DEDUCTIVE RETRIEVAL MECHANISMS FOR STATE DESCRIPTION MODELS

Richard E. FIkes
Stanford Research Institute
Menlo Park, California 94025

Abstract

This paper presents some programming facilities for modeling the semantics of a task domain and for describing the situations that occur in that domain as a task is being carried out. Each such description models a "state" of the task environment, and any given state can be transformed into a new state by the occurrence of an event that alters the environment. Such modeling systems are vital in many AI systems, particularly those that do question answering and those that do automatic generation and execution monitoring of plans. The modeling mechanisms described are basically extensions and modifications of facilities typically found in AI programming languages such as PLANNER, CONN1VER, and QA4. In particular, we discuss our use of a 3 valued logic, generator functions to deduce answers to model queries, the saving and maintaining of derived results, and new facilities for modeling *stntc* changes produced by the occurrence of events.

Introduction

This paper describes some modeling facilities that have been developed as part of a large Computer based Consultant system (CBC) being designed and implemented at the SRI Artificial Intelligence Center. The CBC is intended to serve the role of an expert consultant to a human apprentice doing maintenance and repair operations on electromechanical *devices.* The air compressor shown in Figure 1 is an example of such a device that has been used during the ear ly stages of the project, A complete overview of the project is given elsewhere in this volume in Reference 1.

The models that we are concerned about in this paper could be descriptions of any environment of interest at specific instances in time. Each such description is said to model a "state" of the environment, *and a* state can be transformed into a new state by the occurrence of an event that alters the environment. For the CBC, these models describe the state of the workstation including the device, tools, test equipment, etc., and the events are primarily maintenance and repair operations performed by the apprentice.

Programming facilities for querying state description models and for updating them to reflect the occurrence of *an* event are vital in many AI systems, particularly those that do question answering and those that do automatic generation and execution monitoring of plans. Planners, for example, use these models to simulate potential operator (event) sequences and investigate their consequences.

One of the goals for the CBC project is to have the system synthesize plans for transforming a device from any arbitrary state of assembly or disassembly into any other such state. This planning capability is required for component re-mo va 1 or replacement operations and, in general, whenever a collection of assembly and disassembly operations are needed to help accomplish a task. We will refer to this planning domain for examples throughout this paper.

The modeling mechanisms to be described here are basically extensions and modifications of facilities typically found in recent planning programs implemented in languages such as CONNIVEH (Ref. 2), PLANNER (Ref. 3), and QA4 (Ref. 4). The need for such additional knowledge representation mechanisms is evident as AI projects continue to move in the direction of considering more complex task domains. The presentations here are meant to add an increment to our ability to design and build such large systems.

Expressions and States

Our modeling system is implemented as an extension of the QLISP programming language (Ref. 5). [Note: QLISP is a direct descendant of the QA4 language and was designed to maintain QA4 language features while eliminating the inefficiencies introduced by the QA4 interpreter. It is a set of functions loadable into the INTERL1SP (Ref. 6) system, and QLISP "statements" are basically calls on these LISP functions. Hence, the user can freely intermix QLISP and LISP code. The control features that were lost by eliminating the QA4 interpreter are currently being restored by making use *of* new control facilities provided by the 1NTERLISP system (Ref. 7).l Each state description model can be thought of as a set of QLISP expressions, with each expression having a truth value associated with it. An expression typically is a statement of a relationship among entities in the task domain such as objects, concepts, or other relationships. For example, the expression (CONNECTED PUMP PLATFORM) is a statement that the pump is connected to the platform, and the expression (FASTENER (CONNECTION PUMP PLATFORM) B0LT:1) is a statement that BOLTrl is *a* fastener of the pump-platform connection.

Although each QLISP expression has a L1SP-style property list associated with it, we use

these property lists in our models only for truth values and bookkeeping information. Therefore, instead of putting GREEN as the value of the property COLOR on the property list of PUMP, we would create the expression (COLOR PUMP GREEN) and put TRUE as the value of the property TRUTHVALUE on its property list. This convention creates homogenous models and allows a pattern matching program to associatively retrieve any relationship in a model.

The QLISP context mechanism allows the system to build and manipulate a collection of state descriptions without having to create and maintain a complete copy of each state. A "context tree" is maintained in which each node denotes a state. To represent the new state that is produced by the occurrence of an event in some state Si, the system creates a new node in the context tree as a direct descendent of the Si node. All information in state Si that is not explicitly changed in the new state is assumed to also hold in the new state. That is, each state inherits model information from the state that is its parent in the context tree.

Querying State Description Models

Truth Values

A state description model is a source of information ahout a particular situation, and its primary use is as a data base for answering queries about the situation. Our modeling system interacts with its users (both people and programs) as if QLISP expressions with truth values attached were the only representations being used, even though the option exists for storing information in other forms. Hence, all queries from outside the modeling system concern the truth value of expressions in some given state.

When answering a query about a particular expression in some given state, the system searches for a truth value. The search begins with the expression's property list for the given state. If the property TRUTHVALUE has no value on that property list, the property list for the given state's parent (in the context tree) is checked. The search continues in this manner until a value is found or until all the states in the context are considered. If no value is found, the search returns UNKNOWN as its result.

Since any expression can be stored as the value of property TRUTHVALUE, this retrieval mechanism allows use of an N-valucd logic. For example, one could have "fuzzy" truth values represented as integers from -100 to +100. For our models, we currently are using a 3-valued logic that a llows the system to distinguish expressions that are "known true," "known false," or "have unknown truth value" in any given state. This is the simplist logic thai meets a modeling systems need since state description mode ls arc inhercntly incomplete, and it is important for the system to be aware of what it doesn't know as we1 1 as what it does know.

Generators Instead of Backtracking

QLISP provides facilities for associatively retrieving expressions from the data base that match any given pattern, where a pattern is defined to be an expression that may contain unbound variables. The QLISP statements for querying the data base use this pattern matching facility and are similar to the query statements found in PLANNER and QAA. They are designed to find a single instance of a given pattern. To cause the pattern matcher to continue its search and obtain another such instance, the user's program must return to the query statement via the language's backtracking mechanism (i.e., by "failing").

Using backtracking in this way to sequence through a class of expressions that all match a given pattern has severe limitations in that it ties the sequential production of each expression to the control structure of the user's program. In particular, it requires that the same portion of the user's program be executed for each expression (namely, the statements immediately following the query statement). Also, since all the backtrackable effects of that portion are being "undone" after each failure, it makes cumbersome the saving of results for each expression generated. Such a backtracking mechanism is best suited to a "generate-and-test" situation where the user desires a single expression that not only passes the query statement's tests, but also passes additional tests included in the user's program.

We have adopted the C0NN1VER solution to these limitations in our modeling system by providing functions that are generators of expressions from the data base. For example, there is a generator version of the QLISP IS statement called GEN:IS that finds instances of a given pattern having truth value TRUE in a given state. Each time a generator function such as GEN:IS is called, it produces as many expressions as is convenient for it. These expressions are put on a "possibilities list" along with a "tag" that indicates how the generator can be restarted when more expressions are requested, and this possibilities list is returned by the generator as its value.

If the function TRY:NEXT is called with a possibilities list as an argument, it will remove the first expression from the list and return it as a value. If the possibilities list contains no expressions, then TRY:NEXT attempts to produce new ones by using the tag to restart the generator. Since each call to TRY:NEXT can be made from anywhere in the user's program, generators of this form successfully separate the product ion of a

next data element from the processing that is done on each element.

Consider, for example, a set of queries concerning which components are connected to the pump in state Si. They can be initialized as follows:

(SETQ PL (GEN:IS (CONNECTED PUMP -C) Si)) .

Then whenever one of these components is needed, evaluation of (TRY:.NF.XT PL) will return a true instance of the pattern (CONNECTED PUMP-C) and will set the value of the QLISP variable C to be the "found" component.

We have implemented programming facilities to support the writing atu! use of generator functions using 1NTKRL1SP FUNARC's. A FUNARC is a data object that conceptua1ly represents a copy of *a* function and a private data environment for that copy. This FUNAKG imp]omentation a1lows the definition of a generator function to include a set of variables (i.e., a data environment) whose values will be saved and restored each time the generator is restarted. These "own variables" allow the generator fundion to save pointers indieating where it is in *its* search for generatable iterns. The FUNARG js added to the possibilities list *as* the "tag" that TRY:NEXT uses to restart the generator. Included in the implementation are CONNIVER-style functions such as NOTE, AU-REVOTR, ADIEU, and TRY:NEXT, which make the definition and use of generators convenient and practical.

FUNARG's do not provide a complete co-routine (process) facility because the control environment is not saved between calls, and therefore each restart necessarily returns control to the beginning of the function. Our experience with generators, however, has indicated that making available a FIRST:CALL? predicate function that can be called from inside a generator to distinguish initial calls from restarts removes essentially all of the hindrance caused by not having a saved control environment. Hence, this implementation of generators is simple and effective and can be made available in most LISP systems.

The Query Functions

We can now describe our model querying mechanism. Query functions are available called DEDUCE:ONE, DEDUCE:EACH, DEDUCE:ALL, REFUTE:ONE, REFUTE:EACH, and REFUTE:ALL. Each of these functions takes *a* pattern and a state as arguments. The DEDUCE functions find instances of the pattern that are true in the given state, and the REFUTE functions find instances of the pattern that are false in the given state. The :ONE functions find only a single instance and are not restartable; the :EACH functions are generators and return possibilities lists; and the :ALL functions return a list of all the findable instances.

Known truth values are usually not all explicitly stored in a model. Instead, the user typically provides derivation functions that compute them when they are needed. These functions may embody formal theorem proving strategies or simply be statements of implicationa1 rules derived from the semantics of the task domain. They serve to extend each model in the sense that, from the calling program's point of view, the derived instances of a pattern are indistinguishable front instances actually found in the model.

Our query functions are similar to a PLANNER or QA4 GOAL statement in that they first use the pattern matcher to find suitable instances of the pattern in the data base and then, if more instances are needed, they call user supplied functions to attempt derivations of the *desired* instances. These functions are assumed to be generators that produce derived instances of the pattern.

For example, a deduction function in the CBC system finds and generates true instances of patterns of the form (POSITIONED 'X -Y) by using DEDUCE:EACH to find true instances of the pattern (ATTACHED $X $Y), since components that are ATTACHED are assumed to be POSITIONED. Also, a refutation function finds and generates false instances of patterns of the form (POSITIONED -X -Y) by using DEDUCE:EACH to find true instances of the pattern (REMOVED $X $Y), since components that are REMOVED are assumed to be not POSITIONED.

Such derivation functions are the user's primary means of expressing the semantic links among the relations occurring in the state description models. Also, they can provide an interface to information that is stored in representations other than QLISP expressions. That is, it may be much more convenient and efficient to store some information in arrays, trees, or on disk files; deduce and refute actions serve as the access functions to these alternate data bases.

Storage and Retrieval.of Action Functions

The first element of each nonatomic expression in the model is assumed to be the name of a relation (or a QLISP variable that is to be bound to a relation). Therefore, the DEDUCE and REFUTE functions can use relation names as an index to determine which derivation functions should be called. Accordingly, we associate with each relation two lists of derivation functions that can derive instances of patterns that begin with the relation. One list contains the "deduce actions" used by the three forms of DEDUCE and the other contains the "refute actions" used by the three forms of REFUTE.

This simple indexing technique is used as an alternative to "pattern directed invocation" of the actions (which is available in QLISP).

Pattern directed invocation, where a pattern is associated with each action function and a pattern matcher determines which actions are applicable whenever a query is made, is a useful technique in many situations, but its power and importance should be looked at realistically. In this situation the indexing scheme seems preferable even though it has the disadvantages that an action's local variables do not automatically get bound to elements of the pattern and additional tests may be needed in an action to determine if it is applicable to the pattern.

However, the indexing scheme has the following advantages: (1) the use of the actions is made more efficient by significantly reducing the need for pattern matching; (2) each list of deduce and refute actions is typically quite short so that the user can easily determine an appropriate ordering of each list and similarly can determine where on the list a new action should be inserted; (3) and, finally, we have found that associating semantic information directly with each relation is a convenient way to conceptualize and localize the semantic information in the system. With this organization, the set of action functions associated with a relation specify essentially all the semantic information that the system knows about that relation.

Saving Derived Results

Overview

When a model query causes derivations to be attempted, we want the results of those derivations to be stored and retained in succeeding states as long as they remain valid. In this way the system achieves the maximum benefit from derivations and minimizes unnecessary rederivations. It is a simple matter to store a result In the state where a derivation is done, but more care must be taken if the result is to be made available in other states. In this section we will describe a set of mechanisms designed to provide maximum retention of derived information with a minimum of bookkeeping overhead.

A model query is an attempt to find true (or false) instances of a given pattern. Each time such an instance is determined, our DEDUCE and REFUTE query functions save the derived result by assigning a truth value to the instance (i.e., put it as the value of TRUTHVALUE on the expression's property list) so that the value will not have to be rederived if it is needed again. For example, if a deduce action for ASSEMBLED determines that the pump is assembled by querying the model about each of the pump's components, then the expression (ASSEMBLED PUMP) will be assigned a truth value of TRUE.

If a query is one of the :ALL forms, or if it is an :EACH form and the generation continues until all derivable instances of the pattern are produced, or if the query pattern contains no unbound variables (and therefore has only one possible instance), then the system also records the fact that all instances of the pattern have been derived. Then, if the same query is repeated, the system will know that the action functions cannot find any new instances and can therefore prevent ill-fated attempts at rederivation. For example, if during a query all the components that are positioned with respect to the pump have been found as instances of the pattern (POSITIONED PUMP -C), then when that information is requested in a later query, derivation functions such as the one that looks for components attached to the pump will not be recalled.

These "set completeness indicators" are also frequently useful to indicate the *case* where there are no derivable instances of *a* pattern. For example, if all derivation attempts are unsuccessful at determining whether the pump is assembled, then the set of derived instances is empty and marked as complete.

Saving and Maintaining Derived Instances

Our algorithms for maintaining these saved derived results in succeeding states depend on availability of the "support" for each derivation. The "support" for a derived instance is defined to be those expressions from the model that are used as axioms in constructing the derivation. For example, if an action function queries the model for the locations of two objects and concludes that one of the objects is above the other, then the locations of the two objects form the support set for the result. Actually, since any model query may return a derived result, the support set for the "above" result would be the union of the support sets for the two location expressions.

The user supplied deduce and refute actions are responsible for computing and storing support for each derived instance that they find. In almost all cases this is an easy task. For example, if an action embodies the implication "X implies Y," then the action can simply fetch the support for X and attach it to Y. GET:SUPPORT and PUT:SUPPORT functions are provided for performing these support set manipulations.

Functions are also provided for deduce and refute actions to call when they have derived a truth value for an instance of the query pattern and have determined the support for that derivation. These functions add the expression to the current possibilities list (remember that action functions are generators), set the truth value (i.e., TRUTHVALUE) of the instance, put the support set on the instance's property list, and put a pointer to the instance on the property list of each member of the support set. Use of these functions to "note" derived results implies that derived truth values can be found by the pattern matcher's search through

the data base at the beginning of a model query, and each expression that supports a dertvation will have on its property list a pointer to the expressions that it supports.

A derived result remains valid in succeeding states as long as its support remains valid. We therefore have the system do the required maintenance on derived instances in new states by including with the standard model updating functions (described below) the following facility. Whenever an expression with a known truth value has its truth value changed during a model update, the truth value of each of the expressions that it supports is set to UNKNOWN in the new state. The truth values of these "supportees" may not in fact have changed in the new state, but the derivations that made the truth values known are no longer valid. Hence, if a model query in the new state needs to know one of the deleted truth values, a new derivation must be attempted. For example, if (ATTACHED PUMP PLATFORM) is the support for (POSITIONED PUMP PLATFORM) and a detach pump from platform action causes a new state to be created, then the truth value of (POSITIONED PUMP PLATFORM) will be set to "unknown," the pump may still be in position on the platform, but the justification for the earlier conclusion about the pump's position is no longer valid.

The mechanism we have described thus far is effective, easy to use, and efficient for saving and maintaining derived truth values of expressions that are instances of a query pattern. Each derived instance is assigned a truth value, the support pointers are established, and when a state transition invalidates the derivation the derived truth value is deleted.

Saving and Maintaining Complete Sets

Consider now the maintenance of the "completeness" indicators that are attached to patterns when all derivable instances have been found. Specifying the support for such indicators poses some subtle problems, because the completeness depends not only on the validity of the derived instances but also on whether any other instances are derivable. This latter consideration implies that if any change should occur in a succeeding state that could possibly allow some action function to derive a new instance, then the completeness indicator should be removed from the pattern in the new state* For example, during the determination of all the components that are positioned with respect to the pump, some deduce action may have looked for components that are attached to the pump. Therefore, any succeeding state produced by *an* action that attaches a component to the pump will cause the removal of the completeness indicator from the derived positionings. Each derived instance will still be valid in the new state; only the completeness of the set is in doubt.

We have approached this problem of the validity of completeness indicators by keeping a record of all the model queries made during a derivation. If any of these queries will produce different results in a new state (i.e., a different set of derived instances), then the action functions that made the original derivation may behave differently in the new state and in particular may derive a different set of instances. Therefore, to maintain the completeness indicators we record during a model query (i.e., a call on one of the DEDUCE or REFUTE functions) all the other model queries that occur.

Consider a query to find instances of *a* pattern PO during which are executed queries Q1, Q2, ..., Qn to find instances of patterns P1, P2, ..., Pn. The system records for each query QI whether it is a deduction or a refutation and the pattern Pi associated with it. If all derivable instances of PO are found, then PO is given a completeness indicator, the set of derived instances is put on its property list, the queries QI, ..., Qn are put on "state transition test lists," and a pointer to pattern PO is put on the property list of each of the patterns P1, ..., Pn.

Whenever an expression with a known truth value is having its truth value changed during a state transition, a test is made to determine if it matches the pattern of any query on the appropriate state transition test list. If a match occurs, the completeness indicators of the patterns pointed to by the matching pattern are removed since this state transition may affect the completeness of those derivations. This entire mechanism is built into the DEDUCE, REFUTE, and model updating functions, so that the user who writes action functions need not be aware of it.

In Summary

To summarize, our model querying mechanisms provide for the automatic derivation of truth values for pattern instances when they are not explicitly stored in the model, for the saving *of* all such derived results so that similar later queries will not reinitiate the derivation functions, and for the maintenance of such saved results. Also, an important side effect of these mechanisms is that each derived instance has the support for its derivations computed and stored with it. Such support information has many uses because it indicates precisely which statements in the state model a particular precondition, subgoal, or action depends on (Ref. 8).

State Transitions

Updating Functions

The models of state changing operators that a system works with must contain sufficient information about the effects of each operator so that

they can be simulated and a description produced of the expected resulting state. As in most planning systems, we are assuming that the application of an operator in some state SO is modeled by producing a new state S1 that is conceptually an updated copy of SO (i.e., S1 is a direct descendent of SO in the QLISP context tree). The effects of the operator arc indicated by asserting, denying, and deleting expressions in the new state S1.

In our modeling system we provide the following set of model updating statements:

(S1M:ASSERT <expression <state>),
(S1M:DENY <-expression . <state>), and
(SIM: DELETE <-patterns <state).

SIM:ASSERT (SIM:DENY) changes the truth value of the given expression to TRUE (FALSE) in the given state. SIM:DELETE changes the truth value of all expressions that match the given pattern to UNKNOWN in the given state. These statements also call a sot of user supplied functions (like PLANNER antecedent theorems) that typically make additional changes in the new model that are direct results of the assertion, denial, or deletion being done. These user supplied functions play an important role in simplifying operator models in that they allow the user to express in one place side effects of particular assertions, denials, or deletions that always occur no matter what operator does them. In this way, these side effects do not have to be repeated in each operator that causes them to occur.

As in the case of the DEDUCE and REFUTE functions (and for similar reasons), we have elected to store the user supplied updating functions on each relation's property list. Hence, a relation can have a list of ASSERT:ACTIONS, DENY:ACTIONS, and DELETE:ACTIONS. These lists indicate how a model updating operation should proceed for an expression having the given relation as its first element.

The standard model updating functions (as found in QA4, PLANNER, QLISP, CONNIVER, etc.) apply user supplied functions only after the expression's truth value has been changed. Often it would be convenient to have user supplied functions applied before the expression's truth value is changed as well as after. For example, if a relation like LOCATION has certain uniqueness properties, then any assertion of an expression containing that relation (such as (LOCATION B0LT:1 3 4 5)) can automatically be preceeded by a denial (such as (SIM:DENY (LOCATION B0LT:1 <-LOC))). By allowing the programmer to specify such an automatic denial in an assert "pre-action," we remove the necessity for specifying the denial each time such an assertion is done.

We provide for both "pre-actions" and "post-actions" during model updating by giving the programmer the option of specifying when the truth value of the expression will be changed. This is done by including an asterisk (*) in the action list. The asterisk indicates when in the course of applying the actions that the expression's truth value is to be changed. Hence, all actions preceeding the asterisk are "pre-actions" and all actions following the asterisk are "post-actions."

When the updating functions encounter the asterisk in the action list, they also perform the maintenance operations on derived results. This means that if the expression had a known truth value and that value is being changed, then all the expression's supportees must be deleted. Also, the state transition test lists are checked to determine if any set completeness indicators should be removed.

Model Updating Using. Deduce and Refute Actions

The information in many deduce and refute actions can also be used to determine the secondary effects of assertions and denials. We would like for the system to make use of this information during model updating so that the user does not have to duplicate it in updating functions. For example, if the user has written a deduce action that embodies the rule "X implies Y," then we do not want him to also have to write a deny action for Y that removes from the model any truth values that could be used to derive X. The information necessary to do these changes at the appropriate time is included in the original deduce action.

Consider a consistency checking procedure that would make use of deduce and refute actions and that could be applied as a standard part of model updating. We can describe this procedure by indicating how it would work for denials. The truth value of the expression being denied would be set to UNKNOWN and an attempt would be made to deduce the expression. If this attempt produces a successful derivation, then the new state contains support for the truth of the expression even though it is being denied. The inconsistency can be eliminated by removing the support for the derivation. If the support set has exactly one expression in it, then that expression can have its truth value reversed. (This is the "X implies Y" case where denying Y is implying that X should also be denied.) The reversed truth value would be stored as a derived result with the original expression that is being denied (Y) as its support. When the support set contains more than one expression, we know that at least one of the expressions must have its truth value reversed, but we do not know which onc(s). Therefore, the best we can do is set to UNKNOWN the truth values of all the support expressions.

If the system knows which relations are changeable by events and winch ones are true in all states, then it can decrease unnecessary deletions by removing the unehangcable expressions from the support set before considering deletions or a truth value reversal- After the support for the derivation has been removed, a new derivation is attempted and the process is repeated until no new derivation can be tound.

This updating procedure does not guarantee consistency in the new *stalc* nor does it prevent later chanpes to the state from introducing new inconsistendos. However, it does automatically take care of many model updating details and it removes all existing inconsistoneios in the state that arc discoverablc by the system's deductive machinery. If the system cannot derive the facts from which an inconsistency follows, then the inconsistency is irrelevant and can safely be ignored.

Obviously, there are situations in wbich this procedure initiates computationally expensive derivations and causes many unnecessary deletions. Hence, it must be selectivcly applied. We have put the entire process under user control by allowing individual specification of which deduce and refute actions are employed to determine truth value deletions and reversals during model updating. This is done by providing a function UPDATE:W1TH that takes a deduce or refute action and an expression as arguments, and is used to specify an assert or deny action.

Note that using an "X implies Y" deduce action as a deny action for Y is not the same as writing a deny action for Y that simply denies X. The difference is that in the latter case X would be denied each time Y is denied, and the deny actions associated with X would then trigger off other assertions, denials, and deletions. Such a process could clutter up the model with many irrelevant implicat ions of the denial of Y. However, in the former case where the deduce action is *used as* the deny action for Y, no changes are made in the model if X cannot be derived; and if X can be derived, only the supporters of the derivation are changed. This means that only those truth values that are actually inconsistent with the denial of Y are changed; no irrelevant implications are stored.

The Relations AND, OK. and NOT

AND, OK, and NOT arc "built into" our modeling system in the sense that deduce, refute, assert, deny, and delete actions have been written for each of them. Whenever possible, conjunctions, disjunctions, and negations are decomposed into more primitive forms by the action functions. For example, the assert action for AND also asserts each of the conjuncts, and the deduce action for NOT strips off the NOT from the query pattern and attempts to refute the remaining pattern.

The refute actions for AND and OR translate the query into a call on DEDUC£:EACH by using the rules:

((NOT XI) AND ... AND (NOT Xn)) implies
(NOT (XI OR ... OR Xn), and
((NOT XI) OR ... OR (NOT Xn)) implies
(NOT (XI AND ... AND Xn) .

The deduce actions for AND and OR have an important role to play in that they are the overlords of the derivations of each conjunct or disjunct. They could each be expanded into full problem solving executives that would make use of coroutine facilities to explore alternative derivations in parallel and semantic information to determine the order in which conjuncts or disjuncts are considered. We have experimented with only unsophisticated versions of these actions, but the important point to note is that the query mechanism gives those actions control over the derivation so that the option is there to expand them when needed.

Summary

We have described a set of programming fact1-ities for building, maintaining, and querying state description models. These facilities are useful in systems such as planners, question answerers, and simulators. They allow the storage and retrieval of statements with true, false, and unknown truth values, and provide a programming environment that allows derivation rules embodying the semantics of a task domain to be easily added as functions to the system. These rules can also be used to assist in modeling the effects of an operator that creates a new state. Facilities arc provided to save the results of these derivation functions, and to delete the results in new states whore the derivations are no longer valid. Finally, the semantics of conjunctions, disjunctions, and negations are provided as a part of the system.

Acknowledgments

Many people at the SRI AI Center have contributed to the development of this system. 1 wish to especially thank Marty Rattner, Earl Sacerdoti, and Georgia Sutherland for their important help. The work reported herein was sponsored by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC04-72-C-008 with the U.S. Army Research Office.

References

1. Peter E. Hart, "Progress on a Computer Based Consultant," submitted to IJCAI IV.

2. Drew V. McDermott and Gerald Jay Sussman, The Conniver Reference Manual, AI Memo No. 259, MIT Project MAC, May 1972.

3. C. Hewitt, "Procedural Embedding of Knowledge in Planner,'' Proceedings of IJCAI, London (September 1971).

4. John F. Rulifson, Jan A. Derksen, and Richard J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning," Technical Note 73, SRI Project 8721 (November 1972).

5. Rene Reboh and Earl Sacerdoti, "A Preliminary QLISP Manual," Technical Note 81, SRI Project 8721 (August 1973).

6. Warren Teitelman, INTERLISP (Bolt Beranek Newman, October 1974).

7. Daniel G. Bobrow and Ben Wegbreit, "A Model for Control Structures for Artificial Intelligence Programming Languages," Proceedings of IJCAI, Stanford, California (August 1973).

8. Richard E. Fikes, Peter E. Hart and Nils J. Nilsson, Learning and Executing Generalized Robot Plans", Artificial Intelligence, Vol, 3, pp 251-288, (1972)

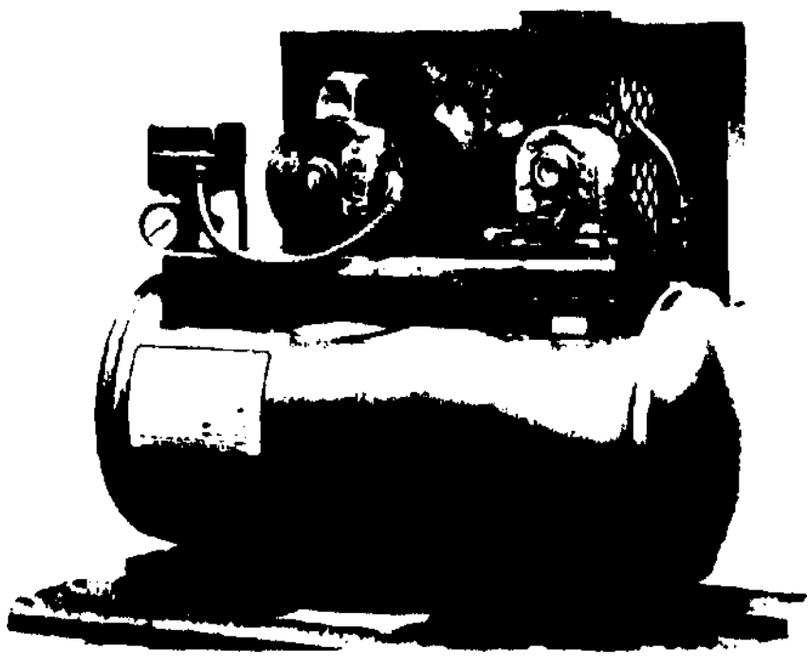

FIGURE 1 AN AIR COMPRESSOR SYSTEM HAVING ELECTRICAL, MECHANICAL AND PNEUMATIC COMPONENTS