $-[89.71]->begin->fbNoLevel->processor->catCable-
41: $-[92.71]->begin->fbNoLevel->processor->catGrappl
43: $-[96.95]->begin->fbLevel->processor->catCable->
45: $-[99.95]->begin->fbLevel->processor->catGrapple-
a suitable system is
$-[89.7]->begin->fbNoLevel->processor->catCable->str
```

Figure 4.6 Feasible Alternatives



```
Please select the state number of the system that
you would like to display.

Operation    Machine      Product       Location
-----------  ----------   ----------   ------------
start         begin       completeTree  standing
fellBunch     fbNoLevel   wholeTree     skidTrail
delimbing     processor   treeLength    skidTrail
skidding      catCable    treeLength    roadSide
loading       strokeBoom  treeLength    onTruck
hauling       logTruck    treeLength    concYard
-----------  ----------   ----------   ------------
Total harvesting cost  = 89.7
```

Figure 4.7 System Report

Operation	Machine	Product	Location	Accumulation
start	begin	completeTree	standing	random
fellBunch	fbNoLevel	wholeTree	skidTrail	bunched or Decked
delimbing	processor	treeLength	skidTrail	bunched or Decked
skidding	catCable	treeLength	roadSide	bunched or Decked
loading	strokeBoom	treeLength	onTruck	bunched or Decked
hauling	logTruck	treeLength	concYard	bunched or Decked

Figure 4.8 Full View System Report.

CHAPTER 5

CONCLUSIONS

The design, evaluation and control of mechanized timber harvesting systems is a problem of considerable importance in the forest products industry, and has economic as well as societal relevance. The type of processing that is chosen affects the type and quality of log that is delivered to the mill, and ultimately the finished wood product. No previous work has provided a design and evaluation tool in this important area of application. The Timber Harvester provides a modeling tool for identifying feasible alternatives for a specific harvesting environment by including such factors as production costs, production efficiency, environmental considerations and specific user goals.

The object-oriented framework proved to have unique flexibility throughout the research, and greatly reduced the development duration. The ability to create working copies of class objects and use them throughout the system greatly reduced the amount of redundant code, while increasing the level of abstraction. The level of abstraction is of supreme importance, for it allows this research to be applied to another process type problem domain. To apply the Timber Harvester to another process problem domain the user interface object TimberSimPane would need to be replaced. The reason for replacing this object is so that pertinent attribute lists and other domain specific attributes could be evaluated. In short the object-oriented framework will permit the application of the Timber Harvester to another problem domain with a minimal amount of programming.

The direction for future research is two fold: methodology enhancement and the expansion of the timber harvesting application. Methodology enhancement will encompass a code translation to a hybrid object-oriented language such as C++, which will result in a compiled version of the Timber Harvester. The compiled version would run much faster and would also give the programmer better control over memory allocation and control, which larger scale applications will require.

Expanding the application example from identifying a suitable system for a single timber stand to encompassing the entire yearly cut for a given mill. For example, a mill that may purchase timber on a National Forest will end up with many different sales with many different environmental characteristics. The mill must allocate available harvesting systems to their most promising sites, both from an economic as well as ecologic view point. The complete model would include screening the feasible alternatives resulting from the matching process against user constraints and objectives. These constraints and or objectives include available capital , long-term investment, production goals, crew skills, environmental constraints, and state and federal regulations.

BIBLIOGRAPHY

Calu, J., Wael, L.D., Esmeijer, E., Kerckhoffs, E.J.H. and Vansteenkiste, G.C., "Knowledge-Base Aspects in Advance Modeling and Simulation," Pro. 1984 Summer Simulation Conf., 1247-1253, 1984.

Conway, S., Logging Practices, Miller Freeman Publ. Inc., San Francisco, 1982.

Cox, B.J., Object Oriented Programming : An Evolutionary Approach, Addison-Wesley Publ. Co., Reading, MA, 1986.

Goldin, S.E. and Klahr, P., "Learning and Abstraction in Simulation," Pro, 7th International Joint Conference on Artificial Intelligence, 212-214, 1981.

Gory, G.A. and Krumland, R.G., "Artificial Intelligence Research and Decision Support Systems," in Ed,. Bennett, J.L., Building Decision Support Systems, Addision-Wesley Publ. Co., Reading, MA, 1983.

Kerckhoffs, E.J.H. and Vansteenkiste, G.C., "The Impact of Advanced Information Processing on Simulation - An Illustrative Review," Simulation, 46, 1, 17-26 1986.

Law, A.M. and Kelton, W.D., Simulation Modeling and Analysis ,McGraw-Hill Book Co., New York, NY, 1982.

Lewis, J.P. and Wiener, R.S., An Introduction to Object-Oriented Programming and Smalltalk, Addision-Wesley Publ. Co., Reading MA, 1988.

Oren, T.E., "Simulation - As it has been, is, and should be, " Simulation, 29, 5, 182-183, 1977.

Oren, T.I. and Ziegler, B.P. , "Concepts for Advanced Simulation Methodologies," Simulation, 32, 3, 69-82, 1979.

Ruiz-Meir, S., Talvage, J. and Ben-Arieh, D., "Towards a Knowledge-Based Network Simulation Environment," Pro. 1985 Winter Simulation Conf., 232-236, 1985.

Shannon, R.E., "Artificial Intelligence and Simulation," Pro. 1984 Winter Simulation Conf., 3-9, 1984.

Shannon, R.E., Mayer, R. and Adelsberger, H.H., "Expert Systems and Simulation, Simulation, 44, 6, 275-284, 1985.

SMALLTALK /V Tutorial and Programming Handbook, Digitalk Inc, Los Angeles, Ca, 1986.

Randhawa, S.U. and Olsen, E.D., "LOGSIM: A Tool for Mechanized Harvesting Systems Design and Analysis," Applied Engineering in Agriculture, 6, 2, 231-237, 1990.

Rich, E., Artificial Intelligence, McGraw-Hill Book Co., New York, NY, 1983.

Tello, E.R., Object-Oriented Programming for Artificial Intelligence, Addision-Wesley Publ. Co., Reading, MA, 1989.

APPENDICES

APPENDIX 1.

USERS MANUAL

Table of Contents

INTRODUCTION

The Timber Harvester was designed to automatically generate feasible timber harvesting systems. System selection is based on the interaction of four major components.

1. The user's specific harvesting environment.
2. The specific harvesting requirements.
3. The available harvesting equipment's operating attributes.
4. The required product and it's final location. (ie) the goal states.

The user is required to build the user environment by using a mouse to select each environment attribute. The harvesting requirements are input in a similar manner, to build in constraints that will determine the direction of the search for feasible alternatives. These requirements might be the allowance or prohibition of using a shear type feller buncher for felling. Each piece of harvesting equipment is assigned to a harvesting operation. If a piece of equipment is multi - operational, it must reside in an operation that is multi-operational. An example would be a feller buncher that not only fells trees but also bunches them, would reside in the operation fellBunch. That actual name of the operation can be anything but some logical expression should be used.

The Timber Harvester uses a best first search, which uses the production cost as the driving factor. This production cost is either an input cost or is calculated using a production equation. The production equation will compute different values for different user environments. The input production cost is only used if no production equation is currently available for the program to access.

Given all four components the Timber Harvester's search algorithm will map the user environment to the available equipment database to produce feasible harvesting alternatives.
Once alternatives have been generated they can be displayed to reveal the following equipment attributes for each operation.

1. Product Produced - The output product produced.
2. Accumulation State - The accumulation state of the output product (bunched etc.).
3. Location - The output product location (landing etc.).
4. Production Cost - Production cost for each operation.

The Timber Harvester always displays all feasible systems until a lowest cost feasible solution is identified.

All of these alternatives can then be used as input into a computer program LOGSIM (Randawa and Olsen 1989). The procedural simulation can then analyze the equipment balancing and other system specifics.

SYSTEM LAYOUT

The Timber Harvester is laid out in a system of rectangular windows. There are two types of windows, list pane windows and text pane windows. There are two text panes and 8 list panes laid out in the following manner.

```
|----------------------------------------------------------|
|                                   |  LIST    |   LIST    |
|       TEXT PANE 1.                |  PANE    |   PANE    |
|                                   |          |           |
|-----------------------------------|   1.     |    2.     |
|      |       |       |     |     |          |           |
|----------------------------------------------------------|
|                                   |  LIST    |   LIST    |
|                                   |  PANE    |   PANE    |
|                                   |          |           |
|                                   |   3.     |    4.     |
|       TEXT PANE 2.                |          |           |
|                                   |----------------------|
|                                   |  LIST    |   LIST    |
|                                   |  PANE    |   PANE    |
|                                   |          |           |
|                                   |   5.     |    6.     |
|                                   |          |           |
|                                   |----------------------|
|                                   |  LIST    |   LIST    |
|                                   |  PANE    |   PANE    |
|                                   |          |           |
|                                   |    7.    |    8.     |
|                                   |          |           |
|----------------------------------------------------------|
```

TEXT PANE 1.

Text pane 1 is a help pane which walks the user through all the operations of the Timber Harvester.

TEXT PANE 2.

Text pane 2 is a response area for the timber harvester.

LIST PANES 1 - 8.

The list panes 2 - 7 can take on different lists depending upon which option is selected in list pane 1.

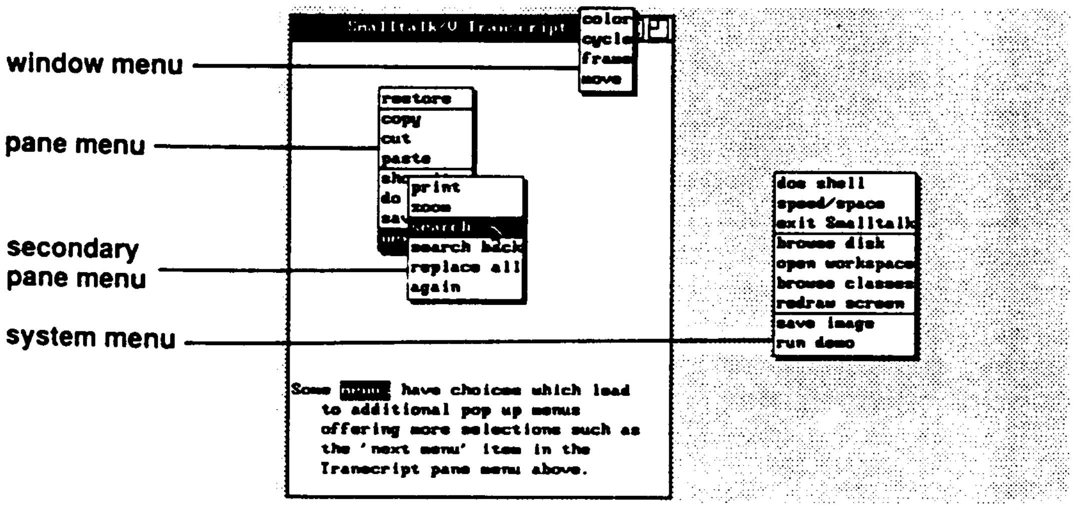
MOUSE

The typical mouse used with the Timber Harvester has two selection buttons. The left button is used to make a selection and the right button is used to bring up a menu.

The window pane feature of the Timber Harvester makes it very user friendly. These panes will be referred to by name in the following chapters. The Timber Harvest is written in an object-oriented language called Smalltalk which an interpreted language. This may require some Smalltalk interaction from the user. The amount of this interaction will be minimal but the basic's of the smalltalk environment need to be discussed.

Exiting Smalltalk

To exit Smalltalk/V and return to DOS, move the cursor to the background and click the right mouse button to bring up the **SYSTEM MENU**. Select the exit smalltalk item by clicking the left mouse button when exit smalltalk is highlighted (figure A1.1). If any changes have been made and the changes should be saved, select the save image option otherwise select forget image or continue to remain in smalltalk. Talking specifically about the



Some menus have choices which lead
to additional pop up menus
offering more selections such as
the 'next menu' item in the
Transcript pane menu above.

Figure A1.1 Different SmallTalk Windows

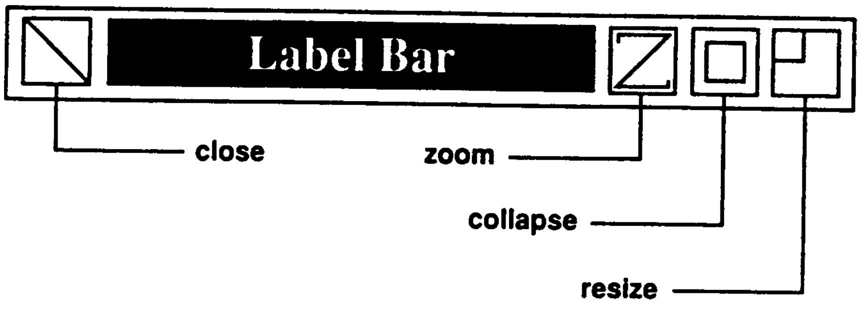


Figure A1.2 Window Label Bar and Window Buttons

Timber Harvester, if you have added equipment or made any changes select the save image option.

Moving Around the Smalltalk Environment

To get around the smalltalk/V environment , you must open up and close windows, make selections from popped up menus and move the cursor using a mouse or the keyboard.

The Cursor

The cursor is the pointer on the screen. It tells Smalltalk/V where you are going to do something like pop-up a window.

The I Beam

The insertion point or I-beam is a special text marker that is used when editing strings. It appears in a text window or pane and marks the spot where new text will be inserted or deleted.

Working with Your Mouse.

Your mouse has two buttons that Smalltalk/V uses. The right button " administrates " your way around the Smalltalk/V environment. Use the right button to bring up menus for selection and for scrolling within a window pane. The left button " selects " things for Smalltalk to execute. Use the left button to select menu items, text or text lines, and objects form a list.

Windows

A window is an object with a border, label bar, window buttons an one or more panes like those previously discussed, and may also have pop-up menus. A window can be active or non-active. Windows can be opened, closed, collapsed, resized and moved around the screen.

The Label Bar

Each window has its own label bar and menu. The window title is displayed at the top with one or more small buttons. The buttons provide quick access to specific window activities (figure A1.2).

<u>CLOSE BUTTON</u>

When selected, the window closes an disappears from the screen. When a window closes all information contained in the window is lost.

<u>ZOOM BUTTON</u>

When selected, Smalltalk/V zooms in on the text pane so that it fills the whole screen. To unzoom the text pane, click on the menu bar with the left button.

<u>COLLAPSE BUTTON</u>

When selected, the window collapses to show only the label bar. If the window is already collapsed, selecting this button expands the window to its original size.

<u>RESIZE BUTTON</u>

Select this button and the system responds with a rectangle outline for resizing the window.

1 - Defining the Harvesting Environment.

1.1 Defining the user environment variables.

The users environment is broken down into three logical units: requirements ,site and stand. They collectively make up the user environment thus defining the harvesting situation. The variables used to define the user environment will now be discussed in detail.

SITE VARIABLES

GroundFirmness

This variable contains a value from 1 - 5, which represents the firmness range. The value of one defines a soft ground firmness and value of five defines a hard ground firmness.

GroundRoughness

This variable contains a value from 1 - 5, which represents the roughness range. The value of one defines a mild ground roughness and a value of five defines a severe ground roughness.

Slope

The slope is defined by a value from -30 to +30. This value represents the average slope on the given site.

HaulDistance

The haul distance defines the average transportation distance from the woods to the landing or roadside.

STAND VARIABLES

Acres

The acres variable represents the size of the stand in acres.

Dbh

The dbh variable represents the average diameter at breast height of the major species in the stand.

MerchTrees

The merchantable tree variable represents the number of merchantable trees per acre.

Species

The species variable represents the name of the major species on the given site.

TreeHeight

The tree height variable represents the average tree height in feet for a stand.

TreeVolume

The tree volume variable represent the average cubic volume per tree for a stand.

UnmerchTrees

The unmerchantable tree variable represents the number of unmerchantable trees per acre.

REQUIREMENT VARIABLES

ShearFelling

The shear felling variable determines if a shear attachment may be used during a felling operation.

Suspension

The suspension variable determines if a log or tree must have one end suspended during transport.

1.2 Meeting the harvesting requirements.

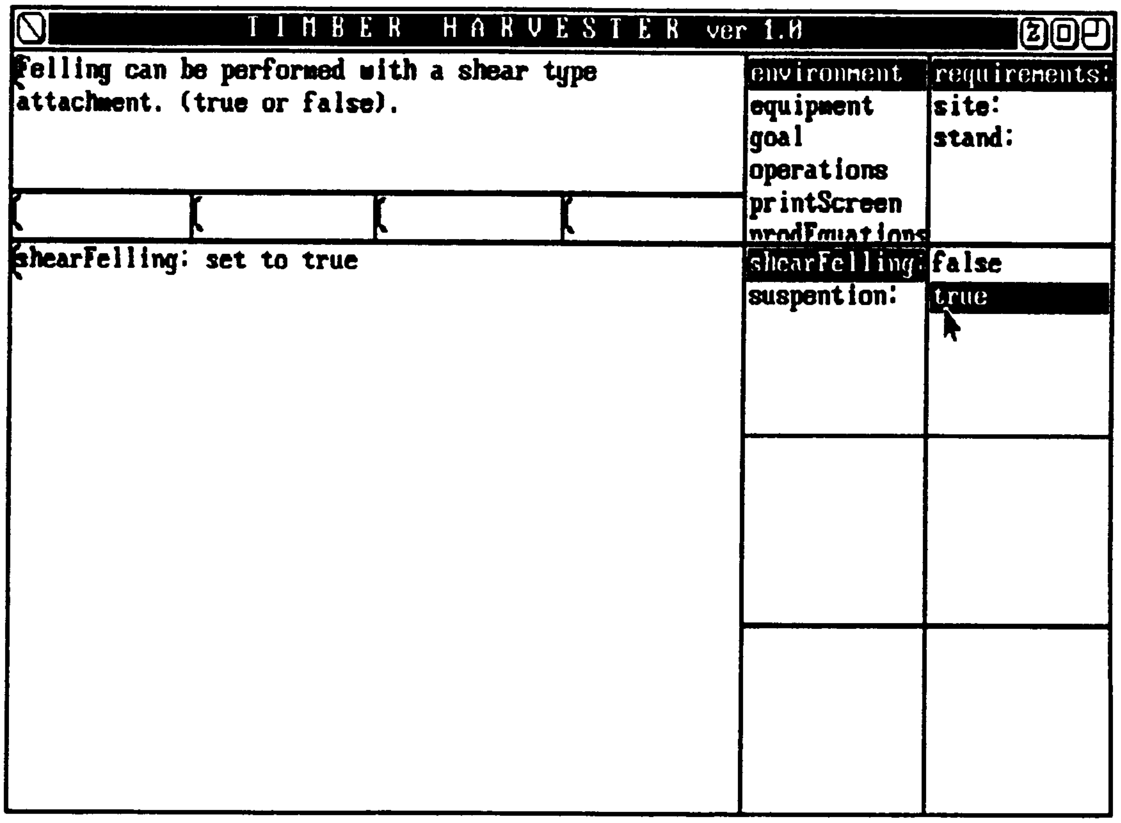
In any given timber harvesting situation there are a number of requirements that must be met. The requirement subset that the Timber Harvester incorporates presently is limited to the following.

1. ShearFelling - Can equipment use a shear for felling ?
2. Suspension - One end must be off the ground during transport (no plowing).

To set the requirements, the following sequence should be followed. (see figure A1.3)

1. Select the environment option in listPane1.
2. Select the requirements option in listPane2.
3. Select the appropriate requirement listPane3.

The requirement selection will basically eliminate any systems that do not satisfy the harvesting requirements. This means that if you select a whole tree harvest, no system involving a bucking operation will be feasible.


TIMBER HARVESTER ver 1.0

Felling can be performed with a shear type attachment. (true or false).

shearFelling: set to true

Figure A1.3 Setting Harvesting Requirements.

1.3 Defining the site specifics.

The site specifics define the terrain by assigning the values of 1 through 5 to ground roughness and ground firmness. The haul distance is a site specific variable that is defined by the distance in feet. The Slope is given a value of -30 to +30 which represents a range from 30 percent down hill to 30 percent uphill. To set the site variables, the following sequence should be followed. (see figure A1.4)

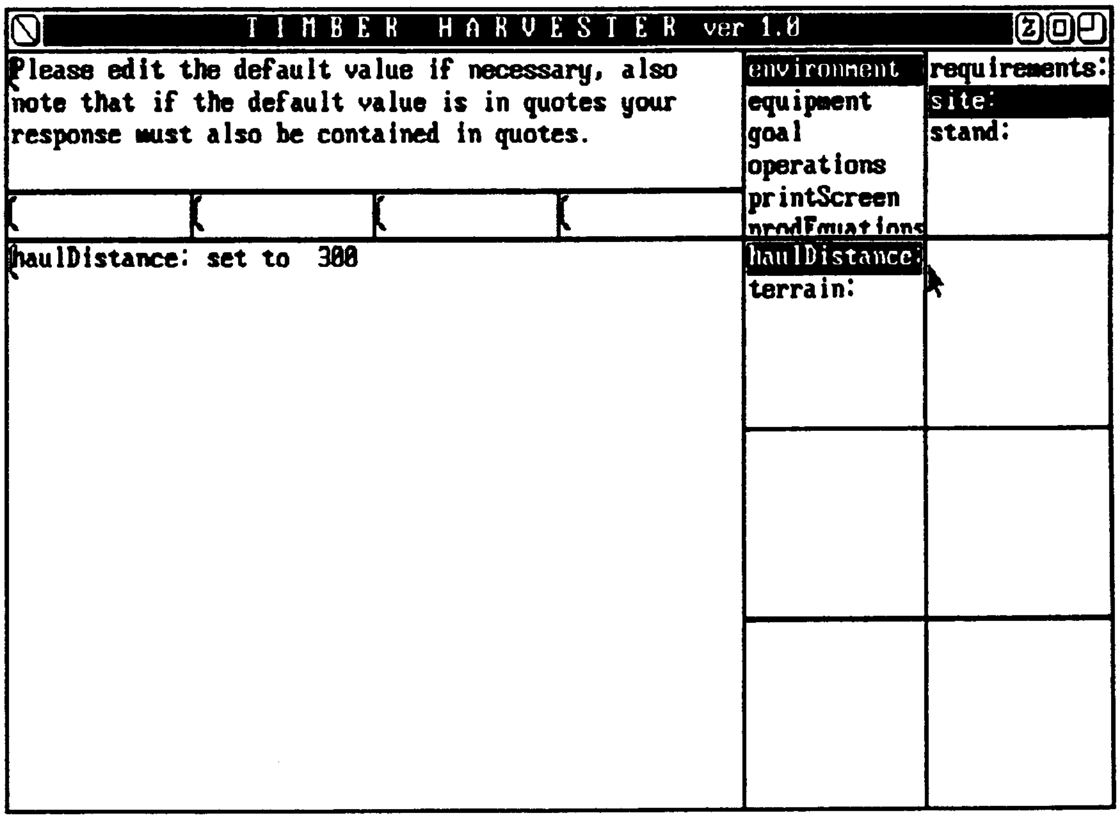
1. Select the environment option in listPane1.
2. Select the site option in listPane2.
3. Select the appropriate site variable in listPane3.

1.4 Defining the stand specifics.

Stand specifics define the timber stand variables such as dbh and species. To set the stand variables, the following sequence should be followed.

1. Select the environment option in listPane1.
2. Select the stand option in listPane2.
3. Select the appropriate stand variable in listPane3.

All of these variables form a user environment vector that will be used during the search procedure to map the user environment to a candidate piece of equipment.



Please edit the default value if necessary, also note that if the default value is in quotes your response must also be contained in quotes.

```
haulDistance: set to  300
```

Figure A1.4 Defining Site Specifics.

2 - Creation and Modification of the Equipment Database.

2.1 Creation/Removal of a harvesting operation.

The Timber Harvester comes as a blank slate. This means it comes with no user environment and no database of available technology. The previous chapter outlined the necessary steps to create the user environment, leaving the equipment database to be defined in this chapter. To begin building our database we first must clearly define what constitutes an operation.

Definition # 1. - An operation is a procedure that takes an input material form at a particular location and transforms the material into an output form at a particular location.

This definition seems simple enough, and it would be if each piece of equipment could perform only one operation. Many types of timber harvesting equipment can perform multiple operations. While these machines are flexible they are not always the most cost efficient for a given user environment. At times a system composed of singular function machines is more efficient. Therefore to allow the Timber Harvester to evaluate both types of systems we need to modify definition # 1.

Definition # 2 - An operation is a procedure that takes an input material form at a particular location and transforms the material into an output form at a particular location. The output form and location must be consistent with the operation for which a given piece of equipment is assigned.

[EXAMPLE] - For a felling operation the input form would be standing timber and the output form would be a whole tree. It would obviously be inconsistent to have a piece of equipment categorized for felling doing delimbing, because the output form would be a delimbed tree. Therefore all delimbing must be done by machines assigned to that operation, even if it is the same type of machine that did the felling.

** Note **

Some equipment will have an output form that is the same as the input form and the only difference will be a change in location, (eg) skidding.

The reason for the modification to definition # 1 will become readily apparent as we discuss building the database. The database of available equipment technology is arranged by operation. We begin building our database with the most primitive operation usually felling, but combination operations are often used. Once an operation is created we can add the available equipment to perform the operation. If a machine can perform multiple operations it must reside in a combination operation that has a specific input material form and a specific output form.

To create an operation perform the following sequence. (see figure A1.5)

1. Select operations in listPane1.
2. Select add option from listPane2.
3. Enter an Alpha-Numeric name in prompter window.

To remove an operation perform the following sequence.

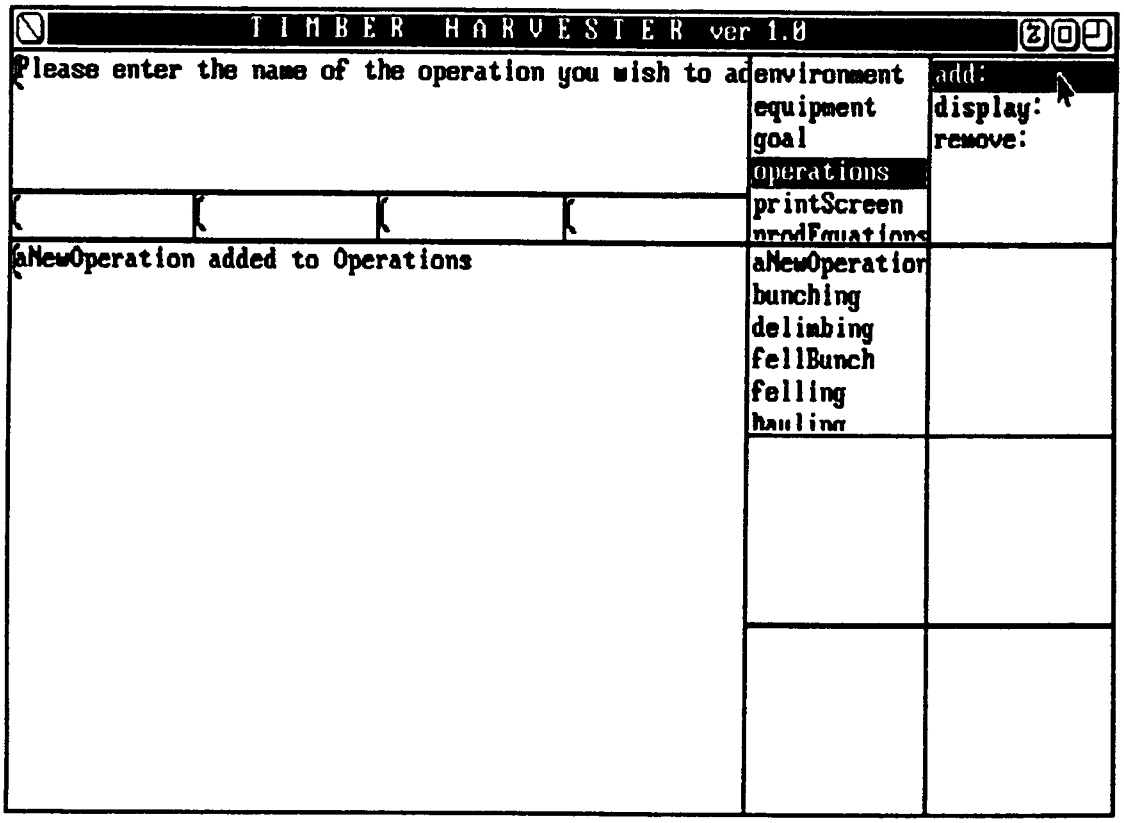
1. Select operations in listPane1.
2. Select remove option from listPane2.
3. Select the operation to remove from listPane3.

CAUTION ==> REMOVING AN OPERATION WILL REMOVE ALL THE EQUIPMENT IN THE SYSTEM THAT IS ASSIGNED TO THAT OPERATION.

2.2 Creation/Removal of harvesting equipment.

Once an operation is in the system, equipment can then be added under that operation to produce that operation's output form at a location. Once the equipment is added then the attributes must be assigned that define the specifics about a given piece of equipment. To create equipment for an operation, perform the following steps.

1. Select equipment in listPane1.
2. Select database in listPane2.
3. Select the appropriate operation in listPane3.
4. Select add in listPane4.
5. Enter an Alpha-Numeric equipment name in the prompter window.



Please enter the name of the operation you wish to ad

aNewOperation added to Operations

Figure A1.5 Creation of an Operation.

2.3 Assignment of equipment attributes.

Assigning attributes to a given piece of equipment is nothing more than giving it a unique identity. These attributes are accessed during program execution to calculate production costs and to determine suitability of the equipment with the current user environment. These attributes will now be discussed in detail.

Accumulation

This attribute is described in two parts. The input accumulation refers to the accumulation state that a given piece of equipment can accept. Given a particular type of input accumulation a given machine will produce an output accumulation state which may or may not be the same as the input accumulation.

Attachment

The attachment attribute refers to a prominent attachment such as a shear attachment for a feller buncher or a grapple attachment on a skidder. This attribute variable is referenced during program execution to calculate production costs.

CostPMH

The cost of the operation per machine hour is an entered value that is independent of the user environment. This value must be known or estimated by the user.

Dbh

The dbh is described in two parts. The lowerbound value represents a piece of equipment's lower bound operating range. The upperBound value represents the upper bound operating range.

Ground

This attribute is described in two parts. The ground roughness refers to general topography and rocks etc. The ground firmness refers to soil texture. A scaling factor from 1 to 5 is used to define the two ranges. A factor of one being mild or soft while a factor of 5 being severe or hard.

HPClass

The horse power class is an entered value describing the horse power class of a given piece of equipment.

Product

The product is described in two parts. The first part is termed the input and refers to an input product. This input product refers to the output product from a piece of equipment that performed the previous operation. Given a particular input form a given piece of equipment will produce an output product.

Location

The location attribute refers to the location where the output product is located after processing. This may or may not be the same location as before processing. An example would be a bucking operation where the output location is the same after the bucking operation as before. With a skidding operation the location changes but the product state remains constant.

MaintCost

The maintenance cost per machine hour is an entered value. The value is entered in a prompter window during attribute assignment.

Operations

This attribute is one of the more important attributes, and can have significant impact on the solution space. A piece of equipment assigned to performing a given operation needs to know what possible operations can follow the operation it performs. An example would be manual felling, at the completion of the felling operation there most probably will be more than one possible way to proceed with harvesting.

Case 1: We might bunch the trees before skidding.

Case 2: We might skid whole trees by themselves.

Case 3: We might delimb the trees.

The following operations would then be included in the attribute operations.

Case 1: bunching.

Case 2: skidding.

Case 3: delimbing.

These operations are all considered valid moves from the operation felling. It is important to realize that this attribute determines how systems can be constructed. It is equally important to realize that the user is in complete control of this attribute. This feature gives the user the flexibility to add operations and equipment freely.

<u>ProdCost</u>

This attribute is a default variable that is entered as a constant when no regression/production equation is defined. It is a constant value that is not affected by the user environment. This feature allows the addition and incorporation of equipment without sufficient data.

<u>ProdEquation</u>

This attribute defines the name of a production equation. During program execution this equation will be utilized to provide an environmentally matched production cost for a given piece of equipment.

<u>Slope</u>

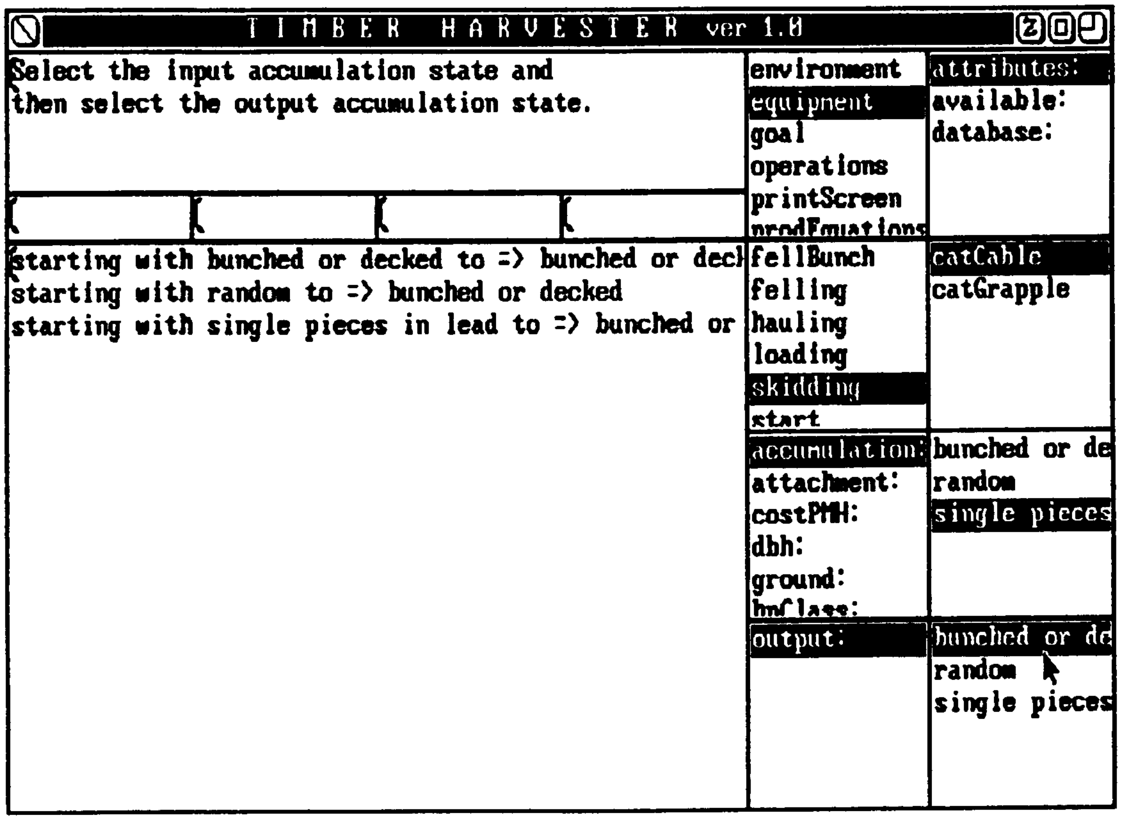
The slope attribute is a two part variable. The lowerBound defines the lowest slope for which a given piece of equipment can operate. The upperBound defines the largest positive slope for which the equipment can operate. Therefore the slope attribute defines the operating range for slope.

<u>Suspension</u>

This attribute is used to reflect a piece of equipment's ability to suspend one end during transport. This is a binary variable, if the equipment does not do any transportation the response should always be yes.

To set the attribute values for a piece of equipment the following sequence should be followed. (see figure A1.6)

1. Select the equipment option in listPane1.
2. Select the attribute option in listPane2.
3. Select the desired operation in listPane3.
4. Select the desired equipment in listPane4.
5. Select the desired attribute in listPane5. Then follow program prompts.


Select the input accumulation state and
then select the output accumulation state.

starting with bunched or decked to => bunched or decl
starting with random to => bunched or decked
starting with single pieces in lead to => bunched or

Figure A1.6 Setting Equipment Attributes.

2.4 Modification of existing equipment.

Modification of attributes for existing equipment is performed in the same manner as setting them for the first time, with one notable exception. If you want to change attributes that have more than a single value, such as location, product and accumulation you must reset all the values. An example might occur in the following.

Operation: Bucking.
Name:Processor1.

input product: whole tree => output product: bucked tree.

input product: delimbed tree => output product:bucked tree.

Lets say this piece of equipment exists. It is later decided that this piece of equipment CANNOT process whole trees. To change attribute you must perform the following.

1. Select the equipment option in listPane1.
2. Select the attribute option in listPane2.
3. Select the desired operation in listPane3.
4. Select the desired equipment in listPane4.
5. Select input in listPane5.

   Once input is selected both whole tree and delimbed tree are erased and you are left to redefine the input/output product relationships.

6. Select delimbed in listPane6.

7. Select output in listPane7.

8. Select bucked in listPane8.

3 - Creation and Modification of Support Databases.

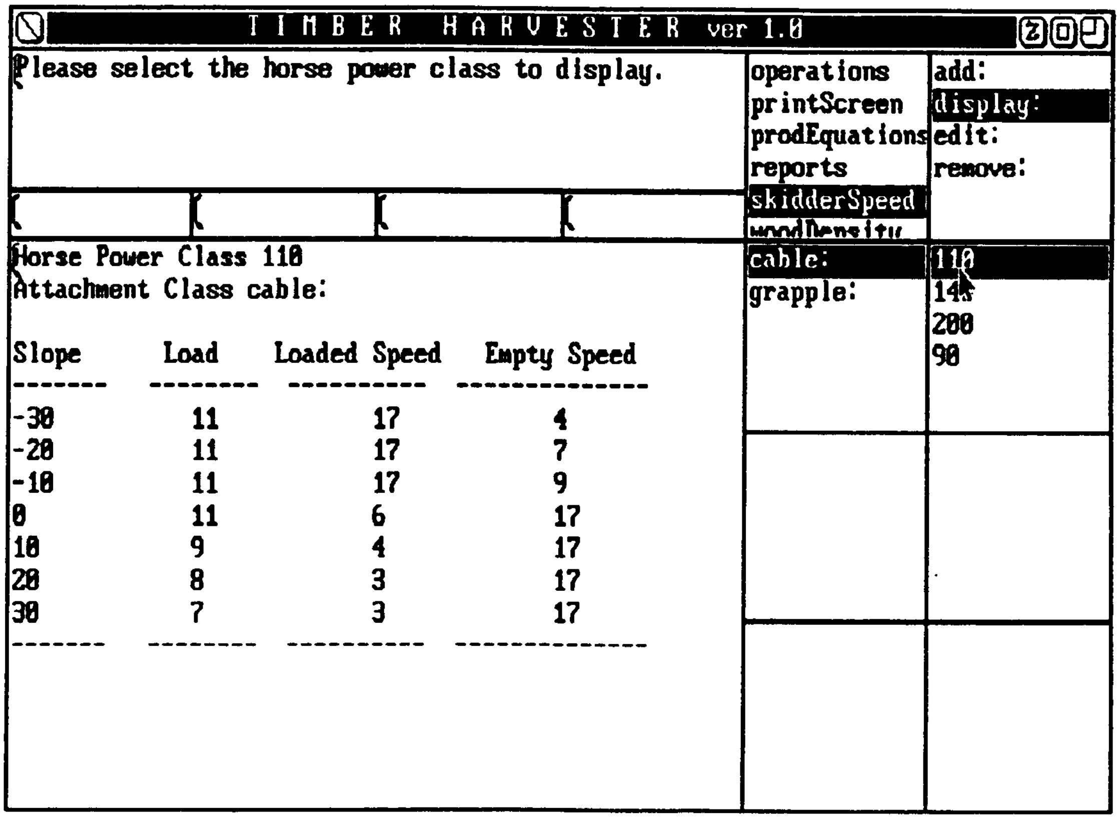
The Timber Harvester uses three support databases that are accessible by the entire system. These databases may be added to, deleted from or can just display the data they contain. Once they are created they should not need much up keep but should be monitored to insure accurate results.

3.1 Skidder Speeds and Loads Database.

This database contains the information needed when calculating the production cost for different kinds of skidding equipment. The database is organized by the type of attachment of the skidding equipment, this can be either a cable or a grapple. Three variables are entered for a given horse power class and slope percentage. A typical data table will look like the following table. (see figure A1.7)

HORSE POWER CLASS: 90
ATTACHMENT CLASS CABLE

SLOPE (%)	LOAD (1000 LB)	LOADED SPEED MPH	EMPTY SPEED MPH
-30	11.0	19.0	4.40
-20	11.0	19.0	5.90
-10	11.0	19.0	9.30
0	11.0	4.90	19.10
10	9.0	3.70	19.00
20	8.0	2.50	19.00
30	7.0	2.20	19.00



Please select the horse power class to display.

```
Horse Power Class 110
Attachment Class cable:

Slope      Load     Loaded Speed    Empty Speed
-------   --------  ----------    --------------
-30         11          17             4
-20         11          17             7
-10         11          17             9
0           11          6              17
10          9           4              17
20          8           3              17
30          7           3              17
-------   --------  ----------    --------------
```

Figure A1.7 Skidder Speeds and Loads.

3.1 a. Creating a new horse power class data table.

To create a new horse power class for a given attachment type, you should perform the following sequence.

1. Select skidderSpeed in listPane1.
2. Select add in listPane2.
3. Select attachment type in listPane3.
4. Enter a horse power class in the prompter window.

3.1 b. Displaying a data table.

To display a data table for a given attachment and horse power class, you should perform the following sequence.

1. Select skidderSpeed in listPane1.
2. Select display in listPane2.
3. Select attachment type in listPane3.
4. Select the horse power class in listPane4.

3.1 c. Editing a data table.

To edit an existing table, you should perform the following sequence.

1. Select skidderSpeed in listPane1.
2. Select edit in listPane2.

3. Select attachment type in listPane3.
4. Select the horse power class in listPane4.
5. Select slope percentage in listPane5.
6. Select variable to change in listPane6.

3.1 d. Removing a data table.

To remove an existing data table, you should perform the following sequence.

1. Select skidderSpeed in listPane1.
2. Select edit in listPane2.
3. Select attachment type in listPane3.
4. Select the horse power class in listPane4.

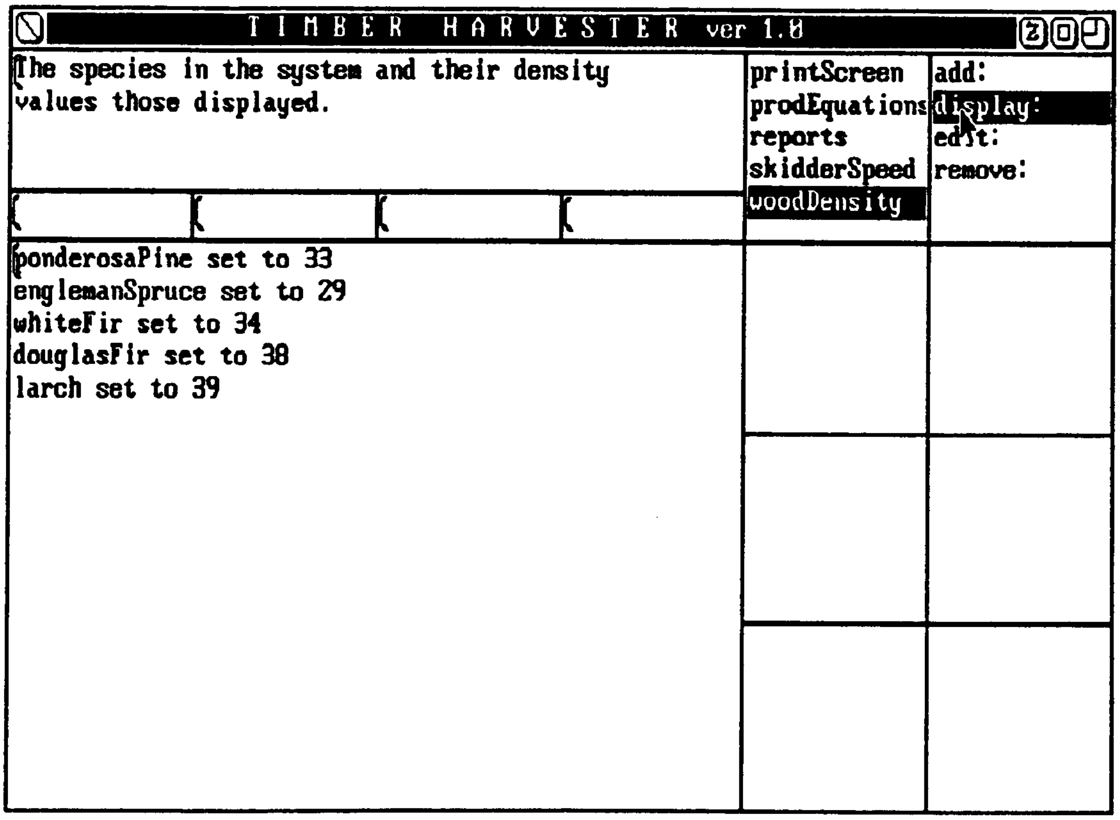
3.2 The Wood Density Database.

The wood density database contains information relating a density value to a particular species of tree. This database must contain the species and the associated value before a harvesting system can be found for a given species. The following database functions are available. (see figure A1.8)

3.2 a. Creating a new species.

To add a new species to the wood density database perform the following sequence.

1. Select woodDensity in listPane1.
2. Select add in listPane2.
3. Enter an Alpha species name in the prompter window.



The species in the system and their density values those displayed.

```
ponderosaPine set to 33
englemanSpruce set to 29
whiteFir set to 34
douglasFir set to 38
larch set to 39
```

Figure A1.8 WoodDensity Data.

3.2 b. Editing an existing species value.

To edit an existing species perform the following sequence.

1. Select woodDensity in listPane1.
2. Select edit in listPane2.
3. Select the species to edit in listPane3.
4. Enter the new value in the prompter window.

3.2 c. Displaying the existing species.

To display all existing species perform the following sequence.

1. Select woodDensity in listPane1.
2. Select display in listPane2.
3. Select the species to display in listPane3.

3.2 d. Removing an existing species.

To remove an existing species perform the following sequence.

1. Select woodDensity in listPane1.
2. Select remove in listPane2.
3. Select the species to remove in listPane3.

3.3 Production Equation Database.

The production system database is just a collection of production equation names. The corresponding equations need to added to the machine class after the name or names of the equations are added to production system equation database. Please refer to Chapter 6 for the specifics on adding the equations to the machine class. (see figure 8)

3.3 a. Adding a production equation name.

To add a production equation name perform the following sequence.

1. Select prodEquations in listPane1.
2. Select add in listPane2.
3. Enter an Alpha-Numeric name in the prompter window.

3.3 b. Displaying the current production equation names.

To display the currently available production equation names, perform the following sequence.

1. Select prodEquations in listPane1.
2. Select display in listpane2.

3.3 c. Removing an existing production equation name.

To remove a currently existing production equation name perform the following sequence.

1. Select prodEquations in listPane1.
2. Select remove in listPane2.
3. Select the production equation name in listPane3.

4 - Running the Equipment Matching Program

Before running the equipment matching algorithm, the user must specify the goal location and the goal product required.

To run the equipment matching algorithm the mouse should be placed in any list pane and the right button should be selected. This procedure should bring up a menu from which the selection match equipment should be chosen. The program will respond by relaying the goal location and the goal product selected. Then the program will begin the search routine. When the search process finds a system that ends in the required goal states and satisfies the user vector requirements, the system will be displayed on the screen. The display shows the state number, the total system cost and a list of equipment. The state number refers to the number of systems evaluated. This number is an easy way to attach an identity to a system and is used for this purpose. An example display might look like the following.

```
34:[112.56]->manual->d8cat->loader->truck.

State number = 34
System logging cost = 112.56
System = ->manual->d8cat->loader->truck.
```

Notice that the user is not explicitly told which operation each piece of equipment performs. To find out the specifics a user must record the state number and display the system specifics at the end of the run.

4.1 Selection of the goal states.

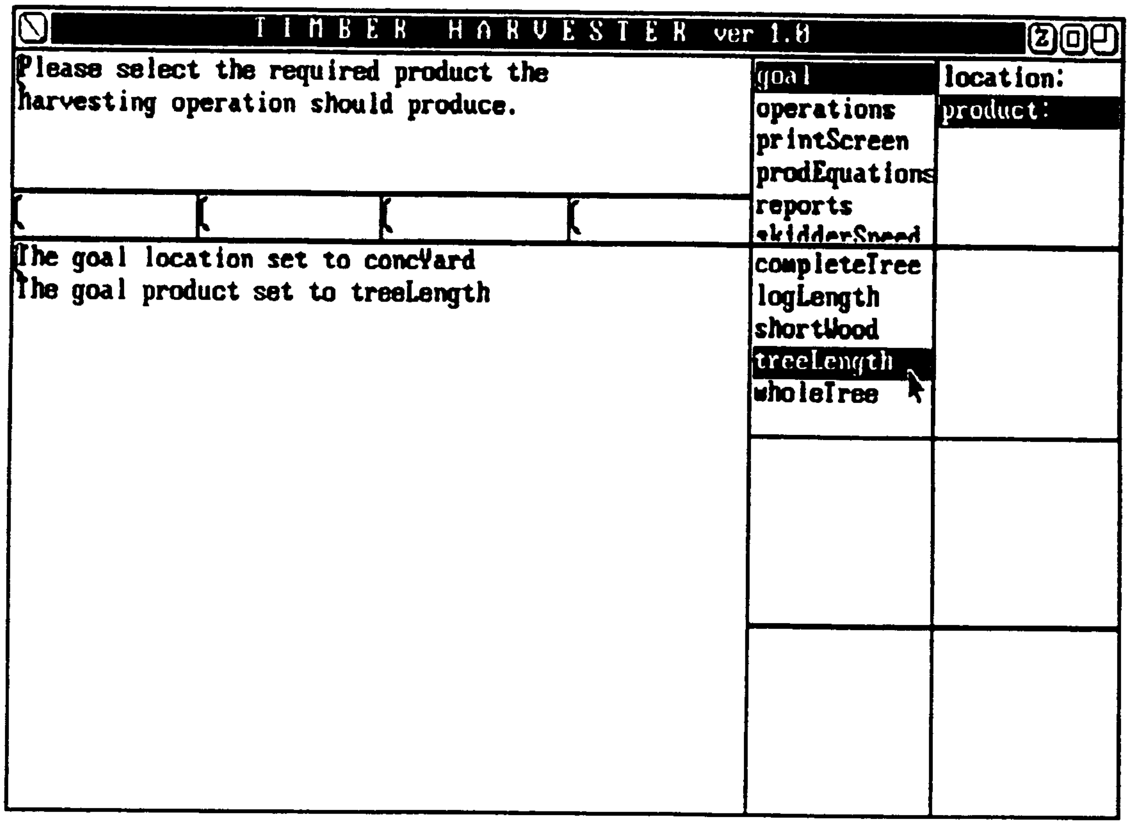
The selection of the goal operation must be selected before running the match equipment algorithm. To select a goal perform the following sequence. (see figure A1.9)

1. Select goal in listPane1.
2. Select the goal state to set in listPane2.
3. Set the goal state in listPane3.

4.2 Initiating the search algorithm.

To initiate the search algorithm perform the following sequence. (see figure 10)

1. Place mouse pointer in any list pane.
2. Select the right mouse button.
3. Select match equipment from the menu.



Please select the required product the harvesting operation should produce.

The goal location set to concYard
The goal product set to treeLength

Figure A1.9 Selection of Goals.

5 - Displaying and Interpreting the Results

The Timber Harvester comes with a feature to display or print the various systems that the search algorithm suggests.

5.1 Displaying feasible systems.

To display any system the Timber Harvest suggests as a feasible harvesting system, perform the following sequence.

1. Select the reports option in listPane1.
2. Select the display option in listpane2.
3. Select the state number of the system in listPane3.

5.2 Printing feasible systems.

To print any system the Timber Harvest suggests as a feasible harvesting system, perform the following sequence.

1. Select the reports option in listPane1.
2. Select the print option in listpane2.
3. Select the state number of the system in listPane3.

6 - Trouble Shooting Errors.

The Timber Harvester is virtually indestructible but in the unlikely event that the program halts, the following should section should get you back up and running. We cannot stress enough the importance of backing up the image file and the change.log file after and database change is made.

6.1 Program halts.

Program halts can occur when smalltalk expects information where none exits. Smalltalk will provide you with a walkback window which is covered in detail in the Smalltalk/V handbook.

6.2 Logic errors.

What to do when the system does not seem to providing the result you expect ? Usually a piece of equipment you created was not created like you think, in other words you probably did not enter the equipment characteristics as you expected to. Check all the pieces of equipment in the sequence you think should have been found. A handy way to do this is to pick goal states early in system creation and progress forward until you can isolate the equipment that is causing the problem.

APPENDIX 2.

PROGRAM LISTING

```
(OrderedCollection) subclass: #System
instanceVariableNames:        cost
classVariableNames:
poolDictionaries:


System class methods


new

  "    create a new initialized system "

^ super new initialize.


System methods


add: aMachine

   " Add aMachine to the receiver. Add the production
     cost/(a mapping factor) of adding a Machine to the cost
     of the receiver. Answer the receiver.  "

  " use the super class method add: to add the object
    machine. The variable super will override the System
    instance method add:. "

    aMachine compCumVector.
    super add: aMachine.
    cost := cost + (aMachine prodCost).

    ^self


contains:anOperation

   "  check to see if the system contains an operation "

    self do: [ :machine |
        (machine operation asSymbol = anOperation)
         ifTrue: [^true]].

^false


cost
     "  Answer the cost of the receiver. "

    ^cost
```

```
endsIn: aSymbol

        " Answer true if the last machine in the receiver is
          aMachine. Otherwise, answer false. "

    ^((self last operation asSymbol) = aSymbol )


endsInLocation: aSymbol

        " Answer true if the last machine in the receiver is
          aMachine. Otherwise, answer false. "

    ^((self last locationOut asSymbol) = aSymbol )


endsInProduct: aSymbol

        " Answer true if the last machine in the receiver is
          aMachine. Otherwise, answer false. "

    ^((self last outputProductState asSymbol) = aSymbol )


initialize

      "  Initialize the receiver: set its cost to 0.
         Answer the receiver. "

    cost := 0.

    ^self


length

      " Answer the length of the receiver: the number of
        Machines contained in it. "

    ^self size
```

```
(Dictionary) subclass:        #DictionaryOrderedCollection
instanceVariableNames:
classVariableNames:
poolDictionaries:


DictionaryOrderedCollection class methods


new

"  Answer a new DictionaryOrderedCollection object capable
   of containing 10 dictonaries. "

 ^ self new: 10


DictionaryOrderedCollection methods


addSymbolKey:aSymbol

   "  Add an OrderedCollection to
      DictionaryOrderedCollection. "


| tempOrderedCollection |

tempOrderedCollection := OrderedCollection new.
self at: aSymbol put:  tempOrderedCollection.

^self


addValueAt:akey with: aSymbol

 "  add an object to an OrderedCollection object within a
    dictionary object. "

(self at:akey) add: aSymbol.

 ^ self
```

removeSymbolKey:aKey

```
  " Answer the receiver without the key/value pair
    whose key equals aKey. If such a pair is not found
    report an error. "

self
    removeKey: aKey
     ifAbsent: [self errorAbsentKey].
^self
```

removeValueAt:aKey with:aSymbol

```
  "   remove a value at aKey. "

(self at:aKey)
    remove:aSymbol
     ifAbsent: [self errorAbsentKey].

^self
```

retrieveCollection:aSymbol

```
 " retrieve an OrderedCollection from a dictionary object."

 ^self at: aSymbol
```

```
(Dictionary) subclass:        #DictionaryDictionary
instanceVariableNames:
classVariableNames:
poolDictionaries:


DictionaryDictionary class methods


new

"  Answer a new DictionaryDictionary object capable
   of containing 10 dictonaries.  "

 ^ self new: 10


DictionaryDictionary methods


addSymbolKey:aSymbol

   "  Add a Dictionary to DictionaryDictionary. "

| tempDictionary |

tempDictionary := Dictionary new.
self at: aSymbol put: tempDictionary.

^self


addValueAt:akey with: aSymbol with:anObject

  " add an object to a dictionary object within a
    dictionary object. "

(self at:akey) at: aSymbol put:anObject.

 ^ self
```

removeSymbolKey:aKey

```
  "  Answer the receiver without the key/value pair
     whose key equals aKey. If such a pair is not found
     report an error. "

self
    removeKey: aKey
     ifAbsent: [self errorAbsentKey].
^self
```

removeValueAt:aKey with:aSymbol

```
   "  remove a value at aKey. "

(self at:aKey)
    removeKey:aSymbol
     ifAbsent: [self errorAbsentKey].

^self
```

retrieveDictionary:aSymbol

```
 "  retrieve a dictionary at aSymbol
    from a dictionary object .  "

 ^self at: aSymbol
```

retrieveValueAt:aKey with:aSymbol

```
  " retrieve an object from a dictionary
    within a dictionary object. "

 ^(self at:aKey) at:aSymbol.
```

```
(Object) subclass:                #Machine
instanceVariableNames:            name
                                  initialCost
                                  maintCost
                                  productionCost
                                  cumScoreVector
                                  outputProductState
                                  inputProductState
                                  productState
                                  predCompatibility
                                  mappingFactor
                                  succCompatibility
                                  succProcess
                                  operation
                                  slopeRange
                                  dbhRange
                                  suspention
                                  locationIn
                                  locationOut
                                  accumulationStateIn
                                  accumulationStateOut
                                  accumulation
                                  location
                                  type
                                  hpClass
                                  attachment
                                  productionEquation
                                  costPMH
                                  roughness
                                  firmness
classVariableNames:
poolDictionaries:


Machine class methods


new

 " create a new instance of machine "

^ super new initialize.
```

Machine methods

```
accumulationState

   " return accumulation dictionary "

^ accumulation


accumulationState:aKey with:aSymbol

" store accumulationStateIn at a key
  (accumulationStateOut) "

 accumulation at:aKey put:aSymbol.

^self


accumulationStateIn

    answer the accumulationStateIn

 ^ accumulationStateIn


accumulationStateIn:aSymbol

    answer the accumulationStateIn

  accumulationStateIn:= aSymbol.

  ^ self


accumulationStateOut

   " answer the accumulationStateOut "

 ^ accumulationStateOut


accumulationStateOut:aSymbol

  " set the accumulationStateOut "

 accumulationStateOut := aSymbol.

 ^ self
```

```
attachment

  " return the attachment type for a machine "

^ attachment


attachment:aString

 "  set the attachment type for a machine "

 attachment := aString.

^self


compCumVector

 " compute the cumulative Score Vector "

 " Temporary variable "

  | sum  |

  " Test to see if a machine causes butt split and the user
    specified that butt split was not allowed. "

(self attachment = #shear and: [(EnvironmentDictionary
at:#shearFelling:) = 'false'])
     ifTrue: [self cumScoreVector:#attachment with:0]
     ifFalse: [self cumScoreVector:#attachment with:1].

" Test to see if a machine does not suspend one
  end of the log and the user specified that one end must be
  suspended. "

(self suspention = 'false' and: [(EnvironmentDictionary
at:#suspention:) = 'true'])
     ifTrue: [self cumScoreVector:#suspention: with:0]
     ifFalse: [self cumScoreVector:#suspention: with:1].


 "  compute the cumScoreVector. "

((slopeRange at:#lowerBound:) <= (EnvironmentDictionary
at:#slope:)
 and: [ (slopeRange at:#upperBound:) >=
(EnvironmentDictionary at:#slope:)])
   ifTrue: [
    self cumScoreVector:#slope with:1]
   ifFalse: [
    self cumScoreVector:#slope with:0].

((dbhRange at:#lowerBound:) <= (EnvironmentDictionary
at:#dbh:)
 and: [ (dbhRange at:#upperBound:) >= (EnvironmentDictionary
at:#dbh:)])
   ifTrue: [
    self cumScoreVector:#dbh with:1]
   ifFalse: [
    self cumScoreVector:#dbh with:0].

((self firmness) >= (EnvironmentDictionary
at:#groundFirmness))
   ifTrue: [
    self cumScoreVector:#firmness with:1]
   ifFalse: [
    self cumScoreVector:#firmness with:0].

((self roughness) >= (EnvironmentDictionary
at:#groundRoughness))
   ifTrue: [
    self cumScoreVector:#roughness with:1]
   ifFalse: [
    self cumScoreVector:#roughness with:0].


 "  calculate the mappingFactor "



 ((cumScoreVector size) = (cumScoreVector occurrencesOf:1))
  ifTrue: [
   mappingFactor := 'pass']
  ifFalse: [
   mappingFactor := 'fail'].



 ^self


costPMH

   " return the instance variable costPMH  "

  ^costPMH
```

costPMH:aNumber

```
 "  set the instance variable costPMH to aNumber. "

  costPMH := aNumber.

^self
```

cumScoreVector

```
 " answer the cumScoreVector "

 ^cumScoreVector
```

cumScoreVector:aSymbol with:aNumber

```
 " store either a 1 for pass or 0 for fail at a Symbol in
   the cumScoreVector dictionary. "

 cumScoreVector at:aSymbol put:aNumber.

^self
```

dbhRange

```
 "  dbh range for a machine ,return the collection. "

^dbhRange
```

dbhRange:aString

```
 "   dbh range for a machine at a key (aString)
     return aNumber. "

^dbhRange at:aString
```

dbhRange:aString with:aNumber

```
"   dbh range for a machine at a key (aString)
    store aNumber. "

dbhRange at:aString put:aNumber.

^self
```

```
fbNoLevel

"  production cost derivation for operation feller bunch
   without leveling. "

   | treesPMH prodFt3PMH prodCostFt3 speedEmpty speedLoaded
     slope dbh merchantableVol unmerchantableVol woodDensity
     haulDistance acres treeVol hookUnhookDeckTime
     travelTime totalCycleTime |


   merchantableVol := EnvironmentDictionary at:#merchTrees:.
   treeVol := EnvironmentDictionary at:#treeVolume:.
   dbh :=  EnvironmentDictionary at:#dbh:.
   unmerchantableVol := EnvironmentDictionary
   at:#unmerchTrees:.

  treesPMH := 214.7 -  0.134 * (merchantableVol/0.4047) -
              53.1 * (unmerchantableVol / merchantableVol) -
              4.2 * (dbh * 2.54) + 29.7 * [4.36 - 0.12 *
              (dbh *2.54) - 0.00052 * (unmerchantableVol /
               0.4047)].

  prodFt3PMH := treesPMH * treeVol.

  prodCostFt3 :=  (self costPMH)/prodFt3PMH.

^prodCostFt3


firmness

   "   return the instance variable firmness. "

  ^firmness.


firmness:aNumber

"   set the instance variable firmness to aNumber. "

  firmness := aNumber.

^self
```

```
hpClass

 "    return the horse power class for a machine "

 ^hpClass


hpClass:aString

 "   set the horse power class for a machine "

 hpClass := aString.

^self


initialCost

"  return production cost "

^initialCost


initialCost:aNumber

 "  update initialCost cost "

initialCost := aNumber.

^self


initialize

"   initialize an instance of the class machine.
    answer the reciever. "

        dbhRange :=  Dictionary new.
        self dbhRange:#lowerBound: with:1.
        self dbhRange:#upperBound: with:40.
        slopeRange := Dictionary new.
        self slopeRange:#lowerBound: with:0.
        self slopeRange:#upperBound: with:30.
        succCompatibility := OrderedCollection new.
        succProcess := OrderedCollection new.
        productState := Dictionary new.
        location := Dictionary new.
        accumulation := Dictionary new.
        initialCost := 0.
        maintCost := 0.
        productionCost := 0.
        cumScoreVector := Dictionary new.
        attachment := 'notDefined'.
        costPMH := 0.
        firmness := 0.
        roughness := 0.
        hpClass := 'notDefined'.
        operation := 'notDefined'.
        productionCost := 0.
        suspention := 'notDefined'.
        productionEquation := 'notDefined'.

 ^self
```

inputProductState

```
 "  answer the inputProductState "

 ^ inputProductState
```

inputProductState:aSymbol

```
 "  set the inputProductState "

  inputProductState := aSymbol .

^ self
```

locationIn:aSymbol

```
  "  answer the locationIn "

  locationIn := aSymbol.

  ^ self
```

locationOut

```
 "  answer the locationOut "

^  locationOut
```

```
locationOut:aSymbol
  "  set the locationOut "
  locationOut := aSymbol.
  ^ self

locationState
   "  return location dictionary. "
^location

locationState:aKey with:aSymbol
 "   store locationOut at a key (locationIn) "
 location at:aKey put:aSymbol.
^self

maintCost
 "  answer the maintCost "
^maintCost

maintCost:aNumber
 "  Update the maintenance cost "
maintCost := aNumber.
^self

mappingFactor
 "   Return the mappingFactor "
^ mappingFactor
```

```
name

    "  Return the name of the equipment "

 ^ name

name:aSymbol

 "   Set the name of the equipment with aSymbol "
 name := aSymbol asString.
 ^ self

operation

 "   Return the name of the operation "

 ^ operation

operation:aSymbol

 "   Set the name of the operation with aSymbol "
 operation := aSymbol asString.
 ^ self

outputProductState

  "  answer the outputProductState "
 ^  outputProductState

outputProductState:aSymbol

  "  answer the outputProductState "
  outputProductState := aSymbol.
  ^ self
```

processCheck

```
   "  check for compatiable process material "


( self accumulationStateOut ) = nil
   ifFalse: [
  ( self outputProductState ) = nil
     ifFalse: [
    · ( self locationOut) = nil
       ifFalse: [ ^true]]].

^false
```

processMaterial:productSymbol with:locationSymbol with:accumulationSymbol

```
"  set input and output product states based from aSymbol
   that is returned from a machine in a system. "

 inputProductState := productSymbol.
 locationIn := locationSymbol.
 accumulationStateIn := accumulationSymbol.

 (productState includesKey:productSymbol)
  ifTrue: [
      outputProductState := productState at:productSymbol]
  ifFalse: [  outputProductState := nil].

  (location includesKey:locationSymbol)
  ifTrue: [
      locationOut := location at:locationSymbol]
  ifFalse: [  locationOut := nil].
  (accumulation includesKey:accumulationSymbol)
  ifTrue: [
      accumulationStateOut := accumulation
      at:accumulationSymbol]
      ifFalse: [ accumulationStateOut := nil].

 ^self
```

prodCost

```
 "  return production cost "

^productionCost
```

```
prodCost:aNumber
 "  update production cost "
productionCost := aNumber.
^self

productionEquation
"   return the value of productionEquation "
^ productionEquation

productionEquation:aSymbol
   " set the value of productionEquation "
    productionEquation := aSymbol.
^self

productState
 "    return the productState dictionary "
^productState

productState:aKey with:aSymbol
 "  store outputProductState at a key (inputProductState)"
 productState at:aKey put:aSymbol.
^self

resetAccumulation
" This method resets the Dictionary for Accumulation .
  called by attributesInput5. "

  accumulation := Dictionary  new.
^self
```

```
resetLocation

" This method resets the Dictionary for location .
  called by attributesInput5. "


  location:= Dictionary  new.

^self


resetOperations

" This method resets the orderedCollection for successor
  processes  "

  succProcess:= OrderedCollection new.

^self


resetProductState

" This method resets the Dictionary for the product state .
  called by attributesInput5. "


  productState := Dictionary  new.

^self


roughness

  "  return the instance variable roughness. "

  ^roughness.


roughness:aNumber

  "  set the instance variable roughness to aNumber. "

  roughness := aNumber.

^self
```

```
skidding

 "  production cost derivation for operation skid. "

   |loads load prodLbsPMH prodFt3PMH prodCostFt3 speedEmpty
speedLoaded slope merchantableVol woodDensity haulDistance
acres treeVol hookUnhookDeckTime travelTime totalCycleTime|

   slope := ((EnvironmentDictionary at:#slope:)
   printPaddedTo:1) asSymbol.
   merchantableVol := EnvironmentDictionary at:#merchTrees:.
   woodDensity :=  WoodDensity at:(EnvironmentDictionary
   at:#species:).
   haulDistance := EnvironmentDictionary at:#haulDistance:.
   acres := EnvironmentDictionary at:#acres:.
   treeVol := EnvironmentDictionary at:#treeVolume:.

 ( self attachment = #cable )
   ifTrue:[
   load := (SkidderSpeedCable at:(self hpClass) )
           retrieveValueAt:slope with:#load.

   speedEmpty := (SkidderSpeedCable at:(self hpClass) )
                 retrieveValueAt:slope with:#emptySpeed.

   speedLoaded := (SkidderSpeedCable at:(self hpClass) )
                  retrieveValueAt:slope with:#loadedSpeed].


  ( self attachment = #grapple )
   ifTrue:[
   load := (SkidderSpeedGrapple at:(self hpClass) )
           retrieveValueAt:slope with:#load.

   speedEmpty := (SkidderSpeedGrapple at:(self hpClass) )
                 retrieveValueAt:slope with:#emptySpeed.

   speedLoaded := (SkidderSpeedGrapple at:(self hpClass) )
                  retrieveValueAt:slope with:#loadedSpeed].


   loads := load/(merchantableVol*treeVol*woodDensity).

   hookUnhookDeckTime := 4.35 + ( 1.35 * loads ).

  ( self attachment = #grapple )
   ifTrue:[ hookUnhookDeckTime := 0.3 * hookUnhookDeckTime].

   travelTime :=
((haulDistance/88)*((1/speedLoaded)+(1/speedEmpty))).
```

```
   totalCycleTime := hookUnhookDeckTime + travelTime.

   prodLbsPMH := (load * 60)/totalCycleTime.

   prodFt3PMH :=  prodLbsPMH/woodDensity.

   prodCostFt3 :=  (self costPMH)/prodFt3PMH.

^prodCostFt3
```

slopeRange

```
 "   slope range for a machine return the collection. "

^slopeRange
```

slopeRange:aString

```
 "   slope range for a machine at a key (aString) "
     return aNumber.

^slopeRange at:aString
```

slopeRange:aString with:aNumber

```
"   update the slope range for a machine at a key (aString)
    store aNumber. "
```

slopeRange at:aString put:aNumber.

```
^self
```

succCompatibility

```
"   return a compatibility OrderCollection which represents
    compatible successor machines. "

^succCompatibility
```

succCompatibility:aSymbol

```
 " add aSymbol ( an equipment name ) to compatibility  list.
   Do this only if the name does not already exist."

(succCompatibility includes:aSymbol)
  ifFalse: [
succCompatibility add:aSymbol].

^self
```

succProcess

```
"   return a compatibility OrderCollection which represents
    compatible successor processes."

^succProcess
```

succProcess:aSymbol

```
"   add aSymbol ( a process name ) to compatibility list.
    Do this only if the name does not already exist."

(succProcess includes:aSymbol)
  ifFalse: [
succProcess add:aSymbol].

^self
```

suspention

```
    " Return the true/false value of suspention "

^ suspention
```

suspention:aString

```
 "  set the true/false value of suspention "

suspention:= aString.

^ self
```

```
(Object) subclass:              #SystemFinder
instanceVariableNames:          statesGenerated
                                startingSystem
                                goal
                                queue
                                goalLocation
                                goalProduct
classVariableNames:
poolDictionaries:


SystemFinder class methods


SystemFinder methods


goal

    " return the current goal state "
^ goal

goal:aSymbol

 "   return the current goal state "
 goal := aSymbol.

^ self

goalLocation

 "   return the current goalLocation  state "
^ goalLocation

goalLocation:aSymbol

 "   set the current goalLocation state "
 goalLocation := aSymbol.

^ self
```

```
goalProduct

 "   return the current goalProduct state "

^ goalProduct

goalProduct:aSymbol

 "   set the current goalProduct state "

 goalProduct := aSymbol.

^ self

nextState
      " Remove the first remaining state (aSystem) from the
        queue and answer it. "

    ^queue removeFirst

queue
 "  return the size of the queue. "

 | tempQueueSize |

 tempQueueSize := queue .

^  tempQueueSize
```

```
startSystem

    "    Create a queue to hold partial solution systems and
         answer the receiver. "

    Temporary variables

     | aNewState |

    self initializeQueue.

    (OperationsAvailable at:#start) do: [:machine |
    aNewState := System new .
    machine processMaterial:#completeTree with:#standing
    with:#random.
    queue add:(aNewState add:machine)].
    statesGenerated := 1.


  "  send message showStartSystem: with aBestFirstSelection
     as the parameter. "

    Timber showStartSystem:self.

    ^self
```

```
(SystemFinder) subclass:      #BestFirstSelection
instanceVariableNames:
classVariableNames:
poolDictionaries:


BestFirstSelection class methods


BestFirstSelection methods


enqueue: anOrderedCollection

      " Add the states contained in anOrderedCollection to
        the queue. In this case, add the new states one at a
        time to the queue. Being a SortedCollection, it will
        Maintain the states in ascending order of cost.
        Answer the receiver. "

    queue addAll: anOrderedCollection.

    ^self


expandState: aSystem


      "  Expand the state aSystem by finding all possible
         states (Systems) that can be reached from it. Print
         the states as they are generated. aSystem is an
         OrderedCollection of machines. "

     Temporary variables

    | successors  successorState |

    successors := OrderedCollection new.

  " Send feasibleMachineFrom: message to a
    BestFirstSelection and do a block of code on each object
    (aMachine) in the OrderedCollection feasibleMachines. "

    (self feasibleMachinesFrom: aSystem) do: [ :aMachine |
        successorState := System new.
        successorState addAll: aSystem.
        successorState add: aMachine .
        statesGenerated := statesGenerated + 1.

        Timber showStatesCount:statesGenerated.
        Timber showQueueCount:queue.
" Send messages to the object Timber to print the
  sucessorState and statesGenerated in a textpane. "

  (( successorState endsInLocation: goalLocation) and:
  [successorState endsInProduct: goalProduct])
      ifTrue: [

     Timber showStatesGenerated:statesGenerated.
     Timber showSuccessorState:successorState.
     Timber addToReportsAt:(statesGenerated storeString)
     with:successorState.
      ].

     " Add the successorState to the successors and when
       complete, return the successors. "

     successors add: successorState ].

 ^successors
```

feasibleMachinesFrom:aSystem

```
    " Answer an OrderedCollection containing the machines
      that the receiver could feasibly move to from the
      last machine in aSystem. "

   Temporary variables

   |feasibleMachines lastMachine candidateMachine
   candidates tempMachine|


 " lastMachine is the machine to be expanded.
   Create an OrderedCollection to hold feasible machines. "


   lastMachine := aSystem last.
   feasibleMachines := OrderedCollection new.
   candidates := OrderedCollection new.

 "  Now check to see what operations follow the last
    machine operation. Then process the equipment in for
    those operations. "


   (lastMachine succProcess) do: [:operation|
    (aSystem contains:operation asSymbol)
     ifFalse:[
     (OperationsAvailable at:(operation asSymbol)) do:
     [:equipment|
      tempMachine := equipment deepCopy.
```

```
          tempMachine processMaterial: (lastMachine
          outputProductState)
          with:(lastMachine locationOut) with:(lastMachine
          accumulationStateOut).
           (tempMachine processCheck)
            ifTrue:[
             ((tempMachine productionEquation = 'notDefined')
              or: [equipment productionEquation =
               #notDefined])
               ifFalse: [ tempMachine prodCost:
                (tempMachine perform:(tempMachine
                  productionEquation))].
                  (candidates includes:tempMachine )
                  ifFalse: [candidates add:tempMachine ]]]]].

     " For each candidate machine map the machine to the user
       environment vector (EnvironmentDictionary). Then check
       the mapping factor for a pass criterion. If candidate
       passes then add machine to feasible machine
       OrderedCollection. "

   candidates do: [:candidateMachine |  candidateMachine
      compCumVector.
     ((candidateMachine mappingFactor) = 'pass')
       ifTrue: [ feasibleMachines add:candidateMachine]].


^feasibleMachines


initializeQueue

" intializeQueue so its an SortedCollection. The queue is
  sorted by the system cost which is the accumlated
  adjusted production cost for each machine. "

   queue := SortedCollection sortBlock: [ :system1
           :system2 | (system1 cost) <= ( system2 cost) ].

 ^self
```

search

```
** This is the main processing loop for the search
algorithm. **


      " While there are states left in the queue, expand
        them and add their successor states to the queue. If
        the goal is found, report success. Otherwise, report
        failure. Answer the receiver. "

    | aState successors |

    [ queue isEmpty ] whileFalse: [
        aState := self nextState.

    "  Check for goal state by sending endsIn: message with
       the instance variable goal as an argument. "

        ((aState endsInLocation: goalLocation) and: [aState
        endsInProduct: goalProduct])
         ifTrue: [
            Timber reportSuccess:aState.
            ^self ].

      " If the goal was not found, then send the message
        expandState: to a BestFirstSelection with the
        current state as aState to expand. "

        successors := self expandState: aState.
        self enqueue: successors ].

    "  If queue is empty then report failure ."

     Timber reportFailure.

    ^self
```

```
(Object) subclass:              #TimberSimPane
instanceVariableNames:          inputString
                                inputPane1
                                inputPane2
                                inputPane3
                                inputPane4
                                inputPane5
                                inputPane6
                                inputPane7
                                inputPane8
                                inputPane9
                                replyStream
                                list1Collection
                                list2Collection
                                list3Collection
                                list4Collection
                                list5Collection
                                list6Collection
                                list7Collection
                                list8Collection
                                choice1
                                choice2
                                choice3
                                choice4
                                choice5
                                choice6
                                choice7
                                choice8
                                operations
                                stateCollection
                                stateCount
                                stateGenWindow
                                queueCount
                                queueCntWindow
classVariableNames:
poolDictionaries:
```

TimberSimPane class methods

new

```
  create a new instance of TimberSimPane

^ super new initialize.
```

```
start

"   Open a TimberSimPane to begin the Timber Harvester "

Timber := TimberSimPane new.
EquipmentSelector:= BestFirstSelection new.
Timber openOn: 'Welcome to the Timber Harvester.

 1) Input your enviroment charactistics.
 2) Match Equipment with enviroment.'.

^self


TimberSimPane methods


acresPrompt

 "  Open a prompter window to set the stand size in acres. "

 Temporary variables

|tempVal temp|


"  get user input for stand size in acres "

tempVal := Prompter prompt: 'Stand size in acres.'
defaultExpression: '' point: 200 @ 250.

  Determine the appropriate operation to locate an object.
   Once the object is located, update the Stand size
information.


     EnvironmentDictionary at:choice3 put:tempVal.
     inputString := (inputString, choice3 printString, ' set
     to      ', tempVal printString,   '\') withCrs.
     self changed: #reply2.


^self
```

```
addToAvailable:tempkey

  "    add to operationsAvailable "

        | temp |
              (OperationsAvailable includesKey:tempkey)
              ifFalse: [  OperationsAvailable
               addSymbolKey:tempkey].
              OperationsAvailable addValueAt:tempkey
              with:choice5 with:(Operations
              retrieveValueAt:tempkey with:choice5).
              inputString := (inputString, choice5
              printString,' added to available ',choice3
              printString, 'equipment.', '\') withCrs .
              self changed: #reply2.

^ self


addToDatabase

  "  add equipment to database "

    | temp tempkey |

    temp:= Prompter prompt: 'equipment name'
            defaultExpression: '#' point: 200 @ 150.
            tempkey := temp.
                  temp := Machine new.
                  temp operation:choice3.
                  temp name:tempkey.
                  Operations addValueAt:choice3
                  with:tempkey with:temp.
                  inputString := (inputString, tempkey
                  printString, ' added to ', choice3
                  printString, '\') withCrs .
                  self changed: #reply2.

 ^ self
```

addToDatabase:aSymbol

```
    "   add equipment to database "

    | temp tempkey  |

    temp:= Prompter prompt: 'equipment name'
             defaultExpression: '#' point: 200 @ 150.
             tempkey := temp.
                   temp := Machine new.
                   temp operation:choice3.
                   temp name:tempkey.
                   Operations addValueAt:choice3
                   with:tempkey with:temp.
                   EquipmentAll at:tempkey put:temp.
                   inputString := (inputString, tempkey
                   printString, ' added to ', choice3
                   printString, '\') withCrs .
                   self changed: #reply2.

 ^ self
```

addToReportsAt:aKey with:aSystem

```
 " add a system to stateCollection for reports. "


stateCollection at:aKey put:aSystem.

^self
```

addToWoodDensity

```
 "  Open a prompter window to add to wood density species."

" Temporary variables "

|tempVal temp|


 " get user input for the specie name "

temp := Prompter prompt: '    specie name      '
defaultExpression: '#' point: 200 @ 250.


     (temp = nil)
     ifFalse: [
     WoodDensity at:temp put:38.
     inputString := (inputString, temp printString, ' added
     and set to 38 ',   '\') withCrs.
     self changed: #reply2].


^self


attributesInput4

"  evaluate attributes for input pane 4. "

   list5Collection:=  PaneSelectors
   retrieveCollection:#attributes.
   replyStream := Reply1Dictionary at: #attributes.
   self changed: #reply1 .

 ^ self


attributesInput5

 "evaluate attribute input for input pane 5."

 (choice5 == #suspention:)
  ifTrue:[
       list6Collection := PaneSelectors
       retrieveCollection:#booleanCollection.
       replyStream := Reply1Dictionary at: #equipSuspention.
       self changed: #reply1].

 (choice5 == #slope:)
  ifTrue:[
    list6Collection := PaneSelectors
    retrieveCollection:#bounds.
    replyStream := Reply1Dictionary at: #equipmentSlope.
    self changed: #reply1].

 (choice5 == #dbh:)
  ifTrue:[
    list6Collection := PaneSelectors
    retrieveCollection:#bounds.
    replyStream := Reply1Dictionary at: #equipmentDbh.
    self changed: #reply1].


 (choice5 == #ground:)
  ifTrue:[
    list6Collection := PaneSelectors
    retrieveCollection:#groundSpecifics:.
    replyStream := Reply1Dictionary at: #ground.
    self changed: #reply1].
 (choice5 == #maintCost:)
  ifTrue:[
       replyStream := Reply1Dictionary at: #promptMessage.
       self changed: #reply1.
       self maintPrompt].

 (choice5 == #input:)
  ifTrue:[
    (Operations retrieveValueAt:choice3 with:choice4)
    resetProductState.
    list6Collection := PaneSelectors
    retrieveCollection:#inputProduct.
    replyStream := Reply1Dictionary at: #input.
    self changed: #reply1].

 (choice5 == #location:)
  ifTrue:[
    (Operations retrieveValueAt:choice3 with:choice4)
    resetLocation.
    list6Collection := PaneSelectors
    retrieveCollection:#location.
    replyStream := Reply1Dictionary at: #location.
    self changed: #reply1].

 (choice5 == #accumulation:)
  ifTrue:[
    (Operations retrieveValueAt:choice3 with:choice4)
    resetAccumulation.
    list6Collection := PaneSelectors
    retrieveCollection:#accumulation.
    replyStream := Reply1Dictionary at: #accumulation.
    self changed: #reply1].

 (choice5 == #attachment:)
  ifTrue:[
    list6Collection := PaneSelectors
    retrieveCollection:#attachments.
    replyStream := Reply1Dictionary at: #attachment.
    self changed: #reply1].

(choice5 == #hpClass:)
  ifTrue:[
 replyStream := Reply1Dictionary at: #hpClass.
 self changed: #reply1.
 self hpPrompt].

(choice5 == #operations:)
  ifTrue:[
    (Operations retrieveValueAt:choice3 with:choice4)
    resetOperations.
    list6Collection := Operations.
    replyStream := Reply1Dictionary at:
```

```
    #equipmentOperations.
    self changed: #reply1].

(choice5 == #prodEquation:)
  ifTrue:[
    list6Collection := PaneSelectors
    retrieveCollection:#prodEquations.
    replyStream := Reply1Dictionary at:
    #equipmentProdEquation.
    self changed: #reply1].

 (choice5 == #prodCost:)
  ifTrue:[
       replyStream := Reply1Dictionary at: #promptMessage.
       self changed: #reply1.
       self prodPrompt].

 (choice5 == #costPMH:)
  ifTrue:[
       replyStream := Reply1Dictionary at: #promptMessage.
       self changed: #reply1.
       self costPMHPrompt].

^self


attributesInput6:aString

 "    attribute selectors for input pane 6. "

        | temp tempkey |

    ( choice5 == #suspention:)
      ifTrue: [
        choice6 := aString.
        replyStream := Reply1Dictionary at: #promptMessage.
        tempkey := choice4.
         ((Operations at:choice3) at:tempkey)
          suspention:choice6.
          inputString := (inputString, choice4 printString,
          ' for ', choice5 printString, ' set to ',
           choice6 printString,\') withCrs.
           self changed: #reply2].

 ( choice5 == #operations:)
      ifTrue: [
        choice6 := aString asSymbol.
        tempkey := choice4.
         (Operations retrieveValueAt:choice3 with:tempkey)
          succProcess:choice6.
          inputString := (inputString, choice6 printString ,
          ' added to succesor operations', '\') withCrs.
          self changed: #reply2].

 ( choice5 == #slope:)
    ifTrue: [
     ( choice6 == #lowerBound:)
       ifTrue: [
       list7Collection := PaneSelectors
       retrieveCollection:#slopeCollection.
       replyStream := Reply1Dictionary
       at:#equipmentSlopeLower.
       self changed: #reply1].

     ( choice6 == #upperBound:)
       ifTrue: [
       list7Collection := PaneSelectors
       retrieveCollection:#slopeCollection.
       replyStream := Reply1Dictionary
       at:#equipmentSlopeUpper.
       self changed: #reply1]].

( choice5 == #dbh:)
      ifTrue: [
       (choice6 == #lowerBound:)
        ifTrue: [
        list7Collection := PaneSelectors
        retrieveCollection:#dbhCollection.
        replyStream := Reply1Dictionary
        at:#equipmentDbhLower.
        self changed: #reply1].

       (choice6 == #upperBound:)
        ifTrue: [
        list7Collection := PaneSelectors
        retrieveCollection:#dbhCollection.
        replyStream := Reply1Dictionary
        at:#equipmentDbhUpper.
        self changed: #reply1]].

 ( choice5 == #input:)
      ifTrue: [
        choice6 := aString asSymbol.
        list7Collection := PaneSelectors
        retrieveCollection:#output:.
        replyStream := Reply1Dictionary at:#input.
        self changed: #reply1].

( choice5 == #accumulation:)
      ifTrue: [
        choice6 := aString asSymbol.
        list7Collection := PaneSelectors
        retrieveCollection:#output:.
```

```
          replyStream := Reply1Dictionary at:#accumulation.
          self changed: #reply1].

( choice5 == #location:)
        ifTrue: [
          choice6 := aString .
          list7Collection := PaneSelectors
          retrieveCollection:#output:.
          replyStream := Reply1Dictionary at:#location.
          self changed: #reply1].

( choice5 == #attachment:)
        ifTrue: [
        ( Operations retrieveValueAt:choice3 with:choice4)
         attachment:choice6.
         inputString := (inputString, 'attachment set to
         ',choice6 printString ,'\') withCrs.
         self changed: #reply2].

( choice5 == #prodEquation:)
        ifTrue: [
        (Operations retrieveValueAt:choice3 with:choice4)
         productionEquation:choice6.
         inputString := (inputString, 'production equation set
         to ',choice6 printString ,'\') withCrs.
         self changed: #reply2].


( choice5 == #ground:)
        ifTrue: [
     (choice6 == #firmness:)
        ifTrue: [
          choice6 := aString asSymbol.
          list7Collection := PaneSelectors
          retrieveCollection:#oneToFive:.
          replyStream := Reply1Dictionary
          at:#equipmentFirmness.
          self changed: #reply1].

   (choice6 == #roughness:)
       ifTrue: [
          choice6 := aString asSymbol.
          list7Collection := PaneSelectors
          retrieveCollection:#oneToFive:.
          replyStream := Reply1Dictionary
          at:#equipmentRoughness.
          self changed: #reply1]].


^ self
```

attributesInput7

```
" evaluate attribute input for input pane 7."


 ((choice7 == #output:) and: [choice5 == #input:])
      ifTrue: [
        list8Collection := PaneSelectors
        retrieveCollection:#outputProduct.
        replyStream := Reply1Dictionary at: #input.
        self changed: #reply1 ].

 ((choice7 == #output:) and: [choice5 == #location:])
      ifTrue: [
        list8Collection := PaneSelectors
        retrieveCollection:#location.
        replyStream := Reply1Dictionary at: #location.
        self changed: #reply1 ].

 ((choice7 == #output:) and: [choice5 == #accumulation:])
      ifTrue: [
        list8Collection := PaneSelectors
        retrieveCollection:#accumulation.
        replyStream := Reply1Dictionary at: #accumulation.
        self changed: #reply1 ].

 (choice5 == #dbh:)
      ifTrue: [
        self dbhSetBounds:choice7 with:choice6.
        self changed: #reply1 ].

(choice5 == #ground:)
      ifTrue: [
        self groundSetMaxes:choice6 with:choice7.
        self changed: #reply1 ].

 (choice5 == #slope:)
      ifTrue: [
        self slopeSetBounds:choice7 with:choice6.
        self changed: #reply1 ].
^ self
```

availableInput4

```
" evaluate available equipment for input pane 4."


  ( choice4 == #display:)
      ifTrue: [
      list5Collection := (OperationsAvailable at:choice3).
      replyStream := Reply1Dictionary at:#availableDisplay.
```

```
        self changed: #reply1].

    ( choice4 == #remove:)
        ifTrue: [

        list5Collection := (Operations at:choice3).
        replyStream := Reply1Dictionary at:#availableRemove.
        self changed: #reply1].

     (choice4 == #add:)
        ifTrue: [
        list5Collection := (Operations at:choice3).
        replyStream := Reply1Dictionary at:#availableAdd.
       self changed: #reply1].

^self


availableInput5:tempkey

"  evaluate available input for input pane 5. "

   ( choice4 == #add:)
       ifTrue: [
        self addToAvailable:tempkey].

   ( choice4 == #display:)

       ifTrue: [
        self displayMachine:(OperationsAvailable
        retrieveValueAt:tempkey with:choice5)].

   ( choice4 == #remove:)

       ifTrue: [
       self removeFromAvailable:tempkey].

^ self


columnDisplay:aString

| inputStream outputStream tempNumber wordNumber word
inputStringTemp |

     tempNumber := 0.
     wordNumber := 0.
     inputStream := ReadStream on:aString.
     outputStream := WriteStream on:
       (String new: inputStringTemp size + 2).

          [inputStream atEnd]
          whileFalse: [
            word := inputStream nextWord.
             wordNumber := wordNumber + 1.
              (wordNumber = 1)
              ifTrue: [ tempNumber := 16 - (word size)].
              (wordNumber = 2)
              ifTrue: [ tempNumber := 12 - (word size)].
              (wordNumber = 3)
              ifTrue: [ tempNumber := 15 - (word size)].
              (wordNumber = 4)
              ifTrue: [ tempNumber := 14 - (word size)].
              (wordNumber = 5)
              ifTrue: [  tempNumber := 8 - (word size)].
              (wordNumber = 6)
              ifTrue: [  tempNumber := 3 - (word size)].
              (wordNumber = 7)
              ifTrue: [  tempNumber := 8 - (word size)].
              outputStream nextPutAll:word.
              1 to:tempNumber by:1  do: [ :number |
              outputStream space]].


             inputStringTemp := outputStream contents.

   ^ inputStringTemp
```

columnDisplaySkidderData:aString

```
| inputStream outputStream tempNumber newWord
inputStringTemp char|


     tempNumber := 0.
     inputStream := ReadStream on:aString.
     outputStream := WriteStream on:
       (String new: inputStringTemp size + 2).

          [inputStream atEnd]
             whileFalse: [
               char := inputStream next.

             (char asciiValue = 32)
              ifTrue: [ inputStream skip:1.
              (newWord = 1)
               ifTrue: [
               1 to:(13 - tempNumber) by:1  do: [ :number |
               outputStream space].
               newWord := 0.
               tempNumber := 0.]]
               ifFalse: [
                tempNumber :=  tempNumber + 1.
```

```
                    newWord := 1.
                    outputStream nextPut:char ]].



               inputStringTemp := outputStream contents.

   ^ inputStringTemp
```

convertStringToNumber:aString

```
"   convert either negative or positive string
    representations to either
    negative or positive numbers. "


    | number neg temp tempkey firstCharacter newString
choice |

        number := 0.
        newString := aString.
        neg := 0.
        firstCharacter := ( aString at:1).

          (firstCharacter isDigit)
             ifFalse: [
              (firstCharacter asciiValue = 45)
              ifTrue: [
                neg := 1.
                newString := ( aString select: [ :character |
                character isDigit])]].

                 newString do: [ :character |
                 number:= number * 10 + character digitValue.
                 choice:= number ].

                 (neg = 1)
                   ifTrue: [choice := choice * -1].

 ^choice
```

costPMHPrompt

```
"   a prompter to set the cost per machine hour cost. "

|tempVal choiceSelector tempkey temp|
tempkey := choice4.
choiceSelector:= (choice5)asSymbol.
tempVal := Prompter prompt: 'Cost per Machine Hour.'
```

```
defaultExpression: '' point: 200 @ 250.
        (Operations retrieveValueAt:choice3 with:tempkey)
        perform: choiceSelector with:tempVal.
        inputString := (inputString, choice4 printString, '
        for ', choice3
        printString, ' at ', choice5 printString,' set to ',
        tempVal printString, '\') withCrs.
        self changed: #reply2.
^self
```

createEquationSymbol

```
"   Open a prompter window to get symbol name."

" Temporary variables "

|tempVal temp|


 " get user input for the symbol name"

tempVal := Prompter prompt: 'Equation Name.'
defaultExpression: '#' point: 200 @ 250.

"  Determine the appropriate operation to locate an object.
   Once the object is located, update the symbol
   information. "

    (tempVal == nil)
     ifFalse: [
     PaneSelectors addValueAt:#prodEquations with:tempVal.
     inputString := (inputString,tempVal printString, '
     added to production equation names. ',   '\') withCrs.
     self changed: #reply2].


^self
```

createOperation

```
"     create a new operation. "

 | temp tempkey newAvailableOperation |

   temp:= Prompter prompt: 'operation name'
             defaultExpression: '#' point: 200 @ 150.
             tempkey := temp.
            (tempkey = nil)
             ifFalse: [
             inputString := (inputString, tempkey
                printString,' added to Operations' asSymbol
                printString, '\') withCrs .
                self changed: #reply2.
                Operations addSymbolKey:tempkey.
                OperationsAvailable addSymbolKey:tempkey].


        ^self

                changed: #input3.
```

databaseInput4

```
" evaluate database input for input pane 4."

 ( choice4 == #remove:)
      ifTrue: [
              list5Collection := (Operations at:choice3).
              replyStream := Reply1Dictionary
              at:#databaseRemove.
              self changed: #reply1].


 (choice4 == #display:)
      ifTrue: [
              list5Collection := (Operations at:choice3).
              replyStream := Reply1Dictionary
              at:#databaseDisplay.
              self changed: #reply1].

    (choice4 == #add:)
              ifTrue: [
              replyStream := Reply1Dictionary at:#equipAdd.
              self changed:#reply1.
              list4Collection:= (Operations at:choice3).
              self addToDatabase].

 ^ self


databaseInput5:tempkey

 evaluate database input for input pane 5.

  ( choice4 == #display:)
        ifTrue: [
        self displayMachine:(Operations
retrieveValueAt:tempkey with:choice5)].
          ( choice4 == #remove:)
                ifTrue: [
               self removeFromDatabase:tempkey].
```

```
^ self


dbhPrompt

 " Open a prompter window to set the dbh weight factor.
   Temporary variables "

|tempVal choiceSelector tempkey temp|

tempkey := choice4.       " store choice4 to tempkey"
" concatenate choice5 with #with: as a sysmbol"

choiceSelector:= (choice5,#with:)asSymbol.

"  get user input for the performance factor "

tempVal := Prompter prompt: 'DBH Performance Factor.'
defaultExpression: '' point: 200 @ 250.

 " Determine the appropriate operation to locate an object.
   Once the object is located, update the dbh information. "


    ((Operations at:choice3) at:tempkey) perform:
    choiceSelector with:choice6 with:tempVal.
    inputString := (inputString, choice4 printString, ' for
    ', choice5 printString,
    ' at ', choice6 printString, ' set to ', tempVal
    printString, '\') withCrs.
    self changed: #reply2.


^self


dbhSetBounds:aString with:aKey

 "    set the upper or lower bounds on dbh. "

       (Operations retrieveValueAt:choice3 with:choice4)
       dbhRange:aKey  with:(self
       convertStringToNumber:aString).
       inputString := (inputString, 'The dbh ', choice6
       printString, ' for ', choice4
       printString, ' is set to ', choice7 asSymbol
       printString,'\') withCrs.
       self changed: #reply2.
```

```
displayEnvironment

"   display the current environment in text pane 2. "

   ((EnvironmentDictionary keys) asSortedCollection) do:
   [ :akey | inputString := (inputString, akey printString,
   ' set to ', (EnvironmentDictionary at:akey) printString,
   '\') withCrs.
   self changed: #reply2] .


^ self


displayFactors

 "  display the current performance factors "

"   Temporary variable "

 |tempMachine|

 "Determine what operation the machine belongs to
  and what attribute is selected. Then display the
  key value pairs using the instance variable inputString."

  (choice5 == #slope:)
  ifTrue: [
    tempMachine := (Operations at:choice3) at:choice4.
    tempMachine slopeDictionary keysDo: [ :akey |
    inputString := (inputString, akey printString, ' set to
    ', (  tempMachine slopeDictionary at:akey) printString,
     '\')  withCrs.
    self changed: #reply2]].

 (choice5 == #dbh:)
  ifTrue: [
    tempMachine := (Operations at:choice3) at:choice4.
    tempMachine dbhDictionary keysDo: [ :akey |
    inputString := (inputString, akey printString, ' set to
    ',(  tempMachine dbhDictionary at:akey) printString,
    '\') withCrs.
    self changed: #reply2]].


^ self
```

displayMachine:aMachine

```
"   Report in the Text pane 2 that the receiver aMachine.
    Answer the receiver."


    inputString := (inputString, 'Machine name => ',aMachine
    name asSymbol printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Operation => ',aMachine
    operation asSymbol printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Cost per machine hour =>
    ',(aMachine costPMH printPaddedTo:1 ) asSymbol
    printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Attachment => ',(aMachine
    attachment) asSymbol printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Horse power class =>
    ',(aMachine hpClass) asSymbol printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Production cost =>
    ',(aMachine prodCost printPaddedTo:1) asSymbol
    printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Max ground firmness =>
    ',(aMachine firmness printPaddedTo:1) asSymbol
    printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Max ground roughness =>
    ',(aMachine roughness printPaddedTo:1) asSymbol
    printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Maintenance cost =>
    ',(aMachine maintCost printPaddedTo:1) asSymbol
    printString, '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Production equation  =>
    ',(aMachine productionEquation) asSymbol printString,
    '\') withCrs.
    self changed: #reply2.
    inputString := (inputString, 'Suspention for one end of
    log => ',(aMachine suspention) asSymbol printString, '\')
    withCrs.
    self changed: #reply2.

 (aMachine dbhRange)  keysDo: [ :akey |
       inputString := (inputString,'dbh range at ', akey
       printString, ' => ', ((aMachine dbhRange) at:akey)
       printString, '\') withCrs.
       self changed: #reply2] .
```

```
  (aMachine slopeRange)  keysDo: [ :akey |
        inputString := (inputString,'slope range at ', akey
        printString, ' => ', ((aMachine slopeRange) at:akey)
        printString, '\') withCrs.
        self changed: #reply2] .

  (aMachine locationState)  keysDo: [ :akey |
        inputString := (inputString,'location ', akey
        printString, ' to final location => '
        ((aMachine locationState) at:akey) printString, '\')
        withCrs.
        self changed: #reply2] .

  (aMachine accumulationState)  keysDo: [ :akey |
        inputString := (inputString,'acc. state ', akey
        printString, ' to => ',
        ((aMachine accumulationState) at:akey) printString,
        '\') withCrs.
        self changed: #reply2] .

  (aMachine productState)  keysDo: [ :akey |
        inputString := (inputString,'Product state ', akey
        printString, ' to => ',
        ((aMachine productState) at:akey) printString, '\')
        withCrs.
        self changed: #reply2] .

  (aMachine succProcess)  do: [ :operation|
        inputString := (inputString,'successor operation =>
        ', operation printString , '\') withCrs.
        self changed: #reply2] .
^self
```

displayOperations

```
" display the operations in Operations DictionaryDictionary
  in text pane 2."

   ((Operations keys) asSortedCollection) do: [ :akey |
        inputString := (inputString, akey printString, '\')
        withCrs.
        self changed: #reply2] .


^ self
```

displayProductionEquations

```
"   display the production equations in text pane 2."

   (PaneSelectors at:#prodEquations) do: [ :anEquation |
        inputString := (inputString, anEquation printString,
        '\') withCrs.
        self changed: #reply2] .


^ self
```

displaySkidderSpeedData

```
" Report in the Text pane 2 the horse power class skidder
  speed data."


 |  tempDictionaryDictionary inputStringTemp|

   inputString := (inputString,'Horse Power Class ',choice4
   asSymbol printString,  '\' ) withCrs.
   inputString := (inputString,'Attachment Class ',choice3
   asSymbol printString,  '\' ) withCrs.
   inputString := (inputString, '\') withCrs.
   inputString := (inputString,'Slope    ', '  Load   ','
   Loaded Speed  ',' Empty Speed ',  '\') withCrs.
   self changed: #reply2.


inputString := (inputString, '-------   ', '--------
','----------  ','--------------   ', '\') withCrs.
self changed: #reply2.

(choice3 == #cable:)
  ifTrue: [
     tempDictionaryDictionary := SkidderSpeedCable
     at:choice4].
(choice3 == #grapple:)
  ifTrue: [
     tempDictionaryDictionary := SkidderSpeedGrapple
     at:choice4].

    ((tempDictionaryDictionary keys) asSortedCollection:
    [:num1 :num2 | (self convertStringToNumber:num1) <=
    (self convertStringToNumber:num2)] )
     do: [ :akey |
     inputStringTemp :=  (akey printString,'  ',
                         (tempDictionaryDictionary
     retrieveValueAt:akey with:#load) printString,'  ',
                         (tempDictionaryDictionary retr
retrieveValueAt:akey with:#loadedSpeed) printString,'  ',
                           (tempDictionaryDictionary
retrieveValueAt:akey with:#emptySpeed) printString ).
  inputString := (inputString,(self
  columnDisplaySkidderData:inputStringTemp),  '\') withCrs.
  inputString := (inputString asSymbol).
  self changed: #reply2].

inputString := (inputString,'-------    ', '--------
','----------  ','--------------    ', '\') withCrs.

self changed: #reply2.


^self


displaySystem:aString

     "   Report in the Text pane 2 the receiver at a string
         in stateCollection .
         Answer the receiver."

 | inputStream outputStream word wordNumber tempNumber
inputStringTemp|

    "  clear reply pane 1."

   inputString := ''.
   self changed: #reply2.


   inputString := (inputString, 'Operation      ', 'Machine
   ','Product     ','Location     ','    Accumulation',
   '\') withCrs.
   self changed: #reply2.


inputString := (inputString, '-----------   ', '----------
','----------  ','--------------    ','--------------    ',
'\') withCrs.

self changed: #reply2.


(stateCollection at:aString) do: [:machine |
   inputStringTemp := ( machine operation asSymbol
   printString,'        '
   ,machine name asSymbol printString,'    '
```

```
    ,machine outputProductState printString,'    '
    ,machine locationOut printString,'    '
    ,machine accumulationStateOut asSymbol printString ,'    '
     ,'\' )  withCrs.
    inputString := (inputString,(self
    columnDisplay:inputStringTemp),  '\')  withCrs.
    inputString := (inputString asSymbol).
    self changed: #reply2].

inputString := (inputString, '-----------   ', '----------
','----------  ','--------------   ','--------------   ',
'\') withCrs.
self changed: #reply2.

  inputString := (inputString, 'Total Environmentally
  adjusted logging cost ',(((stateCollection at:aString)
  cost) printRounded:1),  '\') withCrs.

  self changed: #reply2.

^self
```

displayWoodDensities

```
"   WoodDensity keysDo: [ :akey |
    inputString := (inputString, akey printString, ' set to
    ', ( WoodDensity at:akey) printString, '\') withCrs.
    self changed: #reply2] .

^self
```

editSkidderSpeedData

```
"   Open a prompter window to set the skidder data ."

" Temporary variables "

|tempVal temp|

 " send new prompt to the reply pane 1 ( the help window )."

   replyStream := Reply1Dictionary at:#skidderEditInput6.
   self changed: #reply1.

"  get user input for the value"

tempVal := Prompter prompt: 'parameter value'
defaultExpression: '' point: 200 @ 250.

 " Determine the appropriate operation to locate an object.
```

```
    Once the object is located, update the skidder
    information. "

  (tempVal = nil)
    ifFalse: [
     (choice3 == #cable:)
     ifTrue: [
      (SkidderSpeedCable at:choice4) addValueAt:choice5
      asSymbol with:choice6 with:tempVal.
      inputString := (inputString,'hp ', choice4 asSymbol
      printString, ' for ', choice6 asSymbol printString, '
      at slope ', choice5 asSymbol printString
      ,' set to ', tempVal printString, '\') withCrs.
      self changed: #reply2].

  (choice3 == #grapple:)
      ifTrue: [
      (SkidderSpeedGrapple at:choice4) addValueAt:choice5
      asSymbol with:choice6 with:tempVal.
      inputString := (inputString,'hp ', choice4 asSymbol
      printString, ' for ', choice6 asSymbol printString, '
      at slope ', choice5 asSymbol printString
      ,' set to ', tempVal printString, '\') withCrs.
      self changed: #reply2]].


^self
```

environmentInput2

```
"    evaluate the environment selections for input pane 2."

    ( choice2 == #requirements:)
       ifTrue:  [
         list3Collection:= PaneSelectors
         retrieveCollection:#requirements:.
         replyStream := Reply1Dictionary at: #requirements.
         self changed: #reply1 ].

   ( choice2 == #site:)
       ifTrue:  [
         list3Collection:=  PaneSelectors
         retrieveCollection:#site:.
         replyStream := Reply1Dictionary at: #site.
         self changed: #reply1 ].

    ( choice2 == #stand:)
       ifTrue: [
         list3Collection:= PaneSelectors
         retrieveCollection:#stand:.
         replyStream := Reply1Dictionary at: #stand.
```

```
        self changed: #reply1 ].

^ self
```

equipmentInput2

```
"   evaluate selection for equipment in input pane 2. "

   ( choice2 == #attributes:)
       ifTrue: [
          list3Collection:= Operations.
          replyStream := Reply1Dictionary at: #process.
          self changed: #reply1.].
      (( choice2 == #database:) or: [  choice2 ==
      #available:] )
       ifTrue: [
          list3Collection:= Operations.
          replyStream := Reply1Dictionary at: #process.
          self changed: #reply1].

^ self
```

equipmentInput3:aString

```
"  evaluate input for input pane 3 given equipment was
   selected for input pane 1."


  choice3:=aString.      " store aString to choice3. "

"        Determine if choice2 is add:, and if true open a
         a prompter window to receive input. Store the input
         temp to tempkey which is just a string name. Then
         determine the selected operation to add an object
         to, and create an instance of Machine for the
         required operation. Add the new machine to the
         proper operation collection. Once added answer
         reply pane 2 with an acknowledgement."

          (choice2 == #available:)
            ifTrue: [
             list4Collection :=  PaneSelectors
             retrieveCollection:#operationFunctions.
             replyStream := Reply1Dictionary
             at:#availableOptions.
             self changed:#reply1].
            (choice2 == #database:)
              ifTrue: [
               list4Collection := PaneSelectors
               retrieveCollection:#operationFunctions.
               replyStream := Reply1Dictionary
               at:#databaseOptions.
               self changed:#reply1].



 "  Determine if choice2 was attributes or available."


  (choice2 == #attributes:)
           ifTrue: [


      "  Determine the operation selected and set
         list4Collection. Then set the instance variable
         replyStream and both reply pane 1 and 2."

  listt4Collection:= (Operations at:choice3).
  replyStream := Reply1Dictionary at: #attributeFactors.
                  self changed: #reply1.
                  self changed: #reply2].

^self
```

goalInput2

```
"        Change the state of the harvester panes
         so that specific facts are reviewed or
         selected."



 "   Determine the input2 match and choose the appropriate
     list for the next listPane and clear listPanes beneath
     the next listPane."

 ( choice2 == #location:)
   ifTrue:  [
   list3Collection:= PaneSelectors
   retrieveCollection:#location.
   replyStream := Reply1Dictionary at: #locationGoal].

  ( choice2 == #product:)
   ifTrue:  [
   list3Collection:= PaneSelectors
   retrieveCollection:#inputProduct.
   replyStream := Reply1Dictionary at:#productGoal].

 ^self
```

goalInput3

```
"       send a message to an instance of BestFirstSelector "

 (EquipmentSelector)
       to set goalstates and send response to reply pane 2.

   (choice2 == #location:)
     ifTrue: [
      EquipmentSelector goalLocation:choice3.
      inputString := (inputString, 'The goal location set to
      ', choice3 printString, '\') withCrs .
      self changed: #reply2].

   (choice2 == #product:)
     ifTrue: [
      EquipmentSelector goalProduct:choice3.
      inputString := (inputString, 'The goal product set to
      ', choice3 printString, '\') withCrs .
      self changed: #reply2].


^ self
```

groundSetMaxes:aSymbol with:aString

```
"     set the ground operating maxes. "

       (Operations retrieveValueAt:choice3 with:choice4)
       perform:choice6  with:(self
       convertStringToNumber:aString).
       inputString := (inputString, 'The max ', choice6
       printString, ' for ', choice4
       printString, ' is set to ', choice7 asSymbol
       printString,'\') withCrs.
       self changed: #reply2.
```

```
haulPrompt

"   Open a prompter window to set the haul distance."

" Temporary variables "

|tempVal temp|

tempVal := Prompter prompt: 'haul distance'
defaultExpression: '' point: 200 @ 250.

"  Determine the appropriate operation to locate an object.
   Once the object is located, update the haul distance
   information. "

     EnvironmentDictionary at:choice3 put:tempVal.
     inputString := (inputString, choice3 printString, ' set
     to  ', tempVal printString,   '\') withCrs.
     self changed: #reply2.


^self


hpPrompt

"   Open a prompter window to set the horse power instance
    variable."

" Temporary variables "

|tempVal temp|

"  get user input for the horse power class"

tempVal := Prompter prompt: ' horse power '
defaultExpression: '''' point: 200 @ 250.

" Determine the appropriate operation to locate an object.
  Once the object is located, update the horse power class."


        (Operations retrieveValueAt:choice3 with:choice4)
         hpClass:tempVal.
         inputString := (inputString,'horse power class set
         to ', tempVal asSymbol printString, '\') withCrs.
         self changed: #reply2.


^self
```

```
initialize

" Initialize instance variables."


   stateCollection := Dictionary new.

^ self


input1

"  Initialize inputPane with an OrderedCollection.
   Answer a list1Collection as a SortedCollection."

list1Collection:= OrderedCollection new.
list1Collection:= PaneSelectors
retrieveCollection:#startCollection.

"Return the collection as a SortedCollection."

 ^list1Collection asSortedCollection


input1: aSymbol

 "       Change the state of the harvester panes
         so that specific facts are reviewed or
         selected."

   choice1:= aSymbol.   store input from listPane 1 to
   choice1.

 "  Determine the input1 match and choose the appropriate
    list for the next listPane and clear listPanes beneath
    the next listPane."

   ( choice1 == #environment)
      ifTrue:  [
        list2Collection:= PaneSelectors
        retrieveCollection:#environment:.
        self paneClear.
        replyStream := Reply1Dictionary at: #environment].

  ( choice1 == #equipment)
      ifTrue:  [
        list2Collection:= PaneSelectors
        retrieveCollection:#equipmentSelection.
        self paneClear.
        replyStream := Reply1Dictionary at:#equipment].
  ( choice1 == #operations)
     ifTrue:  [
       list2Collection:= PaneSelectors
       retrieveCollection:#operationFunctions.
       self paneClear.
       replyStream := Reply1Dictionary at:#operations].

   ( choice1 == #goal)
     ifTrue:  [
       list2Collection:= PaneSelectors at:#goalStates:.
       self paneClear.
       replyStream := Reply1Dictionary at:#goal].

  ( choice1 == #reports)
     ifTrue:  [
       list2Collection:= PaneSelectors
       retrieveCollection:#reports.
       self paneClear.
       replyStream := Reply1Dictionary at:#reports].

    ( choice1 == #woodDensity)
     ifTrue:  [
       list2Collection:= PaneSelectors
       retrieveCollection:#dataFunctions.
       self paneClear.
       replyStream := Reply1Dictionary at:#woodDensity].

     ( choice1 == #skidderSpeed)
      ifTrue:  [
       list2Collection:= PaneSelectors
       retrieveCollection:#dataFunctions.
       self paneClear.
       replyStream := Reply1Dictionary at:#woodDensity].

     (choice1 == #prodEquations)
    ifTrue:  [
       list2Collection:= PaneSelectors
       retrieveCollection:#operationFunctions.
       self paneClear.
       replyStream := Reply1Dictionary at:#prodEquation].

   (choice1 == #printScreen)
    ifTrue:  [ Display outputToPrinterUpright].

" send a cascading message to self to update pertinate
  application panes."

  self

       changed: #input2;
       changed: #reply1;
       changed: #reply2;
```

```
          changed: #input3;
          changed: #input4;
          changed: #input5;
          changed: #input6;
          changed: #input7;
          changed: #input8;
          changed: #reply3;
          changed: #reply4;
          changed: #reply5;
          changed: #reply6.


input2

"        contents of inputPane ."

(list2Collection isNil )
   ifTrue: [^ list2Collection:= OrderedCollection new ].

(list2Collection isKindOf: OrderedCollection )
 ifTrue:[
 (list2Collection notEmpty)
 ifTrue: [
 (self isFirstElementAlpha:list2Collection)
 ifFalse: [^list2Collection asSortedCollection: [:num1 :num2
 | (self convertStringToNumber:num1) <= (self
  convertStringToNumber:num2)]]
  ifTrue: [^list2Collection asSortedCollection]]].

 (list2Collection isKindOf: Dictionary)
     ifTrue:[ ^ list2Collection keys asSortedCollection].


input2: aSymbol

 "   Determine the input2 match and choose the appropriate
     list for the next listPane and clear listPanes beneath
     the next listPane."

   choice2:= aSymbol.

   ( choice1 == #goal)
    ifTrue: [ self goalInput2].

   ( choice1 == #environment)
    ifTrue: [ self environmentInput2].

   ( choice1 == #operations)
    ifTrue: [ self operationsInput2].

   ( choice1 == #prodEquations)
    ifTrue: [self productionEquationsInput2].
    ( choice1 == #equipment)
     ifTrue: [ self equipmentInput2].

    ( choice1 == #reports)
     ifTrue: [self reportsInput2].

    ( choice1 == #woodDensity)
     ifTrue: [ self woodDensityInput2].

    ( choice1 == #skidderSpeed)
     ifTrue: [ self skidderSpeedInput2].

"   set instance variables for trailing list panes to nil."

     list4Collection := nil.
     list5Collection := nil.
     list6Collection := nil.
     list7Collection := nil.
     list8Collection := nil.

 "  send a cascading message to self to update pertinate
    application panes."



 self
        changed: #input3;
        changed: #input4;
        changed: #input5;
        changed: #input6;
        changed: #input7;
        changed: #input8;
        changed: #reply1.
```

input3

```
"       contents of inputPane ."

(list3Collection isNil )
   ifTrue: [^ list3Collection:= OrderedCollection new ].
(list3Collection isKindOf: OrderedCollection )
     ifTrue:[
    (list3Collection notEmpty)
       ifTrue: [
        (self isFirstElementAlpha:list3Collection)
         ifFalse: [^list3Collection asSortedCollection:
         [:num1 :num2 | (self convertStringToNumber:num1) <=
         (self convertStringToNumber:num2)]]
         ifTrue: [^list3Collection asSortedCollection]]].

(list3Collection isKindOf: Dictionary)
     ifTrue:[ ^ list3Collection keys asSortedCollection].
```

```
input3:aString

"    Determine the input3 match and choose the appropriate
     list for the next listPane and clear listPanes beneath
     the next listPane."


  choice3 := aString.

  (choice2 == #stand:)
    ifTrue: [ self standInput3].

  (choice2 == #site:)
    ifTrue: [self siteInput3].

  (choice1 == #goal)
    ifTrue: [self goalInput3].

  (choice2 == #requirements:)
    ifTrue: [ self requirementsInput3].

  (choice1 == #operations)
     ifTrue:  [self operationsInput3:aString].

   (choice1== #equipment)
      ifTrue:  [ self equipmentInput3:aString].

   (choice1 == #reports)
      ifTrue:  [ self reportsInput3:aString].

   (choice1 == #woodDensity)
      ifTrue:  [ self woodDensityInput3:aString].

   (choice1 == #prodEquations)
      ifTrue: [ self productionEquationsInput3].

   ( choice1 == #skidderSpeed)
    ifTrue: [ self skidderSpeedInput].

"    set instance variables for trailing list panes to nil."

                list5Collection := nil.
                list6Collection := nil.
                list7Collection := nil.
                list8Collection := nil.

 " send a cascading message to self to update pertinate
   application panes."


^ self
```

```
            changed: #input4;
            changed: #input5;
            changed: #input6;
            changed: #input7;
            changed: #input8.


input4
"        contents of inputPane ."
(list4Collection isNil )
   ifTrue: [^ list4Collection:= OrderedCollection new ].
(list4Collection isKindOf: OrderedCollection )
     ifTrue:[
    (list4Collection notEmpty)
       ifTrue: [
        (self isFirstElementAlpha:list4Collection)
         ifFalse: [^list4Collection asSortedCollection:
         [:num1 :num2 | (self convertStringToNumber:num1) <=
         (self convertStringToNumber:num2)]]
         ifTrue: [^list4Collection asSortedCollection]]].

(list4Collection isKindOf: Dictionary)
     ifTrue:[ ^ list4Collection keys asSortedCollection].


input4: aSymbol

"    Determine the input4 match and choose the appropriate
     list for the next listPane and clear listPanes beneath
     the next listPane."

" Temporary variable "

 | temp tempkey|


   choice4:= aSymbol.

  (choice2 == #site:)
    ifTrue: [self siteInput4].

  (choice2 == #requirements:)
    ifTrue: [ self requirementsInput4].


  (choice2 == #stand:)
      ifTrue: [ self standInput4].


 If choice2 = attributes retreive the list of attributes
  for list5Collection.
```

```
      ( choice2 == #attributes:)
         ifTrue:  [self attributesInput4].


   "    If choice2 = available then determine what operation
        is currently selected and place the selected machine
        in the available equipment list."

   (choice2 == #available:)
     ifTrue: [ self availableInput4].


   (choice2 == #database:)
     ifTrue: [self databaseInput4].

          (choice1 == #skidderSpeed )
             ifTrue: [self skidderSpeedInput4].


  "    set instance variables for trailing list panes to nil."


           list6Collection := nil.
           list7Collection := nil.
           list8Collection := nil.
   self

    " send a cascading message to self to update pertinate
      application panes."


          changed: #input5;
          changed: #input6;
          changed: #input7;
          changed: #input8.


input5
"        contents of inputPane ."

(list5Collection isNil )
  ifTrue: [^ list5Collection:= OrderedCollection new ].
(list5Collection isKindOf: OrderedCollection )
     ifTrue:[
    (list5Collection notEmpty)
       ifTrue: [
        (self isFirstElementAlpha:list5Collection)
         ifFalse: [^list5Collection asSortedCollection:
         [:num1 :num2 | (self convertStringToNumber:num1) <=
         (self convertStringToNumber:num2)]]
         ifTrue: [^list5Collection asSortedCollection]]].
```

```
(list5Collection isKindOf: Dictionary)
     ifTrue:[ ^ list5Collection keys asSortedCollection].


input5: aSymbol

"    Determine the input5 match and choose the appropriate
     list for the next listPane and clear listPanes beneath
     the next listPane."

   | temp tempkey |

choice5 := aSymbol.

(choice2 == #site:)
  ifTrue: [  self siteInput5:aSymbol ].

(choice2 == #attributes:)
  ifTrue: [ self attributesInput5].

 (choice2 == #available:)
   ifTrue: [
     choice5:= aSymbol.
     tempkey:= (choice3 asSymbol).
     self availableInput5:tempkey].


  (choice2 == #database:)
   ifTrue: [
     choice5:= aSymbol.
     tempkey:= (choice3 asSymbol).
     self databaseInput5:tempkey].

   (choice1 == #skidderSpeed )
           ifTrue: [ self skidderSpeedInput5].



           list7Collection := nil.
           list8Collection := nil.
 self

"     Send self a message to update listPane6."

          changed: #input6;
          changed: #input7;
          changed: #input8.


input6
"  contents of inputPane ."
```

```
(list6Collection isNil )
  ifTrue: [^ list6Collection:= OrderedCollection new ].
(list6Collection isKindOf: OrderedCollection )
     ifTrue:[
    (list6Collection notEmpty)
       ifTrue: [
        (self isFirstElementAlpha:list6Collection)
         ifFalse: [^list6Collection asSortedCollection:
         [:num1 :num2 | (self convertStringToNumber:num1) <=
         (self convertStringToNumber:num2)]]
         ifTrue: [^list6Collection asSortedCollection]]].

(list6Collection isKindOf: Dictionary)
     ifTrue:[ ^ list6Collection keys asSortedCollection].
```

input6:aString

```
"  Determine the input6 match and choose the appropriate
   action to be taken next."

"   Temparary variables"


    | tempkey temp |

choice6 := aString.

 (choice2 == #site:)
     ifTrue: [ self siteInput6:aString].

( choice2 == #attributes:)
     ifTrue: [ self attributesInput6:aString].

((choice1 == #skidderSpeed) and: [choice2 == #edit:])
     ifTrue: [ self editSkidderSpeedData].


       list8Collection := nil.

 self
"        Send self a message to update listPane7."

       changed: #input7;
       changed: #input8.
```

```
input7
"       contents of inputPane ."

(list7Collection isNil )
 ifTrue: [^ list7Collection:= OrderedCollection new ].
(list7Collection isKindOf: OrderedCollection )
     ifTrue:[
    (list7Collection notEmpty)
       ifTrue: [
        (self isFirstElementAlpha:list7Collection)
         ifFalse: [^list7Collection asSortedCollection:
         [:num1 :num2 | (self convertStringToNumber:num1) <=
         (self convertStringToNumber:num2)]]
         ifTrue: [^list7Collection asSortedCollection]]].

(list7Collection isKindOf: Dictionary)
     ifTrue:[ ^ list7Collection keys asSortedCollection].


input7:aSymbol

"  Determine the input7 match and choose the appropriate
   action to be taken next."


choice7 := aSymbol.

 (choice2 == #attributes:)
   ifTrue: [self attributesInput7].


 self

          changed: #input8


input8
"       contents of inputPane ."

(list8Collection isNil )
  ifTrue: [^ list8Collection:= OrderedCollection new ].
(list8Collection isKindOf: OrderedCollection )
     ifTrue:[
    (list8Collection notEmpty)
       ifTrue: [
        (self isFirstElementAlpha:list8Collection)
         ifFalse: [^list8Collection asSortedCollection:
         [:num1 :num2 | (self convertStringToNumber:num1) <=
         (self convertStringToNumber:num2)]]
         ifTrue: [^list8Collection asSortedCollection]]].

(list8Collection isKindOf: Dictionary)
```

```
    ifTrue:[ ^ list8Collection keys asSortedCollection].
```

input8:aSymbol

```
"  Determine the input8 match and choose the appropriate
   action to be taken next."

"   Temparary variables"

    | tempkey temp |


tempkey := choice3 asSymbol.


((choice7 == #output:) and: [choice5 == #input:])
   ifTrue: [
    choice8 := aSymbol.
     (Operations retrieveValueAt:tempkey
      with:choice4)
      productState:choice6 with:choice8.
      inputString := (inputString,choice6 printString,
      ' as input will yield ', choice8 printString,' as
      output', '\') withCrs .
      self changed: #reply2].

((choice7 == #output:) and: [ choice5 == #location:])
  ifTrue: [
   choice8 := aSymbol.
    (Operations retrieveValueAt:tempkey with:choice4)
             locationState:choice6 with:choice8.
             inputString := (inputString,choice6
             printString,
             ' at input location => ', choice8 printString,
             '\') withCrs .
              self changed: #reply2].

((choice7 == #output:) and: [ choice5 == #accumulation:])
  ifTrue: [
   choice8 := aSymbol.
    (Operations retrieveValueAt:tempkey with:choice4)
             accumulationState:choice6 with:choice8.
             inputString := (inputString,'starting with ',
             choice6 printString,
             ' to => ', choice8 printString, '\') withCrs .
             self changed: #reply2].



  ^ self
```

```
inputMenu1

"    Answer a Menu for the input Pane."

    ^Menu
        labels: 'display environment\match equipment'
        withCrs
        lines: #(1 2 )
        selectors: #( displayEnvironment runMatchEquip )


inputMenu2

"    Answer a Menu for the input Pane."

    ^Menu
        labels: 'create an operation\remove an operation'
        withCrs
        lines: #(1)
        selectors: #( createOperation removeOperation)


inputMenu3

"  Answer a Menu for the input Pane."
    ^Menu
        labels: 'display performance factors' withCrs
        lines: #()
        selectors: #( displayFactors)


inputMenu5
        Answer a Menu for the input Pane.
    ^Menu
        labels: 'display performance factors' withCrs
        lines: #()
        selectors: #( displayFactors)


isFirstElementAlpha:aCollection

"   check to see if the first element of a collection is a
    Alpha."


    | firstCharacter firstElement|


 firstElement := (aCollection at:1).
 firstCharacter := (firstElement at:1).
 ((firstCharacter isDigit) or: [ (firstCharacter asciiValue
  = 45) ])
            ifFalse: [ ^true]
            ifTrue: [ ^false].
```

maintPrompt

```
"   a prompter to set the maintenance cost."

|tempVal choiceSelector tempkey temp|
tempkey := choice4.
choiceSelector:= (choice5)asSymbol.
tempVal := Prompter prompt: 'Maintenance Cost.'
defaultExpression: '75' point: 200 @ 250.
    (Operations retrieveValueAt:choice3 with:tempkey)
    perform: choiceSelector with:tempVal.
    inputString := (inputString, choice4 printString,
    ' for ', choice3
    printString, ' at ', choice5 printString,' set to ',
    tempVal printString, '\') withCrs.
    self changed: #reply2.


^self
```

merchPrompt

```
"Open a prompter window to set the density of merch trees ."

" Temporary variables "

|tempVal temp|


 " get user input for the merch trees"

tempVal := Prompter prompt: 'density merch Trees/acre'
defaultExpression: '' point: 200 @ 250.

" Determine the appropriate operation to locate an object.
  Once the object is located, update the slope information."

  EnvironmentDictionary at:choice3 put:tempVal.
  inputString := (inputString, choice3 printString, ' set to
  ', tempVal printString,   '\') withCrs.
  self changed: #reply2.


^self
```

```
openOn: aString

"        Create a Timber harvester window with aString."

    | topPane replyPane1 replyPane2 replyPane3 replyPane4
    replyPane5 replyPane6|

    inputString := String new.
    stateCount := String new.
    stateGenWindow := String new.
    replyStream:= Reply1Dictionary at: #begin.
    topPane := TopPane new label: 'T I M B E R   H A R V E S
    T E R  ver 1.0 '.
    topPane addSubpane:
        (replyPane1 := TextPane new
            model: self;
            name: #reply1;
            change:#status:;
            framingRatio: (0 @ 0 extent: 2/3 @ (3/16))).
    topPane addSubpane:
        (replyPane2 := TextPane new
            model: self;
            name: #reply2;
            change:#status:;
            framingRatio: (0 @ (1/4) extent: 2/3 @ (3/4))).
    topPane addSubpane:
         (inputPane1 := ListPane new
            menu: #inputMenu1;
            model: self;
            name: #input1;
            change:#input1:;
            framingRatio: (2/3 @ 0 extent: 1/6 @ (1/4))).
    topPane addSubpane:
        (inputPane2 := ListPane new
            menu: #inputMenu1;
            model: self;
            name: #input2;
            change:#input2:;
            framingRatio: (5/6 @ 0 extent: 1/6 @ (1/4))).
      topPane addSubpane:
        (inputPane3 := ListPane new
            menu: #inputMenu1;
            model: self;
            name: #input3;
            change:#input3:;
            framingRatio: (2/3 @ (1/4) extent: 1/6 @
            (1/4))).
       topPane addSubpane:
        (inputPane4 := ListPane new
            menu: #inputMenu1;
            model: self;
            name: #input4;
        change:#input4:;
        framingRatio: (5/6 @ (1/4) extent: 1/6 @
        (1/4))).
   topPane addSubpane:
    (inputPane5 := ListPane new
        menu: #inputMenu1;
        model: self;
        name: #input5;
        change:#input5:;
        framingRatio: (2/3 @ (2/4) extent: 1/6 @
        (1/4))).
   topPane addSubpane:
    (inputPane6 := ListPane new
        menu: #inputMenu1;
        model: self;
        name: #input6;
        change:#input6:;
        framingRatio: (5/6 @ (2/4) extent: 1/6 @
       (1/4))).
topPane addSubpane:
    (inputPane7 := ListPane new
        menu: #inputMenu1;
        model: self;
        name: #input7;
        change:#input7:;
        framingRatio: (2/3 @ (3/4) extent: 1/6 @
        (1/4))).
    topPane addSubpane:
    (inputPane8 := ListPane new
        menu: #inputMenu1;
        model: self;
        name: #input8;
        change:#input8:;
        framingRatio: (5/6 @ (3/4) extent: 1/6 @
        (1/4))).
    topPane addSubpane:
    (replyPane3 := TextPane new
        model: self;
        name: #reply3;
        change:#status:;
        framingRatio: (0 @ (3/16) extent: 1/6 @
        (1/16))).
    topPane addSubpane:
    (replyPane4 := TextPane new
        model: self;
        name: #reply4;
        change:#status:;
        framingRatio: (1/6 @ (3/16) extent: 1/6 @
        (1/16))).
    topPane addSubpane:
    (replyPane5 := TextPane new
        model: self;
```

```
                name: #reply5;
                change:#status:;
                framingRatio: (2/6 @ (3/16) extent: 1/6 @
                (1/16))).
            topPane addSubpane:
            (replyPane6 := TextPane new
                model: self;
                name: #reply6;
                change:#status:;
                framingRatio: (1/2 @ (3/16) extent: 1/6 @
               (1/16))).

            topPane reframe: (Display boundingBox insetBy: 0@0).
            topPane dispatcher openWindow scheduleWindow
```

operationsInput2

```
"  evaluate the selections for operations for input pane 2."

   ( choice2 == #remove:)
        ifTrue: [
        list3Collection:= Operations.
         replyStream := Reply1Dictionary at: #process.
         self changed: #reply1 ].
       ( choice2 == #add:)
          ifTrue: [
           list3Collection:= Operations.
           replyStream := Reply1Dictionary at: #operationAdd.
           self changed: #reply1.
           self createOperation].
      ( choice2 == #display:)
          ifTrue: [
           replyStream := Reply1Dictionary at:
           #operationsDisplay.
           self changed: #reply1.
           self displayOperations].

^ self
```

operationsInput3:aString

```
" respond to the selection of operations for input 1  and
  the selection of a data function for input pane 2."


            choice3:=aString.      store aString to choice3.
            (choice2 == #remove:)
            ifTrue: [ self removeOperation:choice3].

^self
```

paneClear

```
   "  clears the reply panes "

         list3Collection := nil.
         list4Collection := nil.
         list5Collection := nil.
         list6Collection := nil.
         list7Collection := nil.
         list8Collection := nil.
         inputString:= ''.
         stateGenWindow := ''.
         stateCount := ''.
         queueCount := ''.
         queueCntWindow := ''.

^self
```

prodPrompt

```
"   a prompter to set the production cost."

|tempVal choiceSelector tempkey temp|
tempkey := choice4.
choiceSelector:= (choice5)asSymbol.
tempVal := Prompter prompt: 'Production Cost.'
defaultExpression: '' point: 200 @ 250.
        (Operations retrieveValueAt:choice3 with:tempkey)
        perform: choiceSelector with:tempVal.
        inputString := (inputString, choice4 printString, '
        for ', choice3
        printString, ' at ', choice5 printString,' set to ',
        tempVal printString, '\') withCrs.
        self changed: #reply2.
^self
```

productionEquationsInput2

```
"     evaluate selections for production equations for input
      pane 2."

      ( choice2 == #remove:)
         ifTrue: [
          list3Collection:= PaneSelectors
          retrieveCollection:#prodEquations.
          replyStream := Reply1Dictionary at:
          #prodEquationsRemove.
          self changed: #reply1 ].
```

```
        ( choice2 == #add:)
          ifTrue: [
           replyStream := Reply1Dictionary at:
           #prodEquationsAdd.
           self changed: #reply1.
           self createEquationSymbol].

        ( choice2 == #display:)
          ifTrue: [
           replyStream := Reply1Dictionary at:
           #prodEquationsDisplay.
           self changed: #reply1.
           self displayProductionEquations].

  ^ self
```

productionEquationsInput3

```
"  evalutate production equations for input 3."

      ( choice2 == #remove:)
       ifTrue: [self removeProductionEquation].
           ( choice2 == #display:)
       ifTrue: [self displayProductionEquations].

^ self
```

queueSize

```
" return size of queue"

^queueCount
```

removeFromAvailable:tempkey

```
"   remove from to available equipmenmt "

        | temp |

 ((OperationsAvailable at:tempkey) includesKey:choice5)
  ifTrue: [OperationsAvailable removeValueAt:tempkey
  with:choice5.
  inputString := (inputString, choice5 printString,
  ' removed from ', tempkey  printString, ' operation ',
  '\') withCrs .
  self changed: #reply2]
  ifFalse: [
  inputString := (inputString, choice5 printString,
  ' is already removed from ', tempkey  printString, '
  operation ', '\') withCrs .
  self changed: #reply2].

^ self
```

removeFromDatabase:tempkey

```
"          remove aMachine from both database machines and
           available machines."

| temp |

 Operations removeValueAt:tempkey with:choice5.
 inputString := (inputString, choice5 printString,
 ' removed from database for',  '\') withCrs .
 self changed: #reply2.
 inputString := (inputString, tempkey  printString, '
 operation. ', '\') withCrs .
 self changed: #reply2.

 ((OperationsAvailable at:tempkey) includesKey:choice5)
 ifTrue: [OperationsAvailable removeValueAt:tempkey
 with:choice5.
 inputString := (inputString, choice5 printString,
 'removed from available equipment for the ',  '\') withCrs
.
 self changed: #reply2.
 inputString := (inputString, tempkey  printString, '
 operation.', '\') withCrs .
 self changed: #reply2]
 ifFalse: [
 inputString := (inputString, choice5 printString,
 ' is already removed from the available ',  '\') withCrs .
 self changed: #reply2.
 inputString := (inputString, 'equipment for the ', tempkey
 printString, ' operation.', '\') withCrs .
 self changed: #reply2].

    ^self
```

removeOperation:aString

```
"  remove an  operation. "

 | temp tempkey   |

  Operations removeKey:(aString asSymbol) ifAbsent: [^self].
  Operations do: [:operation|
  operation do: [:equipment|
  ((equipment succProcess) includes:(aString asSymbol))
  ifTrue: [(equipment succProcess) remove:(aString
  asSymbol)]]].


  OperationsAvailable removeKey:(aString asSymbol) ifAbsent:
  [^self].
  inputString := (inputString, aString printString,
  ' removed from Operations' printString, '\') withCrs .
  self changed: #reply2.

                list4Collection := nil.
                list5Collection := nil.
                list6Collection := nil.

      ^self


              changed: #input3;
              changed: #input4;
              changed: #input5;
              changed: #input6.
```

removeProductionEquation

```
"  remove a production equation. "

(choice3 = #notDefined)
 ifFalse: [
   PaneSelectors removeValueAt:#prodEquations with:choice3.
   inputString := (inputString, choice3 printString,
   ' removed from production equations' printString, '\')
   withCrs .
   self changed: #reply2]
   ifTrue: [
   inputString := (inputString, choice3 printString,
   ' cannot be removed from equations'asSymbol printString,
   '\') withCrs .

   self changed: #reply2].


                list4Collection := nil.
                list5Collection := nil.
                list6Collection := nil.

      ^self


              changed: #input3;
              changed: #input4;
              changed: #input5;
```

```
                changed: #input6.

reply1

"        Initialize reply pane with an
         empty String."

    ^replyStream


reply1:aSymbol

"       Initialize replyPanel "

   ^Reply1Dictionary at:aSymbol


reply2

  "      Initialize reply pane with an
         empty String."

   ^inputString


reply3

"        Initialize reply pane with an
         empty String."

   ^ stateGenWindow


reply4

"  Initialize reply pane with an
         empty String."

   ^stateCount


reply5

"        Initialize reply pane with an
         empty String."

   ^queueCntWindow
```

```
reply6

"        Initialize reply pane with an
         empty String."

   ^queueCount


reportFailure

    "   Report in the TimberSimPane text pane 2 that the
        receiver failed to find the goal.
        Answer the receiver."

         inputString := (inputString, 'a suitable system is
         not possible' asSymbol printString, '\') withCrs.
         self changed: #reply2.

    ^self


reportsInput2

"   respond to a selection of reports for input pane 2."

   list3Collection:= (((self stateCollection) keys)
   asSortedCollection: [:num1 :num2 |
   (self convertStringToNumber:num1) <= (self
   convertStringToNumber:num2)] ).
   (choice2 == #display:)
    ifTrue: [
    replyStream := Reply1Dictionary at: #reportsDisplay.
    self changed: #reply1 ].
   (choice2 == #print:)
    ifTrue: [
    replyStream := Reply1Dictionary at: #reportsPrint.
    self changed: #reply1 ].

^ self
```

reportsInput3:aString

```
" respond to the input pane 1 selector being reports.  Using
  both choice2 and choice3 from their respect list panes."


        choice3:= aString.
        (choice2== #display:)
          ifTrue:  [ self displaySystem:choice3].
        (choice2 == #print:)
            ifTrue: [ (inputString, '\') withCrs
            outputToPrinter ].

    ^self
```

reportSuccess:aSystem

```
 "    Report in the Text pane 2 that the receiver found the
       goal. Answer the receiver."



    inputString := (inputString, 'a suitable system is '
    asSymbol printString,  '\') withCrs.
    self changed: #reply2.
    inputString := (inputString,'$-[', (aSystem cost
    printRounded:1),']').
    self changed:#reply2.
    aSystem do: [ :machine |
    inputString := (inputString,'->', (machine name asSymbol
    printString)).
    self changed: #reply2].

    ^self
```

requirementsInput3

```
 "  evaluate requirment selection for input pane 3."

  (choice3 == #suspention:)
    ifTrue: [
     list4Collection :=  PaneSelectors
     retrieveCollection:#booleanCollection.
     replyStream := Reply1Dictionary at:#suspention.
     self changed:#reply1].

  (choice3 == #shearFelling:)
    ifTrue: [
     list4Collection :=  PaneSelectors
     retrieveCollection:#booleanCollection.
```

```
          replyStream := Reply1Dictionary at:#shearFelling.
          self changed:#reply1].

      (choice3 == #product:)
        ifTrue: [
          list4Collection :=  PaneSelectors
          retrieveCollection:#outputProduct.
          replyStream := Reply1Dictionary at:#finishedProduct.
          self changed:#reply1].

   ^ self
```

requirementsInput4

```
"  evaluate requirements for input pane 4."

   (choice3 == #suspention:)
         ifTrue: [
            EnvironmentDictionary at:choice3 put:choice4.
            inputString := (inputString, choice3 printString,
            ' set to ', choice4 printString, '\') withCrs.
            self changed: #reply2 ].

   (choice3 == #product:)
         ifTrue: [
            EnvironmentDictionary at:choice3 put:choice4.
            inputString := (inputString, choice3 printString,
            ' set to ', choice4 printString, '\') withCrs.
            self changed: #reply2 ].

   (choice3 == #shearFelling:)
         ifTrue: [
            EnvironmentDictionary at:choice3 put:choice4.
            inputString := (inputString, choice3 printString,
            ' set to ', choice4 printString, '\') withCrs.
            self changed: #reply2 ].

^self
```

runMatchEquip

```
"    run the search procedure "
 stateGenWindow := 'States Gen.' asSymbol.
 queueCntWindow := 'Queue Size ' asSymbol.
 stateCollection := Dictionary new.
 self
      changed: #reply3;
      changed: #reply5.
 ((EquipmentSelector goalLocation) isNil and:
[(EquipmentSelector goalProduct) isNil])
 ifFalse: [
 EquipmentSelector startSystem.
 EquipmentSelector search].

^ self
```

showQueueCount:aQueue

```
"    show the states generated so far then show it in
     in the text pane 4."

     queueCount := aQueue size printString.
     self changed: #reply6.


^self
```

showStartSystem:aSystemFinder

```
"    show the systemFinder type and the initial state
     in the text pane 2."

     inputString := ''.
     self changed: #reply2.
     inputString := (inputString,'The required harvesting
     product is ' asSymbol printString,
     (EquipmentSelector goalProduct) asSymbol printString,
      '\') withCrs.
      self changed: #reply2.
     inputString := (inputString,'The required final
     location is ' asSymbol printString,
     (EquipmentSelector goalLocation) asSymbol printString,
      '\') withCrs.
      self changed: #reply2.
     inputString := (inputString,'1:',(aSystemFinder
     printString), ' starting with standing timber 'asSymbol
     printString,  '\') withCrs.
     self changed: #reply2.
^self
```

showStatesCount:aNumber

```
"    show the states generated so far then show it in
     in the text pane 4."

     stateCount := aNumber printString.
     self changed: #reply4.


^self
```

showStatesGenerated:aNumber

```
" show the states generated so far then show it in
  in the text pane 2."

  inputString := (inputString, (aNumber printString),': ').
  self changed: #reply2.

^self
```

showSuccessorState:aSystem

```
"    show the new successor state generated in
     in the text pane 2."

    inputString := (inputString,'$-[', (aSystem cost
    printRounded:1),']').
    self changed:#reply2.
    aSystem do: [ :aMachine |
    inputString := (inputString,'->', (aMachine name
    asSymbol printString)).
    self changed:#reply2].
    inputString := (inputString, '\' withCrs).
    self changed:#reply2.

    ^self
```

siteInput3

```
"    evaluate site selectors for input pane 3."

       (choice3 == #haulDistance:)
        ifTrue: [
         replyStream := Reply1Dictionary at:#promptMessage.
            self changed: #reply1.
            self haulPrompt].

       (choice3 == #terrain:)
```

```
        ifTrue: [
         list4Collection :=  PaneSelectors
         retrieveCollection:#terrain:.
         replyStream := Reply1Dictionary at:#terrain.
         self changed:#reply1].

^ self
```

siteInput4

```
"     evaluate input for site in input pane 4. "

  (choice4 == #ground:)
     ifTrue: [
         list5Collection:=  PaneSelectors
         retrieveCollection:#groundSpecifics:.
         replyStream := Reply1Dictionary at: #ground.
         self changed: #reply1 ].

   (choice4 == #slope:)
     ifTrue: [
         list5Collection:=  PaneSelectors
         retrieveCollection:#slopeCollection.
         replyStream := Reply1Dictionary at: #slope.
         self changed: #reply1 ].

^ self
```

siteInput5:aSymbol

```
| temp tempkey |

 choice5 := aSymbol.
     (choice4 == #slope:)
          ifTrue: [
                 temp:= self convertStringToNumber:aSymbol.
                  tempkey:= choice4.
                  EnvironmentDictionary at:tempkey put:temp.
                  inputString := (inputString, tempkey
                  printString, ' set to ', temp
                  printString, '\') withCrs .
                  self changed: #reply2].
       (choice4 == #ground:)
        ifTrue: [
          (choice5 == #firmness:)
            ifTrue: [
                  list6Collection := PaneSelectors
                  retrieveCollection:#oneToFive:.
                  replyStream := Reply1Dictionary at:
                  #firmness.
```

```
                    self changed: #reply1].
                (choice5 == #roughness:)
                  ifTrue: [
                        list6Collection := PaneSelectors
                        retrieveCollection:#oneToFive:.
                        replyStream := Reply1Dictionary at:
                        #roughness.
                        self changed: #reply1]].

^ self
```

siteInput6:aString

```
  "    site selectors for input pane 6 "

    choice6 := aString asSymbol.
    (choice5 == #firmness:)
        ifTrue: [
            EnvironmentDictionary at:#groundFirmness
            put:(aString asInteger).
            inputString := (inputString, 'ground firmness
            set to ',choice6 printString,  '\') withCrs.
            self changed: #reply2].
  (choice5 == #roughness:)
        ifTrue: [
            EnvironmentDictionary at:#groundRoughness
            put:(aString asInteger).
            inputString := (inputString, 'ground roughness
            set to ',choice6 printString,  '\') withCrs.
            self changed: #reply2].

^ self
```

skidderSpeedInput2

```
 "    respond to skidder speed being selected for input1. "

      (choice2 == #add:)
         ifTrue: [
          list3Collection:= PaneSelectors
          retrieveCollection:#skidderSpeed.
          replyStream := Reply1Dictionary at:
          #skidderSpeedAdd.
          self changed: #reply1 ].
      (choice2 == #remove:)
         ifTrue: [
          list3Collection:= PaneSelectors
          retrieveCollection:#skidderSpeed.
          replyStream := Reply1Dictionary at:
            #skidderSpeedRemove.
            self changed: #reply1 ].
        (choice2 == #edit:)
           ifTrue: [
            list3Collection:= PaneSelectors
            retrieveCollection:#skidderSpeed.
            replyStream := Reply1Dictionary at:
            #skidderSpeedEdit.
            self changed: #reply1 ].
        (choice2 == #display:)
           ifTrue: [
            list3Collection:= PaneSelectors
            retrieveCollection:#skidderSpeed.
            replyStream := Reply1Dictionary at:
            #skidderSpeedDisplay.
            self changed: #reply1 ].
 ^self
```

skidderSpeedInput3

```
" evalutate skidder speed for input 3."

   ( choice2 == #add:)
      ifTrue: [
              replyStream := Reply1Dictionary at:#hpClass.
              self changed:#reply1.
              self addToSkidderSpeed].
    ( choice2 == #remove:)
      ifTrue: [
              list4Collection := SkidderSpeedCable.
              replyStream := Reply1Dictionary
              at:#skidderRemove.
              self changed:#reply1].
     ( choice2 == #display:)
      ifTrue: [
              list4Collection := SkidderSpeedCable.
              replyStream := Reply1Dictionary
              at:#skidderDisplay.
              self changed:#reply1].

   ( choice2 == #edit:)
      ifTrue: [
              list4Collection := SkidderSpeedCable.
              replyStream := Reply1Dictionary
              at:#skidderEdit.
              self changed:#reply1].

^self
```

skidderSpeedInput4

```
"  evaluate skidderSpeed input for input pane 4."

      (choice2 == #edit:)
               ifTrue: [
                      list5Collection:=  PaneSelectors
                      retrieveCollection:#slopeCollection.
                      replyStream := Reply1Dictionary at:
                      #skidderEditInput4.
                      self changed: #reply1].
       ( choice2 == #display:)
               ifTrue: [ self displaySkidderSpeedData].


^self
```

skidderSpeedInput5

```
" evaluate skidder speed input for input pane 5."

   (choice2 == #edit:)
               ifTrue: [
               list6Collection:=  PaneSelectors
               retrieveCollection:#skidderSpeedVariables.
               replyStream := Reply1Dictionary at:
               #skidderParameter.
               self changed: #reply1 ].

^ self
```

slopePrompt

```
"   Open a prompter window to set the slope weight factor."

" Temporary variables "

|tempVal choiceSelector tempkey temp|

tempkey := choice4.        "  store choice4 to tempkey"

" concatenate choice5 with #with: as a sysmbol"

choiceSelector:= (choice5,#with:)asSymbol.

"  get user input for the performance factor "

tempVal := Prompter prompt: 'Slope Performance Factor.'
defaultExpression: '' point: 200 @ 250.

" Determine the appropriate operation to locate an object.
  Once the object is located, update the slope information."
```

```
     (Operations retrieveValueAt:choice3 with:tempkey)
     perform: choiceSelector with:choice6 with:tempVal.
     inputString := (inputString, choice4 printString, ' for
     ', choice5 printString,
     ' at ', choice6 printString, ' set to ', tempVal
     printString, '\') withCrs.
     self changed: #reply2.


^self


slopeSetBounds:aString with:aKey

"   set the upper or lower bounds on slope. "

       (Operations retrieveValueAt:choice3 with:choice4)
       slopeRange:aKey  with:(self
       convertStringToNumber:aString).
       inputString := (inputString, 'The slope  ', choice6
       printString, ' for ', choice4
       printString, ' is set to ', choice7 asSymbol
       printString,'\') withCrs.
       self changed: #reply2.


standInput3

"    evaluate selection for stand instance variables in list
     pane input3. "

      (choice3 == #dbh:)
          ifTrue: [
             list4Collection :=  PaneSelectors
             retrieveCollection:#dbhCollection.
             replyStream := Reply1Dictionary at:#dbh.
             self changed:#reply1].

      (choice3 == #species:)
           ifTrue: [
             list4Collection :=  WoodDensity.
             replyStream := Reply1Dictionary at:#species.
             self changed:#reply1].

      ( choice3 == #treeVolume:)
          ifTrue: [
           replyStream := Reply1Dictionary
           at:#promptMessage.
           self changed: #reply1.
           self treeVolumePrompt].

       ( choice3 == #acres:)
```

```
            ifTrue: [
             replyStream := Reply1Dictionary
             at:#promptMessage.
             self changed: #reply1.
             self acresPrompt].

          ( choice3 == #treeHeight:)
            ifTrue: [
             replyStream := Reply1Dictionary
             at:#promptMessage.
             self changed: #reply1.
             self treeHeightPrompt].

          ( choice3 == #merchTrees:)
            ifTrue: [
             replyStream := Reply1Dictionary
             at:#promptMessage.
             self changed: #reply1.
             self merchPrompt].

         ( choice3 == #unmerchTrees:)
            ifTrue: [
             replyStream := Reply1Dictionary
             at:#promptMessage.
             self changed: #reply1.
             self unmerchPrompt].

^ self
```

standInput4

```
"    evaluate stand input for input pane 4."

       (choice3 == #species:)
          ifTrue: [
           EnvironmentDictionary at:choice3 put:choice4.
           inputString := (inputString, choice3 printString,
           ' set to ', choice4 printString, '\') withCrs.
           self changed: #reply2 ].

        (choice3 == #dbh:)
          ifTrue: [
           EnvironmentDictionary at:choice3 put:( self
           convertStringToNumber:choice4).
           inputString := (inputString, choice3 printString,
           ' set to ', choice4 asSymbol printString , '\')
           withCrs.
           self changed: #reply2 ].

^ self
```

```
stateCollection

"  return stateCollection "

^ stateCollection.


treeHeightPrompt

"   Open a prompter window to set the tree Height."

" Temporary variables "

|tempVal temp|


 " get user input for the tree height"

tempVal := Prompter prompt: 'Tree Height.'
defaultExpression: '' point: 200 @ 250.

" Determine the appropriate operation to locate an object.
  Once the object is located, update the slope information."

     EnvironmentDictionary at:choice3 put:tempVal.
     inputString := (inputString, choice3 printString, ' set
     to  ', tempVal printString,   '\') withCrs.
     self changed: #reply2.


^self


treeVolumePrompt

"  Open a prompter window to set the tree volume."

"Temporary variables "

|tempVal temp|


 "get user input for the tree volume"

tempVal := Prompter prompt: 'Tree Volume.'
defaultExpression: '' point: 200 @ 250.

"Determine the appropriate operation to locate an object.
 Once the object is located, update the slope information."

     EnvironmentDictionary at:choice3 put:tempVal.
     inputString := (inputString, choice3 printString, ' set
        to  ', tempVal printString,   '\') withCrs.
        self changed: #reply2.


^self
```

unmerchPrompt

```
" Open a prompter window to set the density of
  unmerch trees ."

" Temporary variables"

|tempVal temp|


 " get user input for the unmerch trees"

tempVal := Prompter prompt: 'density unmerch Trees/acre'
defaultExpression: '' point: 200 @ 250.

" Determine the appropriate operation to locate an object.
  Once the object is located, update the slope information."

        EnvironmentDictionary at:choice3 put:tempVal.
        inputString := (inputString, choice3 printString, ' set
        to  ', tempVal printString,   '\') withCrs.
        self changed: #reply2.


^self
```

woodDensityEdit:aSymbol

```
"   Open a prompter window to set the density value ."

" Temporary variables "

|tempVal temp|


 " get user input for the value"

tempVal := Prompter prompt: 'density value'
defaultExpression: '' point: 200 @ 250.

" Determine the appropriate operation to locate an object.
  Once the object is located, update the density
  information. "
```

```
     WoodDensity at:choice3 put:tempVal.
     inputString := (inputString, choice3 printString, ' set
     to  ', tempVal printString,   '\') withCrs.
     self changed: #reply2.


^self


woodDensityInput2

"  evaluate selection for woodDensity in input pane 2."

      ( choice2 == #display:)
      ifTrue: [
              replyStream := Reply1Dictionary at:
              #woodDensityDisplay.
              self changed: #reply1.
              self displayWoodDensities].

     ( choice2 == #add:)
      ifTrue: [
            replyStream := Reply1Dictionary at:
            #woodDensityAdd.
            self changed: #reply1.
            self addToWoodDensity].

        ( choice2 == #edit:)
         ifTrue: [
           list3Collection:= WoodDensity.
           replyStream := Reply1Dictionary at:
           #woodDensityEdit.
           self changed: #reply1].

         (choice2 == #remove:)
           ifTrue: [
           list3Collection:= WoodDensity.
           replyStream := Reply1Dictionary at:
           #woodDensityRemove.
           self changed: #reply1].
^ self
```

woodDensityInput3:aString

```
"     Evaluate woodDenstiy for input pane 3."

        choice3:= aString asSymbol.
        (choice2== #edit:)
            ifTrue:  [
                self woodDensityEdit:choice3].

        (choice2== #remove:)
            ifTrue:  [
                WoodDensity removeKey:choice3.
                inputString := (inputString, choice3
                printString,  ' removed from wood
                densities.', '\') withCrs .
                self changed: #reply2.
                self changed: #input3].

 ^ self
```