Fault Diagnosis Based On Causal Reasoning

by

John Douglass Hodjat Whitehead

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science Applications

APPROVED:

J. W. Roach, Chairman

D. P. Miller

D. M. Kervick

November 13, 1987

Blacksburg, Virginia
Fault Diagnosis Based On Causal Reasoning

by

John Douglass Hodjat Whitehead

Committee Chairman: John W. Roach
Computer Science

(ABSTRACT)

A "causal" expert system based on hypothetical reasoning and its application to a Mark 45 turret gun's lower hoist are described. HOIST is a system that performs fault diagnosis without the use of a domain expert or "shallow rules". Rather its "knowledge" is coded directly from a structural specification of the Mark 45 lower hoist. The technology reported here for assisting the lesser acquainted diagnostician differs considerably from the normal rule-based expert system techniques: it reasons about machine failures from a functional model of the device. In a mechanism like the lower hoist, a functional model must reason about forces, fluid pressures and mechanical linkages, that is, qualitative physics. HOIST technology can be directly applied to any exactly specified device for modeling and diagnosis of single or multiple faults. Hypothetical reasoning, the process embodied in HOIST, has general utility in qualitative physics and reason maintenance.
Acknowledgements

I would like to thank my father for his love of learning.

This research was supported in part by grants from the Virginia Center for Innovative Technology, the Northern Ordnance division of FMC Corporation, Phillip Morris USA, and Texas Instruments.
Table Of Contents

1. Introduction .................................................................................................................. 1
   1.1. Problems Of Conventional Expert Systems ............................................................... 3
       1.1.1. Unanticipated Faults .................................................................................. 3
       1.1.2. Development With Minimal Expertise ............................................................ 3
       1.1.3. Maintainability ......................................................................................... 4
   1.2. Problem Domain: The Mark 45 Lower Hoist ............................................................. 4
   1.3. A Simple Example ................................................................................................... 5

2. Literature Survey .......................................................................................................... 8
   2.1. Related Work In Qualitative Reasoning .................................................................. 8
       2.1.1. A Qualitative Physics Based On Confluences ................................................. 8
       2.1.2. The Use of Design Descriptions in Automated Diagnosis ................................ 9
       2.1.3. Diagnostic Reasoning Based on Structure and Behavior ............................... 9
       2.1.4. A Theory of Diagnosis from First Principles .................................................. 10
       2.1.5. Counterfactuals For Common Sense Reasoning ............................................ 10
   2.2. Related Work in Counterfactuals ............................................................................. 11
       2.2.1. Rescher's Information Structure .................................................................... 11
       2.2.2. Rescher's Algorithm ................................................................................... 13
       2.2.3. Rescher in Summary .................................................................................... 15

3. Approach ...................................................................................................................... 18
   3.1. Qualitative Physics ................................................................................................. 18
   3.2. Causal Reasoning ................................................................................................... 19
   3.3. Causal Influence Is Local ..................................................................................... 19
   3.4. The Level Of Abstraction ...................................................................................... 21
   3.5. Unified Postdiction and Simulation ........................................................................ 22

4. WIF: Hypothetical Reasoning ....................................................................................... 23
   4.1. Uses ....................................................................................................................... 23
   4.2. Problem formulation ............................................................................................... 24
   4.3. A Program .............................................................................................................. 24
       4.3.1. Facts ........................................................................................................... 25
       4.3.2. Rules .......................................................................................................... 25
       4.3.3. Truth in WIF ............................................................................................. 27
1. Introduction

Expert systems have been identified as a potential tool in fault diagnosis and repair of complex systems. They hold the promise to assisting lesser skilled personnel with the automated expertise of the most skilled. This thesis describes an innovation in expert systems technology to perform postdiction [i.e. diagnosis of a cause after a failure]. It requires a computer model of the device; this model is used as a platform to make inquiries such as, "What if the device exhibited this behavior, with these things known. How could this be possible?". In short, it computes possible device faults based on the expected behavior of its subcomponents. This approach circumvents the necessity for a repair expert to "cook" his knowledge down to a fault decision tree as is common in conventional expert systems for fault diagnosis.

This new expert systems approach grew out of research in qualitative physics and the philosophy of causality, and uses hypothetical reasoning. Expert systems that incorporate causal reasoning represent a second-generation approach to providing diagnostic assistance. This thesis bridges the gap between theory and practice by tackling the difficult problem of postdiction on a complex real world machine. HOIST performs diagnostic reasoning (postdiction) on a faulty hydraulic/mechanical/electrical machine, the lower hoist of a Mark 45 Naval turret gun.

Postdiction and simulation are unified in this thesis as a single theory of device analysis. Postdiction of a single fault is a special case of postdiction of multiple faults. Further, device simulation is a special case of postdiction; namely analysis with no device malfunction. So, the approach presented in this thesis is quite useful, in that a system constructed for postdiction automatically has the capability to perform simulation. FMC Northern Ordnance has utilized this capability; they use the HOIST program as a simulator to drive a graphics interface for a training application.

Conventional rule-based expert systems for diagnostic advice heuristically classify the cause of failure from malfunction symptoms[3]; they have no model of the object for which advice is being given. In HOIST, a system for counterfactual reasoning, WIF (What IF), is used to model the functionality of all of the components thereby creating a "causal model" of the Mark 45 lower hoist. This model can not only simulate the functionality of the lower hoist when it operates correctly, but it can simulate failures as well.
FIGURE 1: Graphical display of the lower hoist as specified by HOIST, a causal reasoning expert system. Animated graphics was possible because the second generation expert system computes the internal states of the lower hoist. Picture courtesy of FMC Corporation.
The technology presented in this thesis required work on three primary fronts. It required a theory of fault diagnosis. This theory requires information generated from a qualitative model of the device of diagnosis. And an engine for hypothetical reasoning was necessary to develop the qualitative model.

1.1. Problems Of Conventional Expert Systems

Conventional expert systems, sometimes called "shallow reasoning" systems, have at least three major shortcomings in fault diagnosis. First, shallow reasoning is incapable of handling unanticipated faults. Second, there exists a significant lag time between initial construction of a device and the development of a conventional expert system for maintenance. Third, devices commonly go through a series of design modifications, and it is unclear whether a shallow reasoning fault advisor is still correct after such modifications. What follows is an analysis of these problems, and a hint as to how a model based expert systems may solve them.

1.1.1. Unanticipated Faults

Conventional expert systems cannot handle unanticipated situations. Typically, if a fault has not been anticipated, shallow reasoning diagnosis either halts with an incorrect answer or gives back little or no information on the suspect components: the performance of a conventional expert system does not degrade gracefully. A causal expert system uses expected versus known machine states to converge on a faulty component and thus has the potential to tell the user that a fault exists between two points.

1.1.2. Development With Minimal Expertise

The notion of conventional expert systems is based on having an expert to emulate. In the field of fault diagnosis, the machine must break several times in order for an expert diagnostician to exist. To be of use to a knowledge engineer, a diagnostician must adequately understand the device and be able to articulate knowledge of frequently observed and probable faults. For a complex system, often a period of years will be needed to produce an experienced expert who adequately understands observed and probable faults. If it then takes an additional year to produce a system to emulate the expert, there is a substantial lag between the time when the machine is first produced and the time the expert system for maintenance is made available.

An expert system that reasons from first principles (i.e. causal reasoning) requires a specification of the functionality of the device's components to diagnose faults. Anyone
with the working knowledge of a mechanical engineer could obtain this information from the device's blueprints, so the potential exists for a new product to be sold with its repair advisor included in the package.

1.1.3. Maintainability

Alterations are part of the evolution of producing modern machines. Some original designs are modified as components are found to be overstressed in field testing. Also, high sales volume tends to spawn a series of similar products, each with its own enhancement. It is unclear how correct a shallow reasoning maintenance advisor would be after such modifications have been made. Experience with R1[2] shows that maintenance of the knowledge can become quite burdensome. Conventional expert systems are produced by observing the expert and imitating his behavior. This behavior is a result of value judgements based on the device as it existed when the expert was interviewed. Some judgements are made unbeknownst to even the expert. If the device at some later time is slightly modified, how is the knowledge engineer to know how many value judgements are affected? Is the knowledge engineer to reconstruct the entire knowledge base each time an alteration is made? Causal reasoning is based on the structure of the device. If a functional alteration is made, the model must be updated accordingly. That is, in the model the existing part's structure must be altered to represent the new part's structure. Causal reasoning systems for fault diagnosis are easily maintained.

1.2. Problem Domain: The Mark 45 Lower Hoist

The lower hoist is part of a naval cannon; it is the transfer mechanism from the ammunition storage room to the gun's ready magazine. Due to present day machine complexity and poor retention rate of skilled repair experts, the Navy has set computer-assisted maintenance of existing machinery as a priority. The lower hoist has approximately one hundred and fifty components, most of which are pipes, but it also contains two solenoids, seven pistons, three latches, four state-detecting switches, a linkage, a chain, a hydraulic rack and a clutch mechanism.

Figure 1 is a hydraulic schematic of the lower hoist. The purpose of the lower hoist is to move ammunition from a lower deck to the ready magazine; the ammunition would rest on the pawls attached to the chain on the lower right portion of the figure. It is powered by a rack that is driven up and down by hydraulics (lower left). The solenoids (upper left) control this mechanism, they convert electrical signals to hydraulic pressure. Through a web of pipes and pistons the rack, latches and a clutch are controlled to drive the
chain in a four step lock-stitch cycle. In phase 1, the rack is down, the clutch is engaged (lower middle) thus connecting the rack to the chain and the latch is released (lower right). Phase 2 drives the rack up to the top so as to move the chain in the desired direction (and thus ammunition resting on the chain's pawls up). Phase 3 engages the latch thus immobilizing the chain, and disengages the clutch thereby disconnecting the rack from the chain. In phase 4, the rack is driven back down (without the chain) so as to prepare to repeat the cycle once again.

1.3. A Simple Example

In order to perform fault diagnosis, all of the components in Figure 1 had to be functionally modeled. This model constrains the internal states of the machine, in that under normal circumstances the internal states can be predicted. And if there is a failure, an enumeration of forms of failure can be computed (this enumeration of faults is subject to the level of detail provided by the model.) For example, a pipe might be represented as:

\[
R1: \quad (\text{(device pipe1 *p1 *p2)} \\
\text{(status *p1 *value1)} \\
\text{(status *p2 *value2)} \\
\text{->} \\
\text{(*value1 = *value2))}
\]

This rule states that for the pipe pipe1 that connects *p1 and *p2, if the pressure at *p1 is *value1 and the pressure at *p2 is *value2, then it had better be the case that *value1 is equal to *value2. Note that any token preceded by a * is a variable and may take on any value; the only constraint maintains that a variable is bound to the same value throughout a rule. The formal reading of a rule is as follows: for each <variables found in the preconditions of the rule>, if <the preconditions are in the database> then there exists <variables found solely in the conclusions of the rule> such that <the conclusions are in the database>.

\[
R2: \quad (\text{(status *point *value1)} \\
\text{(status *point *value2)} \\
\text{->} \\
\text{(*value1 = *value2))}
\]

R2 requires each check point to assume at most one value. For example, for each point, for each *value1, for each *value2 associated with the *point, if (status *point *value1) and (status *point *value2) are both in the database, then *value1 is equal to *value2.
R3: ( (checkpoint *point) 
   -> 
   (possiblevalue *value) 
   (status *point *value) )

R3 states that for each checkpoint, there exists at least one of the possiblevalues such that the status of the checkpoint is that value. [e.g. for each *point, if (checkpoint *point) is in the database, then there exists a *value such that (possiblevalue *value) and (status *point *value) are both in the database].

Let us suppose that normally (device pipel pl p2), (checkpoint pl), (checkpoint p2), (possiblevalue high), (possiblevalue low), (status pl low), (status p2 low) are true, corresponding to "pipe1 is a pipe between p1 and p2", "p1 is a check point", "p2 is a check point", "a possible value of a check point is high", "a possible value of a checkpoint is low", "the pressure at p1 is low", and "the pressure at p2 is low".

Now, if it were observed at p2 on the real world machine that the pressure was high, then (status p2 high) would be hypothesized of the model. Consider R1, it could be instantiated with (device pipel p1 p2) (status pl low) (status p2 high) -> (= low high). In order to preserve this rule, either a precondition must be negated, or the conclusion must be asserted. Since (status p2 high) is the assumption, either negate (device pipel pl p2), negate (status p1 low) [e.g. remove one of these from the database] or assert (= low high) [e.g. include it in the database.] By similar reasoning using R2, either ~(status p2 low) or (= high low) [i.e. where ~ means negate.] And in any case where ~(status p1 low) is true, then (status p1 high) must also be included as a result of R3.

The entailments from the hypothesis (status p2 high) are stated in disjunctive normal form below:

a) ~(device pipel p1 p2) [from R1] and ~(status p2 low) [from R2]  
b) ~(device pipel p1 p2) [from R1] and ~(= high low) [from R2]  
c) ~(status p1 low) [from R1] and ~(status p2 low) [from R2] and (status p1 high) [from R3]  
d) ~(status p1 low) [from R1] and (= high low) [from R2] and (status p1 high) [from R3]  
e) (= low high) [from R1] and ~(status p2 low) [from R2]  
f) (= low high) [from R1] and (= high low) [from R2]

this answer is actually generated by WIF in a tree form as follows:

( (status p2 on)
  =>
  (and (or (not (device p1 p2))
        (not (status p1 low)))
  =>)
Of course, we are normally interested in preserving the meaning of equality, so WIF provides a means for stating the relative "believability" of facts. This is described later, but it allows one to eliminate possibility b, d, e, and f when performing fault diagnosis. Further, it allows one to eliminate possibilities a when simply using the model as a simulator of correct functionality.

Conventional rule-based expert systems attempt to capture an expert's opinion. This thesis describes HOIST, a causal reasoning expert system that significantly differs from this approach since it does not rely upon advice; it instead uses a qualitative model of the physics of the device. The topography of the model itself defines a search space of possible device failures. Thus a causal expert system reasons from first principles. This approach to fault diagnosis requires no expert repairman, and thereby circumvents several of the problems with conventional expert systems for postdiction.

The technology presented in this thesis required work on three primary fronts. It required a theory of fault diagnosis and simulation. This theory uses information generated from a qualitative model of the device of diagnosis. And an engine for hypothetical reasoning was necessary to develop the qualitative model.
2. Literature Survey

2.1. Related Work In Qualitative Reasoning

Work in this thesis was in part inspired by readings in the Artificial Intelligence Journal's "Special Volume on Qualitative Reasoning about Physical Systems"[1]. In this issue, a number of authors discuss similar theories of qualitative reasoning to different ends. Most authors agree that the machine/situation/process being modelled must be divisible into small parts whose individual functions describe the functionality of the entire machine/situation/process when taken as a whole. Further, they all adhere to the principle of locality; that is, any two subcomponents not causally connected act independently and no single sub-component determines the functionality of the entire machine/situation/process. Several of the articles of this volume of the Artificial Intelligence Journal will be discussed, as they are related to the work found in HOIST.

2.1.1. A Qualitative Physics Based On Confluences

Both J. DeKleer and J. S. Brown have authored a number of articles on qualitative physics. The paper "A Qualitative Physics Based on Confluences"[5] brings together much of their work in the field.

The mechanics of a pressure regulator are modelled in this paper as a set of three state variables. If X is a state variable, then X can assume one of the values of +, 0, or -, based on whether the value of X is positive, zero or negative. Or if Z is some value and one is interested in Z's relative value to threshold Y, then X = Z - Y will define X with exactly this property (i.e. If Z is greater than Y then X is +, etc.).

A hydraulic pressure regulator is tricky to model since it is a feedback device and such a mechanism requires some level of understanding of time. DeKleer and Brown opted for the simplest possible qualitative representation of time, a three state variable dX (i.e. a short hand notation for dX/dT), either +, 0, or - based on whether a variable X is increasing, staying the same, or decreasing.

The functionality of the regulator is defined by status variables and relationships between them. For example, the simplest of all subcomponents is the pipe. A pipe might be defined by end-points X and Y. In order for the pipe to function correctly, X must equal Y (i.e. The pressure at one end of a pipe must equal the pressure found at the other.) Further, dX must also equal dY. In this way, components are defined and connected together as required by the topography of the mechanism (i.e. in this case, a pressure
DeKleer and Brown define the relationships between variables as qualitative calculus in that they are to embody the qualitative equivalence of quantitative equations. The complete specification of interrelationships of subcomponents define the functionality of entire device. Further, the functionality of the entire device can be explained, if asked, by the interrelationships of the subcomponents. In this way, DeKleer and Brown have developed an expert system that can predict and explain the behavior of a pressure regulator.

Once the device is modelled, DeKleer and Brown suggest the model can be put through all possible scenarios. Thus the behavior of the mechanism can be encoded into a simple transition system. Behavior prediction from the transition has the advantage of computational efficiency, at the cost of behavior explanation.

2.1.2. The Use of Design Descriptions in Automated Diagnosis

Genesereth uses a model based approach to diagnose faults in mechanisms. An electric circuit is used in his paper, although he has used the technique to diagnose faults in the cooling system of a nuclear reactor. Genesereth was interested in developing an efficient, general alternative to standard test-generation algorithms based on boolean algebra.

The macro structure of Genesereth's diagnosis system is a generate and test loop. His system DART would generate a set of possible failures and instruct the user to perform a test. The results of this test were then fed back into DART for further diagnosis.

The technique used in DART to generate possible failures is constraint satisfaction. The topography of the device specifies constraints between variables. These constraints are used to check consistency of possible failures. Genesereth also includes the single fault and non-intermittent fault assumptions to reduce the number of possibilities and thus computation. DART does not completely generate all possible worlds where the fault exists, instead it computes until it locates a distinguishing characteristic of the worlds where the fault exists. For example, if DART determines that some possible worlds that contain a fault that would explain the failure have $A$ true in them, and other possible worlds that also could explain the failure have $\neg A$; then DART will immediately ask the user to test $A$. DART will not move on to compute whether $B$ is true in some of the worlds where $A$ is true, etc. This was a design decision to reduce computational complexity.

2.1.3. Diagnostic Reasoning Based on Structure and Behavior

Randall Davis suggests that the connections between components should be modelled at different levels. Basically he is interested in the structuring of information
about a device so as to allow the quick solution of simple commonplace failures. However, he is not willing to sacrifice the capability of diagnosing more difficult problems. His solution is to define several different layers of topography, and systematically relax the constraints of each layer as they are ruled out.

Davis uses as his example, an electric circuit on a circuit board. One might consider a couple of different classes of problems which have to be addressed, such as: improper input values, short circuits between adjacent pins on a chip mounting, and thermal coupling between adjacent circuit board traces. The first class is more common and more easily diagnosed than the second which is in turn more easily diagnosed than the third. Each class of problems would require a topography to define the relationships between the components. In order to diagnose faults where the input values to the device are improper, one needs to define the logical adjacency of traces in the circuit. In order to diagnose faults of short circuits between adjacent pins the physical adjacency of traces at pins of chips must be defined. Further, the physical adjacency of circuit traces must be defined to diagnose problems of thermal coupling.

Since the first class of problems has the simplest test, and is found more frequently, one should initially assume that the problem is of this class. If fault diagnosis fails to provide an adequate explanation of the symptoms, then that assumption should be relaxed, so as to allow the faults of the second class to be diagnosed. In this fashion, an engine for diagnosis will only attempt to solve hard problems if in fact the problems are hard.

2.1.4. A Theory of Diagnosis from First Principles

Raymond Reiter introduces a notation in order to define a diagnosis[17]. From this definition he derives statements about the nature of a diagnosis, and their relation to the other newly defined entities: conflict sets and minimal hitting sets. Reiter then defines an algorithm to produce a diagnosis.

Reiter's work is interesting in that he has attempted to define diagnosis in terms of set theory. The bulk of the prologue is filled with impressive looking theorems. But, if one assumes that there exists an immobile object, it should be no surprise that a proof exists concluding that there is no irresistible force. Similarly the definitions of diagnosis, conflict sets, and hitting sets themselves state Reiter's theorems. The real worth of Reiter's work is the insight provided by his abstract definitions. As he is able to relate the works of DeKleer, Williams, Davis, and others to his own.

2.1.5. Counterfactuals For Common Sense Reasoning
Matthew Ginsberg[9] was the first author to publish details of a computer implementation of counterfactuals. His work follows from the writings of David Lewis, author of the 1973 book *Counterfactuals*[14]. The resultant program was used to diagnose faults in an adder circuit.

Ginsberg's work is similar to WIF, at least in concept. Quite frankly, the author had a hard time understanding how Ginsberg accomplished anything useful without the property of transitivity, an inference form strictly forbidden by Lewis. Further, Ginsberg left his theoretical foundation in subsequent writings. In "Reasoning about Action"[10], Ginsberg and Smith discuss their approach to the qualification problem, the use of counterfactuals to generate possible worlds. Conspicuously absent from the paper is the "If I leave by 6 O'clock" example. It is highlighted in Lewis's "Counterfactual Fallacies" chapter[14], and yet Ginsberg and Smith would use counterfactuals to solve the problem. It is clear that Matthew Ginsberg is not using David Lewis' counterfactuals and thus he does not have the theoretical foundation professed in "Counterfactuals For Common Sense Reasoning."

2.2. Related Work in Counterfactuals

For years, philosophers and logicians have studied hypothetical reasoning under the name "counterfactuals". The name is derived from the presumption that a hypothesis may be counter-to or inconsistent with a "believed" set of facts. The works of Nicolas Rescher[20] and David Lewis[14] characterize contrasting theories of counterfactuals in philosophy. WIF, a language for hypothetical reasoning devised in this thesis follows from the works of Rescher.

Nicolas Rescher provided an algorithm that, given a consistent set of facts, remaps truth values to the facts as necessary to allow union with a set of counterfactuals[20]. Each such reassignment is a solution, as each such reassignment allows the counterfactuals to be true. Furthermore, Rescher requires an ordering of the facts of the world based on relative "believability." This ordering is useful in that, laws of the world tend to be believed over transient facts; for example, the fact that lights may either be on or off but not both or neither, is more likely to be correct than a statement that the light in the bathroom is currently off. Using this ordering, relative believability of solutions can be determined.

2.2.1. Rescher's Information Structure

Counterfactuals imply contradictory information. Facts can only be contradictory if they break the stated relationships between them. These relationships are rules. If two
facts contradict a rule, either one fact, the other fact, or the rule that relates them must be removed to restore consistency. (Note that the removal of the fact "there is no bird here" is equivalent to the inclusion of the fact "there is a bird here". Thus the removal of facts does not imply the reduction of the number of believed facts.) But how is one to know which facts to remove?

Example[20]:

(1) Caesar was not a lion (known fact)
(2) Caesar had no tail (known fact)
(3) All lions have tails (accepted covering law)

Counterfactual: Caesar was a lion

[Case 1]: If Julius Caesar had been a lion, he would have had a tail.
[Case 2]: If Julius Caesar had been a lion, there would have been a tail-less lion (because Caesar had no tail).

Notice that belief (1) must obviously be set aside in view of the assumption, but that we have an option between rejecting (2) and (3). In [Case 1] we reject (2), and in [Case 2] we reject (3). However, [Case 1] is "natural" and [Case 2] is "artificial" because the nomological use of counterfactuals represents a determination to retain the appropriate generalization - that is, (3) - at the cost of adapting all else to it.

In order to cope with multiple solutions where one solution is more desirable (i.e. "natural") than others, Rescher assumes a categorization of facts in the world; Rescher introduces the notion of modal categories. The modal category of a fact explicitly represents the fact's believability relative to other facts. A fact with modal category i is more believable than a fact with a modal category greater than i. This forms a partial ordering (i.e. facts of the same modal category are not ordered relative to each other) on the facts of the world that assists in determining the most preferred solution to the counterfactual. The most preferred is the solution that violates only lesser believed facts (i.e., facts with higher modal category).

Example of modal categories forming a preference

<table>
<thead>
<tr>
<th>Lowest modal category</th>
<th># of facts removed</th>
</tr>
</thead>
</table>
where a fact is removed | at that modal category
--- | ---
Solution A -- | i | 5
Solution B -- | i-1 | 1
Solution C -- | i | 3

Preference order from most to least --- C, A, B

Solution A violates facts of a modal categories greater than or equal to i. Solution B violates facts of modal categories i-1 and greater. Solution A would be preferred over solution B since solution A retained the more believable facts of the world at modal category i-1 that solution B threw out. Furthermore since solution A threw out five facts at modal category i and solution C threw out only three facts of modal category i, solution C would be preferred over solution A, both of which would be preferred over solution B.

In the Caesar/lion example, (3) would have a lower modal category than (1) or (2) since (3) is more believable. Thus [Case 1] is more preferred than [Case 2], thereby exhibiting the desirable behavior.

2.2.2. Rescher's Algorithm

Solution generation has not been described, only solution preference. Rescher defines what he calls a Preferred Maximal Mutually-Compatible subset (PMMC), and provides a method of deriving PMMC.

Basic Assumption: We presuppose that we have at hand a selection procedure serving the following function: Whenever a group of statements $P_1, P_2, ..., P_n$ contains at least one non-contradictory statement, this selection procedure will select at least one (though possibly several) "preferred" maximal mutually-compatible subset (PMMC-subset) of the group of the $P_i$, this being a set such that we have it that (inter alia) whenever \( \{P_1^*, P_2^*, ..., P_m^*\} \) is any one of these subsets, then: (i) the $P_i^*$ are mutually compatible, and (ii) none of the nonincluded $P_i$ can be added to the $P_i^*$ without resulting in the inconsistency of the enlarged set.

To restate Rescher, no member of the initial set $P$ that is not already a member of a PMMC can be added to that PMMC without contradiction. This property accounts for the terms "maximal" and "mutually compatible", but not for "preferred". The preference of one
maximal mutually-compatible subset over another will be clarified by the algorithm for generating PMMC subsets.

Rescher's answer to a counterfactual in a set P is the set of all PMMC subsets of P, the "preferred" consistent subsets of P which contain the counterfactual. Before the algorithm is presented, the user should note that Rescher reserves modal category 0 (M₀) for the counterfactual, and modal category 1 (M₁) for the "laws of the universe." M₁ is simply a reserved modal category for facts a user is not interested in questioning, such as perhaps the properties of gravity, motion, etc. As a result, any PMMC subset must contain all elements of initial set P which are of modal category M₀ or M₁. Rescher's algorithm is as follows:

**Modal Categories as Providing a Procedure for the Selection of PMMC-Subsets of Inconsistent Sets of Premisses**

Assume that we have in hand a set S of statements and a family M₀, M₁,..., Mₙ of modal categories for this set. Suppose now that we are given a set {P₁, P₂,..., Pₖ} of mutually inconsistent premisses (of which at least one is not self-contradictory) drawn from S. We shall now specify a procedure for selecting a PMMC-subset of the inconsistent set {P₁, P₂,..., Pₖ}:

1. **Step 0:** Form the subset S₀ of the set P = {P₁, P₂,..., Pₖ} by selecting from this set all of those statements belonging to modal category M₀.
2. **Step 1:** Form the set S₁ by adding to S₀ all of the statements of P belonging to modal category M₁.
3. **Step i+1:** Suppose that Step i has yielded the sets S₁, S₂,..., Sᵢ. Then to each of these, add the maximum number of statements of P belonging to modal category Mᵢ₊₁ whose addition generates no inconsistency, doing this for each of the ways in which this step can be accomplished.

This algorithm terminates after n steps, where n is the number of modal categories found in P. It produces a set of PMMC-subsets by successive addition of maximal subsets of P by modal category.

Rescher's algorithm contains the "preference" automatically. Any PMMC-subset which sacrifices a fact of Mᵢ for facts of modal category greater than i, will never be generated. This follows from the algorithm because facts of Mᵢ will be maximally appended to PMMC-subsets before facts of modal category greater than i will be considered. The preference in PMMC-subsets removes the subsets from PMMC-subsets P₁ that satisfy the following: there exists a fact in initial set P not in P₁ that is consistent with all facts in P₁ of less than or equal modal category.
One can visualize the set of PMMC-subsets generated by this algorithm as a recursive daisy (see Figure 2). Each modal category i spawns a set of k pedals for modal category i+1 representing the k unique sets generated by step i+1 in the algorithm.

If we trace the Caesar/lion example, we will observe that the "natural" solution is in fact the only PMMC subset. The facts of the example; (1), (2), (3), and the hypothesis will be of modal categories M3, M3, M2, and M0 Respectively. These categories are appropriate since the hypothesis is always placed in M0, and (3) is more believed than (2) or (1).

Step 0: P = {hypothesis, (1), (2), (3)}.
Form S0 as {hypothesis}.
Step 1: There are no elements of P at M1, so
S1 = S0.
Step 2: There are two possible subsets of elements of P at M2, these are {} and {(3)}. However, {} is not maximally consistent since (3) may be consistently added to S1 union {}. The remaining possibility generates:
S2 = {hypothesis, (3)}.
Step 3: There are two elements of P at M3. Neither of these are consistent with the set S2. So the empty set is the only subset of elements of P at M3 consistent with S2.
S3 = {hypothesis, (3)}.

Only the Preferred MMC subset is generated, Case 1. Case 2 would have been generated only if Step 2 would have used the subset {} to generate an S2.

2.2.3. Rescher in Summary
Rescher provides an algorithm for generating solutions to a hypothetical introduced into an otherwise consistent set. This subset of all possible solutions are "preferred" in that they specify tolerable alterations to the original set. Tolerability is based on a partial ordering of all knowledge reflecting relative believability. Rescher devised a clever mechanism that exhibits hypothetical reasoning.

Rescher provides an appealing algorithm, but he has no semantics to support it. Questions arise such as: "What does consistency mean in this non-monotonic domain?" "What is really going on when Rescher builds a PMMC subset?" and "How do I know that a PMMC subset is desirable and not another kind of Maximal Mutually Consistent
subset?" Without a semantics there is no common ground for truly understanding and evaluating Rescher's counterfactual reasoning system.

Rescher's algorithm does not adapt well to large worlds. Using his algorithm, each and every element in the initial world P must be reconsidered, since PMMC subsets are generated by successive additions from P starting from the empty set. In a world of 500 rooms, must all the truths of the remaining 499 rooms be reconstructed when a robot turns on a light in room 1? Rescher's algorithm is prohibitively expensive for real world problems. In Roach, Eichelman, Whitehead[18], a new algorithm is described wherein, only alterations to the initial world are computed and represented as "difference trees."
Figure 2: The set of all PMMC subsets (solutions) pictured as a recursive daisy. The union of sets in a path from the center to an outermost pedal represents a single PMMC subset.
3. Approach

Modeling in HOIST required implementation of some form of qualitative physics, enough to be able to perform causal reasoning. Both qualitative physics and causal reasoning are closely related and have been characterized elsewhere[5,6]. Essentially, through causal reasoning the qualitative physics of the lower hoist have been implemented to provide a platform for postdiction. The mechanism for causal reasoning, WIF, is described in Section 4.

3.1. Qualitative Physics

Qualitative physics is the study of the behavior of the world in inexact terms. It has been suggested that humans perceive, understand, and generate expectations about physics of the world in an imprecise form. WIF is a hypothetical reasoning system that can be used to model causality. Causality is the study of how to represent what happens as a result of some action. Qualitative reasoning with physics about physical processes occurs so commonly in our understanding of the world that we are rarely aware it. Consider the humorous "Rube Goldberg" device displayed in Figure 3. The problem is clearly underconstrained, and yet we can easily anticipate the outcome.

Differential equations rarely, if ever, help us calculate the consequence of an action in the world. We evidently can apply fundamental, qualitative knowledge of physics even in highly novel situations. Deducing the consequences of actions on the world has a long tradition in artificial intelligence and has come to be known as the "frame problem"[15].

Hypothetical reasoning as embodied in WIF is expressive enough to solve the frame problem for HOIST. Given an initial world \( W_0 \) and a set of hypotheses \( X \) (i.e., \( X \) is a set of facts not necessarily consistent with \( W_0 \)), WIF generates a set of worlds \( \{W_1, W_2, \ldots, W_n\} \) that represent consistent worlds similar to \( W_0 \) but that include \( X \). Causal rules encode the interconnections between facts within a world. Hence, causal relationships take the form of rules (later called causal equations). In a world with the causal relationships of the Rube Goldberg device of Figure 3, a rule such as:

\[
(\text{on } *\text{something cot}) \rightarrow (\text{sags cot})
\]

would exist. Interpretation is as follows: for any rule in the world, if the precondition (the left hand portion) of the rule is true, then the consequence (the right hand portion) must also be true. Variables, denoted by the star prefix, can assume any value. So the above rule states that in any possible world where it is true that something is on the cot, it is also
true that the cot sags. For a more detailed explanation of WIF see chapter 4, "WIF: hypothetical reasoning."

3.2. Causal Reasoning

A repair expert may have some "rules of thumb" for isolating faults, but he does not follow these rules exclusively, as shallow reasoning expert systems would imply. The general hypothesis of model based expertise asserts that the repair expert has some understanding of the purpose of the machine he is working on, and thereby he knows the expected behavior of the machine. Consider a diagnostician who observes something not expected, would not he begin a process of deduction in order to explain the deviation? There are two consequences of such a deduction: either the diagnosticians expectations of the device are found incorrect and updated; as is likely with a new repairman, or postdiction isolates the faulty component. In either case causal reasoning is the tool: reasoning based on a causal chain of events.

3.3. Causal Influence Is Local

Fortunately, a complex causal chain of events can be simulated by a series of simple causal events placed end to end, if the "principle of locality" has been kept in mind during the modeling stage. The principle of locality[5,6] is a hypothesis which states that no single causal relation influences the behavior of a set of causal relations, except through influence of its neighbors.

In the Rube Goldberg device of Figure 3, it seems natural to say that the sleepy fellow lying in the cot would cause the cat to be put out. But, that situation should not be modeled as (sleepy fellow lying in cot) --> (cat is put out). Such a rule would violate the principle of locality in that the action of the fellow lying would dictate the performance of the entire Goldberg device. If such a rule were used, then it would be impossible to represent a scenario where the fellow had forgotten to reset the anvil. This objection may sound silly, but it precisely exemplifies the purpose of qualitative physics. That is, qualitative physics is useful because it provides a formalism that may represent in a concise way all the consequences of some action in an arbitrary world.

Causality should be modeled with local influence whenever possible. Just as an inventor might devise a machine by connecting old inventions together in a new order, the model constructor should make simple models and connect them together, thereby assuring locality of causal influence.
Somehow we can follow:
the sleepy fellow lies down on the cot, which causes
the cot to sag, which causes
the toothpaste to be squished out of its tube, which causes
the extra weight to move the lever, which causes
the support of the anvil to be knocked off balance, which causes
the anvil to fall, which causes
the string attached to the anvil to be pulled, which causes
* the window to open
* the cage lid to be removed
* the lever attached to the cage to be pulled, which causes
  kitty to be gingerly cat-apulted out the window for
  a feline night on the town

Figure 3: How to put out your cat at night
This does not argue for an increased granularity in modeling, but merely states that the level of granularity should be used consistently. That is, one should not change levels of abstraction while modeling local influence. This is not to be confused with the switching of levels of abstraction in order to obtain a new perspective on a problem[4].

3.4. The Level Of Abstraction

The atomic elements of the model represent components that are modeled in "black boxes." We know that a black box has internal components, but they are not represented in the model, either because

* Further detail in the model is inappropriate for diagnosing/fixing,
  the purpose of the model in the first place, or
* We do not have adequate time/memory or other computational
  resources to utilize greater detail, or
* We do not really know what goes on inside the "black box."

Hence, a model may have "uniform granularity," even though black boxes may differ in internal complexity - hypothetical or real.

Before beginning to create rules to model causal relationships, the level of abstraction (granularity, level of detail) of the representation must be carefully chosen. A high level of abstraction is computationally efficient, but less expressive in that some important functionality may not be represented. A low level of abstraction (e.g. colliding molecules in a hydraulic pipe) allows modeling a multitude of phenomena, but with a consequent increase in memory requirements and computational complexity. The world should be modeled only with the necessary level of detail for the problem at hand.

HOIST uses boolean representations to simplify the model whenever possible. The lower hoist is weakly instrumented. The only indications of internal states of the hoist are electrical on/off indicators. Hydraulic pressures cannot be measured because disassembly destroys the object of measurement. And the transfer functions for state change of pistons are not available. Thus highly detailed qualitative (i.e. such as quantizing pressure into 5+ regions) or quantitative models were inappropriate. Hydraulic pressures and voltages are represented by boolean variables (i.e., high pressure or voltage is represented by "on", low pressure or voltage by "off"). Mechanical linkages are quantized into discrete positions (e.g., a given piston may have three positions: "on", "off", and "center"). At the level of
abstraction chosen, a few components do not translate well into HOIST’s model. An orifice, whose purpose is to restrict the rate at which a hydraulic line may change value, is currently modeled no differently than a hydraulic pipe. The high level of abstraction chosen for HOIST is computationally efficient and is sufficient for representing most components of the Mark 45 lower hoist.

3.5. Unified Postdiction and Simulation

All too often, research projects do exactly as they were designed to do, and no more. This criticism is especially sharp in artificial intelligence, where a "tweaked" system may come off seeming more artificial than intelligent. For instance, if a project for learning by example can not learn from its own behavior, one might question the legitimacy of the approach.

Diagnostic systems based on causality to date, tend to be constructed with a specific research goal in mind; either they address single fault diagnosis or multiple fault diagnosis. DeKleer, for example, has produced both systems for diagnosing single[5] and systems for diagnosing multiple faults[6]. It would seem that the single fault assumption was so ingrained in the early stages of development of the first systems, that the work had to be scraped when considering multiple faults.

To be fair, accurate and speedy multiple fault diagnosis involves a different set of issues than single fault diagnosis. But the two problems are highly related; they both need a model of the device, so why can the same model not be used for both?

In HOIST, the single fault assumption is simply a special case of multiple fault diagnosis. Quite often heuristics are used to minimize the complexity of diagnosing multiple faults. The single fault assumption is one such heuristic, so HOIST diagnoses single and multiple faults.

Further, the approach to postdiction used in HOIST allows other forms of model based reasoning. The HOIST program is simply a model; with a diagnostic question, WIF generates a diagnostic answer. (e.g. How is what I have seen possible, given that components can fail?) However, the HOIST model and a simulation question will generate a simulation answer as well (e.g. If these are the inputs and no components can fail, then what should I expect?). Since WIF is a generic mechanism for maintaining consistency, the same model can be used for simulation and postdiction of single and multiple faults.
4. WIF: Hypothetical Reasoning

Hypothetical reasoning is the process of determining the implications of assumptions. A set of assumptions may or may not be valid, but the acceptance of assumptions as valid can imply the validity or invalidity of other knowledge. WIF is an implementation of hypothetical reasoning.

The example explored in depth in *A Coherence Logic For Counterfactual Reasoning* [18] is as follows. A world consists of a robot and a box in room 1, and the robot is holding the box. What if we were to assume that the robot were in room 2. The assumption implies two alternate sets of deductions: either the box is in room 2, or the robot is no longer holding the box. The assumption leads irrevocably to one of two conclusions in that to alter the world to include the assumption and not one of the two conclusions is to have an inconsistent set of facts. Hypothetical reasoning is a mechanism to restore consistency when a contradictory fact is injected into a previously consistent set of facts.

4.1. Uses

Given that WIF is a mechanism for restoring consistency, it is a tool of general application. It contains no information specific to any one application area. As such, it requires that the user specify the laws of his/her problem domain. The following are a few domains to which WIF has been used thus far.

WIF has been used to model mechanics for fault diagnosis. In this thesis, we have used WIF to represent the causal connections in an electronic-hydraulic-mechanical machine, the Mark 45 hoist. We encoded machine failure conditions as hypotheses and WIF generated possible internal machine states that would exhibit that failure. Further, since detailed internal states were provided by WIF, a graphic display of machine internal conditions was possible.

WIF has been used to model drug interaction in the body. In her thesis, Scheckler[19] modelled the four stages in the heart as a biochemical machine. By hypothesizing the presence of drugs, Scheckler was able to predict, to some extent, its effect on the blood pressure.

WIF has been used to discern incomplete knowledge in a multi-agent robot domain. In his thesis, Graham[11] represented the physics of objects existing in a multi-room
environment. Questions such as, "if I left the broom in this room, and it is not here now, how is this possible and where could it be? were answered by WIF.

WIF is being used to implement hypothetical time reasoning for planning. In his forthcoming thesis, Pendergraft[16] is using WIF to generate possible plans to accomplish some goals. This requires the "physics" of time be represented so that the current state of the world can be assumed at $t_0$ and the desired goal at $t_n$. WIF then computes all possible sets of $t_1$ through $t_{n-1}$ thereby generating all possible ways in which the goal can be accomplished.

Hypothetical reasoning provides a mechanism for solving the AI "frame problem"[12]. The AI frame problem, simply stated, is the question, "How can I figure out the resultant effects of an action?" Or more properly, "How can I represent the world such that I can determine the effects of any arbitrary action?" This is identical to the question, "If I were to assume that some arbitrary action occurred, what would that imply about the rest of the world?", which is a hypothetical question. So, if hypothetical reasoning can answer hypothetical questions, then hypothetical reasoning can solve the AI frame problem.

4.2. Problem formulation

In order to use WIF, a problem must be specified such as, how is X "possible" if Y is "normal". The terms "possible" and "normal" are used imprecisely; the following are the formulations of the examples mentioned above. How is [input condition is on and output condition is off] "possible" when the hoist is "normally" [... statement of normal internal conditions ...]? How is [heart continued function with a drug] "possible" given the heart "normally" [... statement of normal heart conditions and normal drugs...]? How is it "possible" that [garbage is on the floor] when [I cleaned it up this morning]? How could I get to there if I am here now?

4.3. A Program

To create a system that will answer hypothetical questions, a network of facts must be constructed to represent the nature of the object of a hypothetical question. In the robot world example briefly mentioned earlier, all of the rules that define what it means to be an object, and the properties of "holding a box" must be defined. Rules such as: Any object can only be in one place at a time, room 1 is different than room 2, and anything that holds an object must be in the same room as that object. The rules create relationships between individual facts so that a hypothetical question can be answered.
The problem of constructing a system that will answer hypothetical questions for some domain becomes a problem of how to represent the nature of the domain. This problem is unavoidable and sometimes quite difficult. The number of facts and rules we, as humans, apply to the robot world domain is very large and usually we are not aware that we are using them. Facts such as, each object takes up unique space unless it is a container, it displaces air, it has a color, it has a texture, weight, air resistance, temperature, etc. and any number of rules relating these facts. Fortunately not all of these properties are necessary for answering some questions. The facts and relating rules that describe the nature of the domain constitutes a program in WIF.

A world consists of facts and rules defined on those facts.

4.3.1. Facts

A fact is a piece of information of the form:

```
(<attribute_name> <entry>*)  <modal_category>
```

where

- `<attribute_name>` is an arbitrary token,
- `<entry>` is either a token, or a list of entries, and
- `<modal_category>` is an integer indicating believability.

Modal categories are a convenient tool for indicating relative believability among facts. Facts that are always true should have modal category 0. Facts with modal category n are believed more strongly than facts of modal category greater than n. For example, a world with facts:

```
(IS_A blue_container box) 0
(IS_IN blue_container the_Whitehouse) 1
(IS_IN President_Reagan the_Whitehouse) 2
```

would indicate that it is always true that the blue_container is a box. Further, it would indicate [the blue_container is in the Whitehouse] is more strongly believed than [President_Reagan is in the Whitehouse]. Hypothetical assumptions are automatically assigned modal category -1, in that they are believed above all facts.

4.3.2. Rules

Rules are of the form:

```
(<varying_fact>+  <rule_delimiter>  <varying_fact>+ )
```
where $\langle\text{rule\_delimiter}\rangle$ is defined as either: $\rightarrow$ indicating a normal rule
or $\rightarrow>$ indicating a rule without modus tollens

and $\langle\text{varying\_fact}\rangle$ is defined as either:

$\langle\text{attribute\_name}\rangle\langle\text{varying\_entry}\rangle*$

$\langle\text{attribute\_name}\rangle\langle\text{varying\_entry}\rangle*$ $\langle\text{modal\_category}\rangle$

where $\langle\text{attribute\_name}\rangle$ is an arbitrary token,
$\langle\text{varying\_entry}\rangle$ is either a token,
a variable, or
a list of varying\_entries,

and $\langle\text{modal\_category}\rangle$ is the local modal category of the $\langle\text{varying\_fact}\rangle$

Variables are bound local to a single rule.

Rules are a way of specifying the relationships between facts. The semantics of the rule, $(\text{lhs} \rightarrow \text{rhs})$, where lhs and rhs are one or more $\langle\text{varying\_fact}\rangle$s is: for all variables found in lhs, there exists variables found uniquely in rhs such that, if lhs is true, then rhs is true.

The following is a commonly used rule:

$(\text{INROOM }*\text{person }*\text{place1})$
$(\text{INROOM }*\text{person }*\text{place2})$

$\rightarrow$

$(= *\text{place1 }*\text{place2})$

It states that for all persons, for all place1s, and for all place2s, if the person is in place1 and the person is in place2, then place1 is equal to place2. In other words, no person can be in more than one place at a time. For examples with variables uniquely found in the right hand side of the rule, see the section below entitled, "Existential Quantification."

Some have suggested that rules should have a modal category, just as facts. There is no explicit modal category for rules, thus rules are considered at modal category 0 (i.e. always believed). However, a simple label for each rule will effectively give it a modal category. For example, if a user wanted the rule: $( (A) \rightarrow (B) )$ to have modal category 3, then the fact: $(\text{rule1})$ with modal category 3 and the new rule: $( (\text{rule1}) (A) \rightarrow (B) )$, where $(\text{rule1})$ is found in no other rule, will effectively implement the user's desire. This works because, as long as $(\text{rule1})$ is true, the rule will be in effect; once $(\text{rule1})$ is false, the relationship between $(A)$ and $(B)$ will not be regulated. Further, $(\text{rule1})$ has a modal
category of 3; so (rule1) can be considered a label for the rule, and as long as the label is true, the rule is used to maintain consistency.

4.3.3. Truth in WIF

The user sets up $W_0$, the initial set of true facts and rules that govern those facts. WIF assumes that the user constructed $W_0$ consistently under propositional logic. That is, WIF assumes that there is no fact $F$ and $\neg F$ both true in $W_0$, and for all rules $R$ in $W_0$ if the antecedent of $R$ is true in $W_0$, then the consequence of $R$ is also true in $W_0$. The user specifies what is true in $W_0$ by declaration. If the user declared $F$, then $F$ is true in $W_0$ and $\neg F$ is false in $W_0$. If the user declared $\neg F$, then $\neg F$ is considered true, and $F$ is considered False in $W_0$. If the user declares neither $F$ nor $\neg F$ then $F$ is considered False in $W_0$ and $\neg F$ is considered true.

When the user makes a declaration, he/she must associate with the fact $F$ a modal category $i$ [described briefly in section 4.3.1]. In such a case, modal category $i$ asserts the relative believability of both $F$ and $\neg F$. If the user declared facts of various modal categories from 1 to $n$, then all assumed true facts of the form $\neg F$ and assumed false facts of the form $F$ have modal category $n+1$.

Thus, WIF uses a modified closed world assumption. Since facts in WIF must have a modal category associated with them, WIF assigns a modal category $n+1$ to these facts, where $n$ is the highest modal category explicitly mentioned by the user. In other words, facts that are assumed false are believed false with less conviction than anything explicitly declared.

4.4. A Hypothetical Reasoning Example

Consider a rule:

\[
(\text{enrolled} *\text{person} *\text{course}) \\
(\text{attends\_class} *\text{person} *\text{course}) \\
(\text{studies} *\text{person} *\text{course}) \\
\rightarrow \\
(\text{completes\_course} *\text{person} *\text{course}) \\
(\text{obtains\_an\_A} *\text{person} *\text{course})
\]

Now also consider facts:

(\text{enrolled jack history}) 1
This rule means that when (enrolled *person *course), (attends_class *person *course) and (studies *person *course) match formulas in the knowledge base [that is, whenever someone enrolls in a course, attends class and studies], (completes_course *person *course) and (obtains_an_A *person *course) must also match formulas in the knowledge base [That person completes the course and obtains an A]. This small program will allow us to answer hypothetical questions about grades:

Question 1: What if (enrolled jill history)
Answer 1: Then the following is a consistent alteration
(enrolled jill history)

Question 2: What if (enrolled jill history) and (attends_class jill history) and (studies jill history)
Answer 2: Then the following is a consistent alteration
(enrolled jill history) and (attends_class jill history) and (studies jill history) and (completes_course jill history) and (obtains_an_A jill history)

Question 3: What if (not (obtains_an_A jack history))
Answer 3: then the following is a consistent alteration
   (not (obtains_an_A jack history)) and
   (or
      (not (enrolled jack history))
      (not (attends_class jack history))
      (not (studies jack history)) )

Question 4: What if (studies jack math)
Answer 4: then the following is a consistent alteration
   (studies jack math) and
   (or
      (obtains_an_A jack math)
      (not (enrolled jack math))
      (not (attends_class jack math)) )

The rule is preserved at all cost. In question 1, the rule did not apply and so no additional inferences occur. In question 2, the entire left hand side of the rule is assumed true. This implies that the right hand side must also be true. In question 3, the right hand side is assumed false. This implies that at least one of the left hand side facts are false. Question 4 assumes one of the premises true, thereby completing the truth of the left hand side. In order to restore consistency, either the conclusion must be true or at least one of the other premises must be false.

In the example above, the world is so simple that execution stops after considering only one rule. In more interesting worlds, not only must the assumptions be checked out in all rules, but all inferred facts must also be checked against the rules.

4.5. WIF, More Formally

Hypothetical reasoning is unlike first order logic since it must successfully reconcile contradictory statements. As in Rescher's theory, formulas are assigned an ordering or "modal category" perhaps best thought of as indicating relative believability. The modal category indicates what formulas will be cast out first when a counterfactual (hypothesis, presumed to contradict currently believed facts) is introduced.

A rule in the counterfactual logic can be represented as:

   [Rule 1] P, Q, R --> S, T

This rule means that when P, Q and R match formulas in the knowledge base (that is, when P, Q, and R are true in a world), S and T must also match formulas in the knowledge base
(S, T must be true in that world). If either S or T is not true, then consistency must be restored if possible.

A world is made up of a set of consistent facts and a set of rules defined on the facts. If a counterfactual is introduced, then the rules themselves will suggest possible worlds (i.e. Rescher's PMMC sets without a preference) where the counterfactual would be an element of the set of consistent facts. Counterfactual reasoning uses restoration to generate all possible worlds similar to the initial world but which also contain the counterfactuals.

**RESTORATION ALGORITHM**

FUNCTION Restore( Cfs_to_process, Modal_Cat, Known_cfs, World, RETURNED:answer ):BOOLEAN
BEGIN
  [Return FALSE if it is not possible to restore consistency]
  answer.type = AND
  answer.value = {}  
  FOR every clause C in Cfs_to_process
    [For every counterfactual, AND results]
    BEGIN
      IF C is in Known_cfs THEN NEXT C  [If already a counterfactual, skip to next]
      IF ~C is in Known_cfs THEN RETURN( FALSE )  [Counterfactual C and ~C exist, Fail]
      IF C is not in WORLD AND
         ( Get_Modal_Cat( C ) <= Modal_Cat ) THEN RETURN( FALSE )  [If C is not true in world, and C is believed, Fail]
      Known_cfs = { C } + Known_cfs
      anded_ans.type = AND
      anded_ans.value = {}
      FOR every rule R in World
        [For every rule, AND results]
        BEGIN
          IF C is in R's preconditions [left hand side] THEN
            BEGIN
              IF ( Consistent(R's preconditions, World, Modal_Cat)  
                   AND NOT(Consistent(R's consequences, World, Modal_Cat)) ) THEN
                BEGIN  [For every consistency restoration, OR results]
                  ored_ans.type = OR
                  ored_ans.value = {}
                  IF Restore( R's consequences, Modal_Cat, Known_cfs, World, ans1 ) THEN
                    ored_ans.value = ored_ans.value + {ans1}  [this implements Modus Ponens]
                  FOR every P in R's preconditions
                    IF Restore( (~P), Modal_Cat, Known_cfs, World ) THEN
                      ored_ans.value = ored_ans.value + {ans1}  [this implements Modus Tollens]
                    IF ored_ans.value == {} THEN RETURN( FALSE )
                  anded_ans.value = anded_ans.value + {ored_ans}
                END
            END
          answer.value = answer.value + {anded_ans}
        END
      RETURN( TRUE )
    END
  FOR every clause C in Set

FUNCTION Consistent( Set, Modal_Cat, Known_cfs, World ):BOOLEAN
BEGIN
  [Return TRUE if Set is consistent with cfs and World at given Modal_Cat]
  FOR every clause C in Set
BEGIN
IF C is in Known_cfs THEN NEXT C [If C is in cfs, then C is consistent]
IF ~C is in Known_cfs THEN RETURN( FALSE ) [If ~C is in cfs, then C is not consistent]
IF C is in World THEN NEXT C [If C is in initial world, C is consistent]
IF ~C is in World THEN RETURN( FALSE ) [If ~C is in initial world, C is not consistent]
IF C is not of form ~X for some X THEN [If C is not in World by default, C is not consistent]
RETURN( FALSE )
END
RETURN( TRUE ) [All are consistent, so set is consistent]
END

FUNCTION Gct_Modal_Cat( C )INTEGER [user defined routine that assigns modal categories to clauses]
BEGIN [= and <> are shown here as an example]
IF C is an = expression THEN RETURN( 0 )
IF C is a <> expression THEN RETURN( 0 )
END

Consider a world W₀ containing only P, R, all other assumed negated facts and
Rule 1 involving P through T above. Now, if the hypothetical Q is introduced (Q is a
hypothetical since ~Q is assumed to be true in W₀ ), Rule 1 in W₀ (with counterfactual Q) has all its preconditions met in W₀ (that is: P, Q and R are true). S and T, however, are not
true in W₀. Rule 1 itself suggests that a possible world that would contain Q would be one
similar to W₀ except that S and T must also be true in that world (i.e., Modus Ponens using
Rule 1). Two other possible worlds where Q could exist are: one with ~P and another
with ~R, because if the precondition of the rule is not true in a world, then the consequence
need not be true in that world (i.e., Modus Tollens using Rule 1). In more complicated
examples these suggested alterations to W₀ must recursively invoke WIF with themselves
as counterfactuals, because the "possible world" must be consistent.

In the restoration algorithm above, a possible world is found when the algorithm
terminates successfully. An implementation that explicitly returned a set of worlds in
disjunctive normal form would be unwieldy. In WIF, the tree of differences from the
original world is returned. Each invocation of the restoration algorithm generates a node in
the tree that consists of the newly introduced counterfactuals. If that invocation causes
recursion, then that node has a subtree associated with it. The method of "difference trees"
avoids having to reconstitute the entire knowledge base. Each traversal from branch to root
of the tree collects a set of alterations of the original world, which define a new world
where the counterfactual is consistent (see Figure 2). Given the "difference tree"
representation, the set of all solutions can be placed in disjunctive normal form using a
utility outside of WIF; as would be expected the output in DNF is quite large (e.g. pages)
even for moderate size programs in WIF. Thus the "difference tree" representation
elegantly define the set of all "possible world" solutions where the counterfactual is consistent.

4.6. WIF, In Summary

WIF provides a mechanism for hypothetical reasoning by maintaining consistency in a database. Rules place constraints between facts and modal categories establish relative believability of facts. WIF resolves inconsistent groupings of facts, as indicated by the rules, by inserting or removing facts whose modal categories are beyond the threshold of belief.

In HOIST, rules are used to define the relationships between input and output variables of components. These components are modeled separately and connected together (i.e. the output of one component is connected to the input of another). Thus a model of the entire lower hoist was constructed. This model was then used for both simulation and postdiction. In the case of simulation, the modal threshold is set high enough so that subcomponents are not allowed to fail, and hypothesized inputs define a unique set of internal states consistent with those inputs (i.e. the internal states found when the machine operates as designed). When the model is used for postdiction, the modal category is lowered so that a hypothesized set of inputs and outputs can be explained by a possible world where a malfunction of one or more subcomponents has occurred. WIF generates all such possible worlds, thereby insuring that the fault in the real hoist is contained in one of these "possible" worlds (i.e. assuming that the fault is not beyond the assumptions of the model, such as a thermal coupling fault in the electrical system).
5. Modeling The Hoist

This chapter will introduce the reader to a small part of the Mark 45 lower hoist. A simple latch piston will be used to exemplify the process of creating a qualitative model for a physical device. Essential to the construction of a model of a complex mechanism is causal reasoning's "principle of locality" (see Causal Influence Is Local, in Section 3). With this assumption in mind, a "complex model" can be constructed by connecting adjacent "simple model" subcomponents.

The lower hoist is part of a naval cannon; it is the transfer mechanism from the ammunition storage room to the gun's ready magazine. It has approximately one hundred and fifty components, most of which are pipes, but it also contains two solenoids, seven pistons, three latches, four state-detecting switches, a linkage, a chain, a hydraulic rack and a clutch mechanism.

In HOIST, each electrical, hydraulic or mechanical part's function is modeled as a set of causal equations. Any component may have direct influence only over neighboring components. No one component may directly influence the behavior of the entire machine. In the example that follows, two components (a piston and a block) are presented to illustrate implementation of the principles of causal modeling and locality in HOIST.

In Figure 2, *c1, *c2, *i1, *i2, and *o are hydraulic pressure lines. *pistonf is the direction of force exerted by the piston as a result of the relative pressures of *c1, *c2 and a spring. *m is the mechanical position of the piston. Figure 4 shows a picture of piston UVK10 in three different mechanical positions: *m=off, *m=on, and *m=center.

The center, dark section of each drawing represents the piston itself, and the slashed area represents the piston housing. The piston has a spring on the right hand side. UVK10 is modeled by two different sets of causal relationships. The first set of causal equations specifies that *c1, *c2 and the spring determine the direction of force of the piston (i.e. the direction the piston would travel if uninhibited).
The second set of causal equations dictates that the mechanical position \(*m\) of the piston specifies whether \(*i1\) or \(*i2\) is connected to the output \(*o\).

Note that force along the piston \(*pistonf\) and the mechanical position of the piston \(*m\) are variables that may assume values and are modeled no differently than hydraulic pressure. \(*c1\) and \(*c2\) of equation set 1 have no direct influence over \(*m\) of equation set 2 even though all three variables are part of the same piston. The only influence that \(*c1\) and \(*c2\) may have over \(*m\) is that they specify \(*pistonf\), and \(*pistonf\) is connected to some neighboring device which is eventually connected back to \(*m\). This sequence of interconnections is referred to as a "logical pipe," to be described later.

The second component to be modeled is called a block (see Figure 5). This block is a simplification of another piston in the lower hoist. The block can move vertically, and can be found only in one of two positions. When the block is in position pos1, the slot in
the block is aligned with the latch. When the block is in position pos2, the slot is not
aligned with the latch. *f_latch is the direction of force the latch is exerting. *latch is the
mechanical position of the latch.

<table>
<thead>
<tr>
<th>Equation Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*f_latch = off) -&gt; (*latch = off)</td>
</tr>
<tr>
<td>(*f_latch = on) and (*block = pos1) -&gt; (*latch = on)</td>
</tr>
<tr>
<td>(*f_latch = on) and (*block = pos2) -&gt; (*latch = center)</td>
</tr>
</tbody>
</table>

This model represents part of the behavior of the latch tongue in a simple
mechanical latch. The block can move up or down. (Actually the block is prevented from
moving up when *latch = on, but that is a different set of causal equations not shown
here.) If the block's slot is aligned with the latch (i.e., *block = pos1) then the tongue may
move as force *f_latch dictates. But if *block is in position pos2 while *f_latch is on, the
most *latch can do is rest against the block (i.e. *latch = center).

Now connect these three sets of causal equations. Place a connection between *f of
equation 1 and *f_latch of equation 3. Also place a connection between *latch of equation
3 and *m of equation 2. The combined device created by this union is pictured in Figure 6.
Figure 4: Hydraulic Schematic Of Piston UVK10

TYPE: PISTON4

INSTANCE: UVK10
No pipes are depicted connecting *f and *f_latch or *m and *latch. These variables are not hydraulic pressures; they are forces and mechanical position indicators and are not captured in a picture very well. In the case of *f and *f_latch, the direction of force of the piston UVK10, *f, causes a similar direction of force on the latch tongue *f_latch. This causal connection is similar to the properties of hydraulic pipes. The value at one end of the connection (be it hydraulic pressure, direction of force, or mechanical position) must be the same as the value at the other end. This simple form of causal equation is labeled a "pipe". Hydraulic connections of this form are called "physical pipes". Force connections and mechanical linkages are called "logical pipes". The causal equations indicating the connections between variables for the two logical pipes in Figure 6 are expressed in equation set 4.

\[
\begin{align*}
\text{Equation Set 4} \\
\text{-} &\quad (*f = *f\_latch) \\
\text{-} &\quad (*m = *latch)
\end{align*}
\]

We can see in Figure 6 that *c1 and *c2 of equation 1 effectively drive a mechanical latch and that the state of the latch is indicated by *o. The dissection of the latch into subcomponents that interrelate only by affecting their neighbors' variables was useful for reducing the complexity of modeling. Further, any design change in UVK10 that has the same variables will not affect the block or any other device in HOIST.

Other mechanisms in the lower hoist were modeled in similar fashion. Hydraulic pressure was always one of two discrete values on or off, indicating high or low pressure. The mechanical position of pistons or the clutch were quantized into at most three positions. Electricity was either high or low voltage, and the force a piston exerts can be in one of two directions. Just as described for UVK10, the individual pieces of the lower hoist were modeled using sixty five WIF rules and connected together with pipes, and the result modeled the Mark 45 lower hoist.
A block which slides vertically and a latch

Figure 5: Hydraulic Schematic Of BLOCK
Figure 6: Combined Schematic Showing UVK10 And BLOCK Together
6. Postdiction

In building HOIST we wanted to compute what single and multiple component failures could explain the state of the lower hoist after malfunction. After a hoist malfunction, various internal states of the machine are known because of internal state detecting switches. So the question is asked, "How could it be that components X, Y, and Z are in states A, B, and C?" This is postdiction.

The unified theory of simulation and diagnosis used in this thesis is simple and intuitive:

* A properly functioning device has predictable behavior.
  (simulation of a single device)
* A set of properly functioning devices placed in combination has predictable behavior.
  (simulation of a set of devices)
* Malfunction is detected by deviation from that behavior.
* A device that malfunctions is composed of two sets:
  A) A finite set of properly functioning subdevices.
  B) A finite non-empty set of malfunctioning subdevices.
* Only elements of set B can explain deviated behavior.
* It requires less effort to isolate the subdevices in B, if one assumes that |B| = 1.

This is known as the "single fault assumption"

WIF by design generates all possible worlds where a solution exists. In the diagnostic domain, this consists of all possible worlds where the machine would exhibit the observed behavior. So, WIF generates all possible sets of malfunctioning subdevices B and the internal states of the machine when subdevices B malfunctioned. WIF automatically diagnoses all possible simultaneous faults in the machine for a set of symptoms.

The number of solutions when allowing multiple faults can typically be quite large. Take for example a widget that during some test phase exhibits ten outputs contrary to the widget's correct performance. It may be the case that a single component malfunction creates all of the observed incorrect outputs. If we allow multiple simultaneous faults, however, there may be ten faults each of which is immediately before one of the observed incorrect outputs. The number of combinations of faults for complex machinery is
tremendous and this combinatorial explosion is an unavoidable consequence of diagnostic systems allowing multiple faults.

Given the structure of the widget and the observed outputs, WIF generates all possible sets of faults and thus encounters the problem of combinatorial explosion. Combinatorial explosion leads to disastrous run times, but is unavoidable if one wishes to isolate multiple faults. In HOIST, an additional constraint was added to curb combinatorial explosion: the single fault constraint. In other words, we request that WIF generate all possible worlds explaining the observed phenomena in which at most one component malfunctions.

The single fault assumption is merely a solution constraint. The assumption of up to two faults is also a solution constraint. HOIST allows the user to specify any value of \( n \), where \( n \) specifies the maximum number of faults allowable in the solution. By invoking HOIST multiple times, a user or a program can first search for a single fault, then two faults, three faults, etc. until the fault(s) are isolated. Such an invocation strategy solves the simple problems first and addresses the more general and more difficult problems only if needed. The user systematically relaxes constraints thereby incurring only as much search (and thus computation) as needed.

Controlling the number of possible faults allows one kind of constraint on the search space. Heuristic search can also be added by the user. Heuristics allow a search for a subset of the entire search space that still tends to yield solutions. Any hypothesis a user may have should be presented with the fault isolation request to WIF. The added constraint of the hypothesis will define a subset of the entire search space accordingly. If the user's hypothesis is correct, the solution will still be in the reduced search space and WIF will find it. WIF is not a heuristic for multiple fault isolation; it is a tool for implementing a heuristic for multiple fault isolation.

WIF requires no patch or reconstitution to diagnose multiple faults, indeed, WIF does this by default. In HOIST, we included the single fault assumption in order to prune the search space. WIF can diagnose multiple faults after the single fault assumption fails thereby achieving speedy results on simple problems without giving up the capability to solve the more difficult problems. The capability of isolating single and multiple faults is more than a "neat" feature of our counterfactual reasoning system: in science, the simplest explanation is usually perceived as the closest to the truth.

6.1. Postdiction On The Lower Hoist
The example of UVK10 forming a latch will be used to exemplify HOIST postdiction. For the purpose of this section assume the values of *c1, *c2, and *o are directly verifiable by having someone walk up to the real machine and read a dial. Assume further that all other internal states cannot be immediately checked. Unverifiable internal states are the norm in the lower hoist, since direct observation of parts usually requires a hydraulic shutdown that destroys the state to be observed.

Assume *c1 and *c2 are known to be both off, and *o is known to be on. Figure 4 pictures the latch with inputs *c1 and *c2 off when the latch is working as designed. Observe that the predicted state of *o is off. This contradicts the known observation that *o is on. WIF is invoked with a statement that corresponds to:

Hypothesize that (*c1 is off), (*c2 is off), (*o is on) and a single fault caused the *o to be on.

What does this hypothesis imply about the structure of the device? A partial trace of the execution of WIF will show how this hypothesis is resolved.

*c1 and *c2 are off as in the model. [*c1 and *c2 will not be allowed to change value, since they are part of the hypothesis.] *o is on, which does not correspond with the model. If *o is on, then *o is not off [because of a rule of mutual exclusion]. In causal equation set 2, a relationship is asserted between *i1 and *o when UVK10 is in mechanical position on [*m is on]. So if *o is not off, then one of three possibilities exists, either:

A: UVK10 is not functioning as designed, or
B: *i1 is not off, or
C: *m is not in position on

Since the full trace of these options can be tedious, we shall concentrate here on the most interesting and instructive alternative C. If *m is not on, then two possibilities exist [because *m must be one of three values]:

C1: *m is center, or
C2: *m is off

Possibility C1: if *m is center, then *latch must be center. The only way *latch could be center is if *block is in pos2 and *f_latch is on. *f_latch is confirmed on, however *block was thought to be in pos1, so remove the (*block is pos1) fact. Returning to the matter of *m being center, it follows that *i1 must not be off. So, *i1 must be on. The following alterations to the original world would repair consistency:

((*o is on), (*o is not off), (*m is not on), (*m is center),
(*latch is center), (*latch is not on), (*block is pos2).
(*block is not pos1), (*i1 is not off), (*i1 is on)).

Possibility C2: if *m is off, then *latch must be off. If *latch is off, then one of the
two possibilities exist, either:

C2A: BLOCK is not acting as designed, or
C2B: *f_latch is off

Possibility C2A: if the block is not acting as designed, then the performance of the
block is not predictable. Thus *f_latch being on and *latch being off is acceptable. The
following alterations would restore consistency in the original world:

((*o is on), (*o is not off), (*m is not on), (*m is off),
(*latch is off), (*latch is not on), (BLOCK malfunction)).

Possibility C2B: if *f_latch is off, then *f must be off. *f can be off in one of two
possible ways:

C2B1: *c1 is off and *c2 is on, or
C2B2: UVK10 is not performing as designed

Possibility C2B1: *c1 is off is in accord with the hypothesis. However, *c2 is on
is inconsistent with the hypothesis that *c2 is off. Possibility C2B1 cannot restore
consistency to the original world.

Possibility C2B2 is similar to case C2A above and will not be pursued further here.

WIF generates solutions in a depth-first fashion and returns the solutions as a tree
of alterations. Any traversal from root to leaf represents a single set of alterations that
restores consistency to the original world. The full answer to the original query is given in
Figure 7.

WIF is implemented on a 1000 LIPS Prolog interpreter for the Vax 11/780. Fault
Diagnosis required 20 cpu minutes for HOIST under these conditions.
Figure 7: Full answer to the query:
What could explain *c1=off, *c2=off and *o=on
where at most one component may malfunction.

(and (*o is on)
    (*o is not off)
    (or (UVK10 malfunction)
        (and (*i1 is not off)
            (*i1 is on))
        (and (*m is not on)
            (or (and (*m is center)
                (*latch is center)
                (*latch is not on)
                (*block is pos2)
                (*block is not pos1)
                (*i1 is not off)
                (*i1 is on))
            (and (*m is off)
                (*latch is off)
                (*latch is not on)
                (block malfunction))
        )
    )
)
7. Results, Comments And Conclusions

A full model of the lower hoist has been encoded in coherence rules and a running simulation was produced and delivered to the FMC Corporation in the Fall of 1986. The model was built to act as a diagnostic expert system, and the model has been tested successfully in several diagnostic situations.

At least initially, FMC has been using the HOIST project as a simulator. In order to address the problem of fault diagnosis, a model of the mechanics of the device's components was required. This model allowed WIF to deduce all possible device configurations that would explain device behavior (i.e., all possible worlds where the hypotheses exist.) If the only assumed states of the device are its inputs, then WIF will deduce all the internal states and the outputs of the machine. If the device is deterministic (as nearly every real world machine is), then with the inputs WIF will generate exactly one such solution. In essence this degenerative use of hypothetical reasoning has the effect of simulating the device as it functions properly.

Shortly after delivery, FMC developed a graphics front end to the HOIST causal expert system (see Figure 1). This was a natural step since HOIST deduces all internal states of the lower hoist for a possible diagnostic solution (i.e. all things true in the world), or for that matter, all internal states as the lower hoist functions properly. A graphics front end simply displays a unique icon for each possible state of each component, preferably one that looks like the component in that state, as in Steamer[13]. The obvious uses for such a system include training, interactive reference material, as well as fault diagnosis. HOIST is an excellent example of applied AI research. The HOIST causal reasoning expert system has successfully transitioned from an interesting theory to a useful tool for industry.

7.1. A Poor Use of HOIST Technology

A sister project to FMC's HOIST modelled a cigarette maker for the Phillip Morris Corporation. The cigarette maker was a highly instrumented mechanical device. Modelling the MAKER was fairly straight forward, and diagnosis was accomplished. However, little additional information was generated from MAKER that could not be observed immediately from the control panel. In short, the fact that the internals of the Mark 45 lower hoist were not immediately observable made HOIST a useful expert system for diagnosis. HOIST postulated the internal states of the lower hoist; this was a service that was unnecessary for the Phillip Morris cigarette maker.
It has been suggested that the increased data pool of the Phillip Morris cigarette maker could have been used to diagnose poor performance trends. While this is possible, the topography of device performance is less obvious than the topography of the functionality of the device. If one assumes topography of device performance is obtainable from the blue-prints of the mechanism (e.g. as is done with the topography of the functionality of the device), then the principle of locality no longer holds (i.e. component A's slight deviation in performance may not affect its connected component B, but may affect one of B's connected components C). If one assumes the principle of locality, then it is clear that the topography of device performance is not obtainable from device blue-prints. Research is needed in the causality of naive trends.

7.2. Machines Not Currently Modeled in HOIST

Most of the components of the lower hoist of the Mark 45 operate during a single "clock pulse." That is, a signal is sent to indicate that the hoist is to operate (e.g. raise or lower) and no new signals are accepted until it completes its operation. In HOIST, counterfactual reasoning is used to compute the results of some action one "clock pulse" later. That is, the HOIST at that moment (one "clock pulse" after the action signal) has transitioned all that it will transition; no amount of waiting will cause the HOIST to change state without new input.

Any device that is heavily dependent on time will require a more detailed representation of physics than that used in HOIST. In much the same way that the time it takes for a TTL circuit to change state is often ignored, the transition time of the machinery in HOIST is ignored. There is a large class of machines that need a more precise representation of time. For instance, assembly lines and pipeline machines process across multiple time units. But perhaps the most difficult are feedback mechanisms, in that they suggest a need for a continuous representation of time.

Time is not represented in WIF in any form. WIF is simply a mechanism for insuring consistency in an abstract domain. If that world happens to map to instantaneous qualitative physics, then WIF would generate all results of some assumption at a given time instant, as in HOIST. If that world were to map to qualitative event scheduling such as in temporal reasoning, WIF would generate all results of some assumption for a time interval (i.e., all possible time lines where the assumption is true). WIF can be used for time reasoning.

7.3. WIF As A Tool Of General Application
WIF has no control or structure specific to fault diagnosis; WIF is a language for hypothetical reasoning. Hypothetical reasoning is a general inference technique that can be applied to a multitude of problems. Scheckler[19] used WIF to model the heart's reaction to the introduction of drugs by asserting causal relationships between tissue response, physiology and clinical effects. Graham's multiple robot domain[11] used WIF to rectify truth when an individual robot discovers that it's knowledge base is incomplete or inaccurate (this is sometimes referred to as "belief revision" or "reason maintenance"). Diverse application suggests a generality of approach.

7.4. Conclusions

First generation expert systems have been based on superficial knowledge gleaned from human experts. A more fundamental approach would use functional models to generate advice automatically. A methodology is needed to specify the nature of individual components and their interrelationships. Causal reasoning is emerging as a technology for just this purpose: modeling qualitative physics. A causal reasoning expert system represents second generation technology because the computer creates its own advice.

Advantages of our approach over conventional expert systems stem from the nature of causal reasoning. Causal reasoning's principle of locality insures locality of modifications and the ability to model many different "complex" mechanisms uniformly by reducing them to interconnecting "simple" relationships. This in turn assures system maintainability and allows development without the expert usually found in expert system projects.

Building a causal model of the lower hoist required us to address qualitative physics. We have modeled the functionality of the mechanical, hydraulic and electrical systems in HOIST using a hypothetical reasoning language called WIF. One year after its genesis in the Spring of 1986, WIF has spawned work in a number of different domains: causal based fault diagnosis, robot world belief revision, and qualitative heart simulation. Hypothetical reasoning is a major facet of human intelligence and is not tied to any specific application area.
8. Literature Cited


Appendix I: WIF User Functions

Summary of useful items in the COUNTERFACTUAL reasoning engine

(counterfactual <list of counterfactuals> <belief> <answer>)

This is the basic form of invocation of the counterfactual engine. It supposes that the <list of counterfactuals> is true, and places all that is implied by it in <answer>. <belief> defines the limit of the search, if <belief> is n then all facts in the system with modal category of n or less will be believed (i.e. these facts will not be considered for alteration when trying to compute all implications of <list of counterfactuals>). Whenever the <list of counterfactuals> cannot be explained for the <belief> used, or if the <list of counterfactuals> is inconsistent amongst itself then counterfactual will return nil.

(cf <list of counterfactuals> <belief>)

This is a short hand version of counterfactual. It simply invokes counterfactual and pretty prints the answer.

(best_account <list of counterfactuals> <returned belief required> <answer>)

This is not an especially efficient version of the function counterfactual. It merely invokes the counterfactual engine with successive belief thresholds until it finds one which returns an answer. <returned belief required> is the belief threshold for which at least one answer was found. NOTE: <answer> is not necessarily a unique answer, merely the answer found at the greatest possible modal category (ie. belief level).

(choose_one <list of counterfactuals> <answer from cf engine>

This routine presumes that the user has already invoked either
counterfactual or best_account. Using the <answer from the
counterfactual engine> this routine chooses the "best" answer
and returns this as a simple list (i.e. <one answer in list form>).
The modal category required for the answer is also returned.
The definition of "best" in this case, means where there are
more than one choice (e.g. (OR <clause1> <clause2>)), the choice
which has the greatest possible modal category wins.

(flatten <answer from cf engine> <returned answer in DNF>)
This routine accepts an and/or tree such as the one generated
as a result of the cf engine and flattens it down into
Disjunctive Normal Form. As such, <returned answer in DNF>
is of depth at most 2. (i.e. an OR of ANDs of facts or
negated facts.)

(mk_facts <modal category> <fact1> <fact2> ... <factN>)
This routine asserts facts 1 through N in modal category
<modal category>.

(rem_facts <modal category> <fact1> <fact2> ... <factN>)
This routine removes the first fact found for each of <fact1>
through <factN>. If <modal category> is bound, then removal
will only occur from that modal category, otherwise removal
will occur from any modal category.

(del_facts <modal category> <fact1> <fact2> ... <factN>)
This routine is identical to rem_facts, with the exception
that all facts will be deleted that match any of <fact1>
through <factN>.

(update_facts <list of clauses>)
This routine attempts to update the knowledge base, presumably
after the counterfactual engine was invoked, and the user
approves of the result. If a clause is of the form (not *x)
it first tries to delete *x, and failing this, it asserts
(not *x). All other clauses are asserted (ie. mk_fact).
This routine attempts to place a clause to be asserted in
the same modal category as clauses with the same relation
name which already exist in the system.

(mk_rules <rule1> <rule2> ... <ruleN>)
This routine asserts counterfactual rules. Counterfactual
rules are of the form:
(<lhs1> <lhs2> ... <lhsP> -> <rhs1> <rhs2> ... <rhsQ>)
In this example there are P clauses on the left hand side
and Q clauses on the right hand side.

(showfact <fact>)
This routine will pretty print all facts which match <fact>

(showf <factname>)
This routine is a short hand version of showfact. <factname>
is merely the relationship name, and showf will pretty print
all facts which have that relationship name.

(showrule <lhs clause of a rule>)
This routine pretty prints all rules which have the clause
 lhs clause of a rule in their left hand side.

(showr <clause name of lhs clause of a rule>)
This routine is a short hand version of showrule.
clause name of lhs clause of a rule is merely the
relationship name, and showr will pretty print all rules
which have clauses on the left hand side which have that
relationship name.

TRACE Facilities
A set of flags which help the user trace the counterfactual
reasoning engine as it exhaustively attempts to find rules pertaining to the counterfactual clauses, and instantiate them. This, of course, could propagate counterfactual clauses back any number of levels.

TRACE HEADER

(<what> <level>/<clause>.<rule>,<instance>) <value>

<what> may be:

Q meaning Query a new level
R meaning Return from a level
QC meaning Query of one of the clauses of the cf
RC meaning Return from the Query about a clause
QR meaning Attempt to match a clause in a rule with the current QC. (ie. Query Rule)
RR meaning Return from considering the rule QR
QI meaning Show the new instantiation of the rule QR
    (ie. the rule with its variables bound)
RI meaning Return from the current instantiation.

<level> indicates to what level of depth the cf engine is working
<clause> is the clause being considered from <list of counterfactuals> from the invocation of the counterfactual engine.

<rule> is a number representing the current rule being considered. This number coincides with the rules as listed with the user facility showr.

<instance> is a number representing the number of different times the same <rule> has been bound.

<value> on a Query, <value> will be the level, clause, rule, or instance being considered at the time.

on a Return, it assumes the value of t or nil. Here t means that the level, clause, rule, or instance was found to be consistent with the world. nil implies that the object of the return was NOT consistent.
TRACE FLAGS

- cf_t is a flag which enables any tracing for the cf engine.
- cf_l is a flag which enables the printing of the list of clauses used for the invocation of the cf engine. (This flag is on by default)
- cf_c is a flag which enables the printing of the clause being considered.
- cf_r is a flag which enables the printing of the rule being considered.
- cf_i is a flag which enables the printing of the instance being considered. (This flag is on by default)
- cf_s is a flag which enables the printing of the list of instantiated clauses for counterfactuals which have variables in them.
- cf_p is a flag which specifies that all printing is to be pretty.
- cf_o is a flag which specifies that the leaf nodes of the overall output of the cf engine are to be printed as they are encountered. (These leaf nodes correspond to cf which do not cause further levels of cf reasoning). (This flag is on by default)
- cf_rv is a flag which specifies that if the cf_o flag is on that the output line should be placed in reverse video. Note- this only works with terminals which follow VT100 conventions.

(cf_trace_all) will turn on all counterfactual trace flags above.

- cf_n is a flag which enables the n-fault mode, where cf_number is n.
  - If the modal category is 1, cf_n is t, and cf_number is three;
  - Then at most three facts can be thrown out at modal category 2, and any number of things can be thrown out of modal categories above 2. This has been useful in the past to implement the single fault constraint of fault diagnosis.

User Definable Functions

The user has the capability to define his/her own functions. This is an escape to prolog, where the prolog call should succeed only if the user wants the WIF function to be true. If the WIF function was called with variables unbound, a successful return may bind those variables. All that is required is that the user asserts a prolog rule of the form:
Perhaps the simplest example is the system defined = and <>.

\[
(\text{fact } (= \; *x \; *x) \; 0))
\]

\[
(\text{fact } (<> \; *x \; *y) \; 0) \; :\; - \; (! \; (== \; *x \; *y) \; ))
\]

Often user functions are defined to be truly relational. For example addition might be implemented as follows:

\[
(\text{fact } (\text{plus} \; *x \; *y \; *2) \; 0) \; :\; - \; \text{(boundp} \; *x)
\]

\[
\text{(boundp} \; *y)
\]

\[
(= \; *z \; (+ \; *x \; *y))
\]

\[
(\text{cut})
\]

\[
(\text{fact } (\text{plus} \; *x \; *y \; *2) \; 0) \; :\; - \; \text{(boundp} \; *x)
\]

\[
\text{(boundp} \; *z)
\]

\[
(= \; *y \; (- \; *z \; *x))
\]

\[
(\text{cut})
\]

\[
(\text{fact } (\text{plus} \; *x \; *y \; *2) \; 0) \; :\; - \; \text{(boundp} \; *y)
\]

\[
\text{(boundp} \; *z)
\]

\[
(= \; *x \; (- \; *z \; *y))
\]

\[
(\text{cut})
\]

User functions were devised to allow the user to implement any form of constraint between variables. However, there is no limitation what user functions can do; they may have side effects that redefine modal categories, for example. It is advised that the user exercise caution in creating functions with side effects.

**Existential Quantification**

There is a weak form of existential quantification in WIF. Typically, the user wants to define the variables in the premise of a rule and the relationship between these variables in the conclusion. In such a rule, without existential quantification WIF rules try to throw out old information, but will not propagate positive information in the database. In a world where the light bulb is hypothesized not off, WIF will not be able to infer that the light bulb is on. In WIF existential quantification is found only in the conclusions of rules, and it comes in two flavors: finite and infinite.

**Finite Existential Quantification**

Perhaps the simplest way to clarify this point is by example. Consider the following rules:
Rule 1: \[ (\text{Pressure } \ast \text{place } \ast \text{value}1) ; \text{This rule states that the pressure} \]
\[ (\text{Pressure } \ast \text{place } \ast \text{value}2) \quad ; \text{at any place must be at most} \]
\[ -\ast \text{one value.} \]
\[ (= \ast \text{value}1 \ast \text{value}2) \]
\[ \text{Rule 2:} \quad [ (\text{Device pipe } \ast \text{endpoint}1 \ast \text{endpoint}2) \quad ; \text{This rule states that pipes} \]
\[ \text{always} \]
\[ (\text{Pressure } \ast \text{endpoint}1 \ast \text{stat}1) \quad ; \text{have the same pressure at either} \]
\[ (\text{Pressure } \ast \text{endpoint}2 \ast \text{stat}2) \quad ; \text{end.} \]
\[ -\ast \]
\[ (= \ast \text{stat}1 \ast \text{stat}2) \]

The facts: (Device pipe junction1 junction2) 1 \quad ; \text{pipe between junctions 1 and 2}  
(Pressure junction1 low) 3 \quad ; \text{Pressure at junction1 is low}  
(Pressure junction2 low) 3 \quad ; \text{Pressure at junction2 is low}  

And the Hypothesis: (Pressure junction1 high) \quad ; \text{Assume Pressure at junc.1 is high}  

If the modal threshold is set to 2, then the pipe is not allowed to malfunction (i.e. at modal category 1), and the meaning of equality is preserved (i.e. system sets = at modal category 0). Without some form of existential quantification, WIF would generate the following with the hypothesis above:

\(( (\text{Pressure junction1 high}) \quad 
\Rightarrow 
\quad (\text{and (not (Pressure junction1 low)}) \quad ; \text{Generated from rule 1} 
\quad (\text{not (Pressure junction2 low)}) )) \quad ; \text{Generated from rule 2} \)

As you can see, WIF does not make the deduction that since \((\text{not (Pressure junction low)})\), then \((\text{Pressure junction high})\). In order to allow this sort of deduction, the following rule and facts must be included:

Rule 3: \[ [ (\text{not (Pressure } \ast \text{place } \ast \text{value})) \quad ; \text{This rule states that for every} \]
\[ -\ast \text{pressure place, there is a} \]
\[ (\text{Pressurevalue } \ast \text{value}1) \quad ; \text{pressure value.} \]
\[ (\text{Pressure } \ast \text{place } \ast \text{value}1) \]

Facts: (Pressurevalue low) 1 \quad ; \text{The two pressure values defined} 
(Pressurevalue high) 1 \quad ; \text{are low and high}
This rule has a sort of procedural feel to it. It should be read as, if (not (Pressure someplace somevalue)) is ever a hypothesis, then that same place must have a pressure value. One might think that a rule of the form:

\[ \rightarrow (\text{Pressurevalue *value1}) \; (\text{Pressure *place *value1}) \]

would suffice. But, this rule states that there is a place that has a pressure value.

Furthermore, the rule:

\[ (\text{Pressureplace *place}) \rightarrow (\text{Pressurevalue *value1}) \; (\text{Pressure *place *value1}) \]

might logically represent every pressure place has a pressure value, but this rule will never work in WIF. WIF attempts to match counterfactuals to the premises [left hand side] to determine if a rule is applicable. As a result, this rule will never be checked, since no premise will become a counterfactual. A future version of WIF could allow these sorts of rules.

**Infinite Existential Quantification**

The rules 1-3 above exemplify finite existential quantification, in that the clauses in the conclusion can assume values from a finite set (i.e. in this case, *value1 can be either low or high). On occasion one needs infinite existential quantification. The following rules encode the gas laws and use infinite existential quantification.

**Rule 1:**

\[ (\text{Status P *Pval}) \; (\text{Status V *Vval}) \; (\text{Status N *Nval}) \; (\text{Status R *Rval}) \; (\text{Status T *Tval}) \rightarrow \; (\text{gaslaw *Pval *Vval *Nval *Rval *Tval}) \]

; gaslaw is a user defined

; fully relational function implementing PV=NRT

**Rule 2:**

\[ (\text{not (Status *what *value1)}) \rightarrow \; (\text{Status *what *value2}) \]

In order to understand how infinite quantification works, once again you must think procedurally. Suppose \( W_0 \) has values for \( P, V, N, R, \) and \( T \) all of which are 1 [Normally \( R \) is 6.023E-23, but 1 makes a simple example]. If the hypothesis (Status P 2) is introduced, then rule 1's premises are true, and conclusions are false. Each of the premises will be negated and assumed as a counterfactual in turn [ the results of each of these will be ored together]. Take (not (Status T 1)) for example. (not (Status T 1)) will match the
premise of rule 2. Rule 2 will succeed only if WIF can consistently find a value for *value2. First WIF looks for facts of the form (Status T *value2) for suggestions as to how to bind *value2 [see finite existential quantification above]. Since there are no facts of that form, WIF attempts to find rules that have a premise of the form (Status T *value2). Rule 1 has such a premise. Further, rule 1 binds *value2 to 2, since the user defined function gaslaw is truly relational (i.e. since only one variable is unbound, namely *Tval, then *Tval will assume the value of [*Pval times *Vval] divided by [*Nval time *Rval]). All applicable rules will generate candidates for *value2 in this fashion and if there is one such candidate that is consistent (such as 2), then rule 2 will be satisfied. Of course, if rule 2 is not satisfied, then the hypothesis (not (Status T 1)) will be inconsistent and failure will propagate back to the rule or user who hypothesized (not (Status T 1)).

The existential quantification is infinite in this example, since the variable can potentially be bound to any real number. Whether finite or infinite, existential quantification will generate candidates for a variable. WIF will try each candidate in turn and return an or-tree of these possible instantiations of the variable. The existential quantification is weak, however, since a fact or rule must generate a consistent candidate for the quantified variable (i.e. no candidate, then existential quantification is false).
Appendix II: HOIST Program

/*****************************/
* filename: lhoist.hc
* lhoist and lhmore define all of the rules and facts for WIF
/*****************************/

(print "loading cf engine")
(load "[-DEMO]cf.hc")
(print ")")
(print "Setting up lower hoist coherence rules")
(print ")")
; load in the routines that preprocess mk_device into several mk_fact calls
(load lhsupport)
; load in the routines that converse with the graphics workstation
(load netserver)
(mk_rules
 ;
 ; No Status point can have two values
 ;
 ; ( (STATUS *what *name *location *val1)
 (STATUS *what *name *location *val2)
 ->
 ( = *val1 *val2 )
 ;
 ; All Status points must have some sort of value,
 ; Thus, if a value is to be thrown away, another
 ; must be selected.
 ;
 ; ( (not (STATUS *what *name *where *val))
 ->
 (STATUS *what *name *where *otherval) )
 ;; (pipe)
The name of the pipe automatically specifies its end points.

Whatever is at one end of a pipe is at both ends of the pipe.

```
(DEVICE external_pipe (*loc1 *loc2) (*a *b))
(STATUS *x *a *loc1 *val1)
(STATUS *y *b *loc2 *val2)
->
   (= *val1 *val2) )
```

```
(DEVICE pipe *location (*a *b))
(STATUS *x *a *location *val1)
(STATUS *y *b *location *val2)
->
   (= *val1 *val2) )
```

```
(DEVICE orifice)
```

```
example -- lower hoist UOK3
```

```
h1 -- hydraulic line on one end
h2 -- hydraulic line on other end
```

```
(DEVICE orifice *location *x)
(STATUS orifice (*x h1) *location *h1)
(STATUS orifice (*x h2) *location *h2)
->
   (= *h1 *h2) )
```

```
(DRIVE no_default *ch1 *ch2 *m *of)
```

```
example -- lower hoist UVK4
```

```
c1 -- end of piston which if powered will drive piston towards gravity
```
c2 -- end of piston which if powered will drive piston away from gravity.
m -- the previous state of the piston (on = as c1 would drive it)
f -- the force along the piston (on = force in direction c1 would drive it)

((DEVICE *what *location *x)
 (DRIVE no_default *what)
 (STATUS *what (*x f) *location center)
 ->
 (= 1 2))
 ((DEVICE *what *location *x)
 (DRIVE no_default *what)
 (STATUS *what (*x c1) *location off)
 (STATUS *what (*x c2) *location on)
 (STATUS *what (*x f) *location *f)
 ->
 (= *f off))
 ((DEVICE *what *location *x)
 (DRIVE no_default *what)
 (STATUS *what (*x c1) *location on)
 (STATUS *what (*x c2) *location off)
 (STATUS *what (*x f) *location *f)
 ->
 (= *f on))
 ((DEVICE *what *location *x)
 (DRIVE no_default *what)
 (STATUS *what (*x c1) *location off)
 (STATUS *what (*x c2) *location off)
 (STATUS *what (*x p) *location *p)
 (STATUS *what (*x f) *location *f)
 ->
 (= *f *p))
 ((DEVICE *what *location *x)

(DRIVE no_default *what)
(STATUS *what (*x c1) *location on)
(STATUS *what (*x c2) *location on)
(STATUS *what (*x p) *location *p)
(STATUS *what (*x f) *location *f)
->
(= *f *p) )

; ; (DRIVE one_side_default *name)
;
; example -- lower hoist UVK5
;
; c1 -- end of piston which has spring in it(reinforces default)
; c2 -- end of piston which tends to push away from default
; f -- the force along the piston (on = force in default direction)
;
( (DEVICE *what *location *x)
(DRIVE one_side_default *what)
(STATUS *what (*x f) *location center)
->
(= 1 2 )
)
((DEVICE *what *location *x)
(DRIVE one_side_default *what)
(STATUS *what (*x c2) *location off)
(STATUS *what (*x f) *location *f)
->
(= *f on )
)((DEVICE *what *location *x)
(DRIVE one_side_default *what)
(STATUS *what (*x c1) *location off)
(STATUS *what (*x c2) *location on)
(STATUS *what (*x f) *location *f)
->
(= *f off) )

(DEVICE *what *location *x)
(DRIVE one_side_default *what)
STATUS *what (*x c1) *location on)
STATUS *what (*x f) *location *f)
->
(= *f on) )

; (DRIVE center_default *name)
;
; example -- lower hoist UVK3
;
; c1 -- end of piston which pushes piston to on position
; (usually pictured right)
; c2 -- end of piston which pushes piston to off position
; (usually pictured left)
; f -- the force along the piston (on, off, or center)
;
(DEVICE *what *location *x)
(DRIVE center_default *what)
STATUS *what (*x c1) *location *c)
STATUS *what (*x c2) *location *c)
STATUS *what (*x f) *location *f)
->
(= *f center) )

(DEVICE *what *location *x)
(DRIVE center_default *what)
STATUS *what (*x c1) *location off)
STATUS *what (*x c2) *location on)
STATUS *what (*x f) *location *f)
->
(= *f off) )

(DEVICE *what *location *x)
(DRIVE center_default *what)
(STATUS *what (*x c1) *location on)
(STATUS *what (*x c2) *location off)
(STATUS *what (*x f) *location *f)

->

(= *f on ) )

; (DEVICE solenoid )

; example -- lower hoist LHK1, LHK2

; c1 -- LC1, always pictured above *ie2
; c2 -- LC2, always pictured below *ie1
; ih1 -- always fed to the center of the solenoid
; ih2 -- always fed to the shell of the solenoid (usually low pressure)
; f -- the position of the piston within the solenoid.
; on is the position c1 always reinforces(pictured left)
; off is the opposite of on(pictured right)
; center is in the middle
; oh1 -- always pictured left of *ih1
; oh2 -- always pictured right of *ih1

( (DEVICE solenoid *location *x)
(STATUS solenoid (*x f) *location on)
(STATUS solenoid (*x ih2) *location *ih2)
(STATUS solenoid (*x oh1) *location *oh1)

->

(= *oh1 *ih2) )

( (DEVICE solenoid *location *x)
(STATUS solenoid (*x f) *location on)
(STATUS solenoid (*x ih1) *location *ih1)
(STATUS solenoid (*x oh2) *location *oh2)

-
(= *oh2 *ih1)

(DEVICE solenoid *location *x)
(STATUS solenoid (*x f) *location center)
(STATUS solenoid (*x ih2) *location *ih2)
(STATUS solenoid (*x oh1) *location *oh1)
->

(= *oh1 *ih2)

(DEVICE solenoid *location *x)
(STATUS solenoid (*x f) *location center)
(STATUS solenoid (*x ih2) *location *ih2)
(STATUS solenoid (*x oh2) *location *oh2)
->

(= *oh2 *ih2)

(DEVICE solenoid *location *x)
(STATUS solenoid (*x f) *location off)
(STATUS solenoid (*x ih1) *location *ih1)
(STATUS solenoid (*x oh1) *location *oh1)
->

(= *oh1 *ih1)

(DEVICE solenoid *location *x)
(STATUS solenoid (*x f) *location off)
(STATUS solenoid (*x ih2) *location *ih2)
(STATUS solenoid (*x oh2) *location *oh2)
->

(= *oh2 *ih2)

; (DEVICE piston2 *location *name)
;
; example -- lower hoist UVK5, UVK6
;
; m -- mechanical input (on = default position)
; which specifies that the piston is physically
; in one position or another.
; f — direction of force along the piston (on = default position)
; ih1 -- feeds the center of the piston
; ih2 -- feeds the chamber nearest the spring when the
; chamber is not being fed by ih1
; ih3 -- feeds the chamber furthest from the spring when
; the chamber is not being fed by ih1
;
; oh1 -- draws from chamber nearest spring
; oh2 -- draws from chamber furthest from spring
;
((DEVICE piston2 *location *x)
 (STATUS piston2 (*x f) *location on)
 (STATUS piston2 (*x ih1) *location *ih1)
 (STATUS piston2 (*x oh1) *location *oh1)
 ->
 ( = *oh1 *ih1 ))

((DEVICE piston2 *location *x)
 (STATUS piston2 (*x f) *location on)
 (STATUS piston2 (*x ih3) *location *ih3)
 (STATUS piston2 (*x oh2) *location *oh2)
 ->
 ( = *oh2 *ih3 ))

((DEVICE piston2 *location *x)
 (STATUS piston2 (*x f) *location off)
 (STATUS piston2 (*x ih2) *location *ih2)
 (STATUS piston2 (*x oh1) *location *oh1)
 ->
 ( = *oh1 *ih2 ))

((DEVICE piston2 *location *x)
 (STATUS piston2 (*x f) *location off)
 (STATUS piston2 (*x ih1) *location *ih1)
 (STATUS piston2 (*x oh2) *location *oh2)
 ->

(= *oh2 *ih1) )

; (DEVICE piston3 *location *name)
;
; example -- lower hoist UVK3
;
; m -- mechanical state of piston
; off = piston in direction of chamber1
; center = piston in middle...the default
; on = piston in direction of chamber2
; f -- direction of force (either on or off as in m)
; ih1 -- feeds the center of the piston
; ih2 -- feeds chamber1 when chamber not being fed by ih1
; ih3 -- feeds chamber2 when chamber not being fed by ih1
; oh1 -- draws from chamber1
; oh2 -- draws from chamber2
;
((DEVICE piston3 *location *x)
 (STATUS piston3 (*x f) *location off)
 (STATUS piston3 (*x ih2) *location *ih2)
 (STATUS piston3 (*x oh1) *location *oh1)
 ->
  (= *oh1 *ih2) )
((DEVICE piston3 *location *x)
 (STATUS piston3 (*x f) *location off)
 (STATUS piston3 (*x ih1) *location *ih1)
 (STATUS piston3 (*x oh2) *location *oh2)
 ->
  (= *oh2 *ih1) )
((DEVICE piston3 *location *x)
 (STATUS piston3 (*x f) *location center)
 (STATUS piston3 (*x ih2) *location *ih2)
 (STATUS piston3 (*x oh1) *location *oh1)
 ->)
(= *oh1 *ih2) )

(DEVICE piston3 *location *x)
(STATUS piston3 (*x f) *location center)
(STATUS piston3 (*x ih3) *location *ih3)
(STATUS piston3 (*x oh2) *location *oh2)
->
(= *oh2 *ih3) )

(DEVICE piston3 *location *x)
(STATUS piston3 (*x f) *location on)
(STATUS piston3 (*x ih1) *location *ih1)
(STATUS piston3 (*x oh1) *location *oh1)
->
(= *oh1 *ih1) )

(DEVICE piston3 *location *x)
(STATUS piston3 (*x f) *location on)
(STATUS piston3 (*x ih3) *location *ih3)
(STATUS piston3 (*x oh2) *location *oh2)
->
(= *oh2 *ih3) )

)

;

; Drive definitions for devices in this file
;

(mk_facts 0
 (DRIVE center_default solenoid)
 (DRIVE one_side_default piston2)
 (DRIVE center_default piston3)
)

;

; Pipes galore
(mk_device
  (pipe 2860021 (P11 (LHK2 ih1)))
  (pipe 2860021 (P2 P2j1))
  (pipe 2860021 (P2j1 P2j13))
  (pipe 2860021 (P2j13 (UVK3 ih1)))
  (pipe 2860021 (P2j13 P2j131))
  (pipe 2860021 (P2j131 (UVK5 ih1)))
  (pipe 2860021 (P2j131 P2j1311))
  (pipe 2860021 (P2j1 P2j11))
  (pipe 2860021 (P2j11 (UVK6 ih1)))
  (pipe 2860021 (P6 P2j11))
  (pipe 2860021 (P9 (UVK6 oh1)))
  (pipe 2860021 (P3 (UVK6 oh2)))
  (pipe 2860021 (P12 P12j2))
  (pipe 2860021 (P12j2 (LHK1 oh2)))
  (pipe 2860021 (P12j2 (UVK5 c2)))
  (pipe 2860021 (P7 P7j3))
  (pipe 2860021 (P7j3 P7j33))
  (pipe 2860021 (P7j33 (UVK5 ih3)))
  (pipe 2860021 (P7j33 P7j333))
  (pipe 2860021 (P7j333 (UVK5 ih2)))
  (pipe 2860021 (P7j333 (UVK3 ih3)))
  (pipe 2860021 (P7j333 P7j3333))
  (pipe 2860021 (P7j3333 (UVK5 c1)))
  (pipe 2860021 (P7j3333 (UVK3 ih2)))
  (pipe 2860021 (P7j33 P7j34))
  (pipe 2860021 (P7j34 (LHK1 ih2)))
  (pipe 2860021 (P7j34 (LHK2 ih2)))
  (pipe 2860021 (P7j3 P7j32))
  (pipe 2860021 (P7j32 (UVK4 h4)))
  (pipe 2860021 (P10 (LHK1 oh1)))
  (pipe 2860021 (P8 (UVK3 oh2)))
(pipe 2860021 (P1 P1j2))
(pipe 2860021 (P1j2 (UOK1 h1)))
(pipe 2860021 (P1j2 (UVK4 h3)))
(pipe 2860021 (P4 (UVK4 h6)))
(pipe 2860021 (P5 (UOK3 h2)))
(solenoid 2860021 LHK1)
(solenoid 2860021 LHK2)
(pipe 2860021 ((LHK2 oh1) (UVK3 c1)))
(pipe 2860021 ((LHK2 oh2) (UVK3 c2)))
(piston3 2860021 UVK3)
(pipe 2860021 ((UVK3 oh1) UVK3oh1j1))
(pipe 2860021 (UVK3oh1j1 (UVK4 h5)))
(pipe 2860021 (UVK3oh1j1 (UVK4 c1)))
(piston2 2860021 UVK5)
(pipe 2860021 ((UVK5 oh1) (UVK6 c1)))
(pipe 2860021 ((UVK5 oh2) (UVK6 c2)))
(piston2 2860021 UVK6)
(pipe 2860021 ((UVK6 ih2) UVK6ih23j1))
(pipe 2860021 ((UVK6 ih3) UVK6ih23j1))
(pipe 2860021 (UVK6ih23j1 (UVK4 h2)))
(orifice 2860021 UOK1)
(pipe 2860021 ((UOK1 h2) (UVK4 h1)))
(pipe 2860021 ((UVK4 c2) (UOK3 h1)))
(orifice 2860021 UOK3)
(rack 2860020 UCK1)
)
(load lhmore)
(write "Initialize the lhoist world")(print "")

/********************************************************************************
* filename: lhmore.hc
* lhoist and lhmore contain all of the facts and rules for WIF
********************************************************************************/
(mk_rules
  
(piston4)

  example -- lower hoist UVK10

  
c1 -- control line 1, the spring end of the piston
c2 -- control line 2, the non-spring end of the piston
f -- the direction of the force as a result of c1, c2 and spring.
  the spring applies force in the direction "on"
m -- the mechanical position of the piston, on is the default
center is resting against UCK2, off is towards spring.
i1 -- the pipe which feeds the chamber which is closest to c2.
i2 -- the pipe which feeds the chamber which is closest to c1.
o -- the output line which is on the center of the piston.

  (DEVICE piston4 *location *x)
  (STATUS piston4 (*x m) *location off)
  (STATUS piston4 (*x o) *location *o)
  (STATUS piston4 (*x i2) *location *i2)
  ->
  (= *o *i2) )

  (DEVICE piston4 *location *x)
  (STATUS piston4 (*x m) *location center)
  (STATUS piston4 (*x o) *location *o)
  (STATUS piston4 (*x i1) *location *i1)
  ->
  (= *o *i1) )

  (DEVICE piston4 *location *x)
  (STATUS piston4 (*x m) *location on)
  (STATUS piston4 (*x o) *location *o)
  (STATUS piston4 (*x i1) *location *i1)
  ->
(= *o *i1) )

; ;

; (piston5)

; example -- Lower Hoist, UVK9

; c1 -- the control line nearest the spring
; c2 -- the control line furthest from the spring
; h1, h2, ..., h9 -- the hydraulic lines leading to the piston which
; are pictured on the bottom on sketch 2860022.
; They are numbered from left to right along
; the bottom of piston UVK9
; h10 -- the hydraulic line pictured on the top of piston UVK9 on the
; sketch 2860022.
; f -- the direction of force as dictated by c1, c2 and the spring.
; "on" as always, is the default direction(away from spring)
; m -- the mechanical position of the piston, on means away from the
; spring, off means toward the spring center means against
; UCK2 but not in the slot.

; (DEVICE piston5 *location *x)
(STATUS piston5 (*x h2) *location *h2)
(STATUS piston5 (*x h3) *location *h3)
(STATUS piston5 (*x m) *location on)

->

(= *h2 *h3)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h5) *location *h5)
(STATUS piston5 (*x h6) *location *h6)
(STATUS piston5 (*x m) *location on)

->

(= *h5 *h6)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h8) *location *h8)
(STATUS piston5 (*x h9) *location *h9)
(STATUS piston5 (*x m) *location on)
->
(= *h8 *h9)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h1) *location *h1)
(STATUS piston5 (*x h2) *location *h2)
(STATUS piston5 (*x m) *location center)
->
(= *h1 *h2)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h4) *location *h4)
(STATUS piston5 (*x h5) *location *h5)
(STATUS piston5 (*x m) *location center)
->
(= *h4 *h5)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h8) *location *h8)
(STATUS piston5 (*x h9) *location *h9)
(STATUS piston5 (*x m) *location center)
->
(= *h8 *h9)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h1) *location *h1)
(STATUS piston5 (*x h2) *location *h2)
(STATUS piston5 (*x m) *location off)
->
(= *h1 *h2)

(DEVICE piston5 *location *x)
(STATUS piston5 (*x h4) *location *h4)
(STATUS piston5 (*x h5) *location *h5)
(STATUS piston5 (*x m) *location off)
   ->
   (= *h4 *h5)
(DEVICE piston5 *location *x)
(STATUS piston5 (*x h8) *location *h8)
(STATUS piston5 (*x h10) *location *h10)
(STATUS piston5 (*x m) *location off)
   ->
   (= *h8 *h10)

; ;
; (piston6 *location *name *name_up_slot_piston *piston_type_up_slot
 ;   *name_down_slot_piston *piston_type_down_slot)
;
;
; example -- lower hoist UCK2
;
; c1 -- the control line which reinforces the default
; c2 -- the control line which counters the default
; f -- the direction of force as dictated by c1, c2, gravity.
;   On is the default(direction of gravity)
; t1f -- Tongue one force. The force of the Tongue which will seek the
 ;   uppermost slot(this is a latch mech.). on means the
 ;   Tongue is seeking the uppermost slot in UCK2.
; t1m -- Tongue one mechanical position. on means in the uppermost slot
 ;   of UCK2. off means away from UCK2. center means resting
 ;   against UCK2, but not in slot.
; t2f -- Tongue one force. The force of the Tongue which will seek the
 ;   lowermost slot(this is a latch mech.). on means the
 ;   Tongue is seeking the lowermost slot in UCK2.
; t2m -- Tongue one mechanical position. on means in the lowermost slot
 ;   of UCK2. off means away from UCK2. center means resting
 ;   against UCK2, but not in slot.
(DEVICE piston6 *location *x)
(STATUS piston6 (*x f) *location off)
(STATUS piston6 (*x m) *location *m)
(STATUS piston6 (*x t1m) *location *t1m)
(<> *t1m on)

; (STATUS piston6 (*x alignment) *location load/unload_pos)
   ->
      (= *m off) )

(DEVICE piston6 *location *x)
(STATUS piston6 (*x f) *location on)
(STATUS piston6 (*x m) *location *m)
(STATUS piston6 (*x t2m) *location *t2m)
(<> *t2m on)
   ->
      (= *m on) )

(DEVICE piston6 *location *x)
(STATUS piston6 (*x m) *location on)
(STATUS piston6 (*x t1f) *location *t1f)
(STATUS piston6 (*x t1m) *location *t1m)
   ->
      (= *t1f *t1m) )

(DEVICE piston6 *location *x)
(STATUS piston6 (*x m) *location off)
(STATUS piston6 (*x t1f) *location on)
(STATUS piston6 (*x t1m) *location *t1m)
   ->
      (= *t1m center) )

(DEVICE piston6 *location *x)
(STATUS piston6 (*x m) *location off)
(STATUS piston6 (*x t2f) *location *t2f)
(STATUS piston6 (*x t2m) *location *t2m)
   ->
      (= *t2f *t2m) )
(DEVICE piston6 *location *x)
(STATUS piston6 (*x m) *location on)
(STATUS piston6 (*x t2f) *location on)
(STATUS piston6 (*x t2m) *location *t2m)

->

(= *t2m center)

(rack)

example -- Lower Hoist UCK1

c1 -- the control line which reinforces the default, gravity.
c2 -- the control line which fights the default.
f -- the direction of force, with gravity is on, against it off.
m -- the mechanical position of the rack. With gravity is on, all
the way against it is off and anything else is center.
debris -- debris = "no_debris" is the rack is free of debris.

(DEVICE rack *location *x)
(STATUS rack (*x f) *location *f)
(STATUS rack (*x m) *location on)
(STATUS rack (*x debris) *location no_debris)

->

(= *f on)

(DEVICE rack *location *x)
(STATUS rack (*x f) *location *f)
(STATUS rack (*x m) *location off)
(STATUS rack (*x debris) *location no_debris)

->

(= *f off)

(DEVICE rack *location *x)
(STATUS rack (*x f) *location *f)
(STATUS rack (*x m) *location *m)
(STATUS rack (*x debris) *location no_debris)
->
   (= *f *m) 
(DEVICE rack *location *x)
(STATUS rack (*x f) *location *f)
(STATUS rack (*x m) *location center)
->
   (STATUS rack (*x debris) *location debris) )

; (coupling)

; Example -- Lower Hoist coupling

; f -- the direction of force of the coupling, on is coupled,
;     off is not coupled.
; m -- the mechanical position of the coupling, on is coupled,
;     off is not coupled.
; (UCK2 m) -- the position of UCK2
; (UCK1 m) -- the position of UCK1

(DEVICE coupling 2860021 LH_coupling)
(STATUS coupling (LH_coupling f) 2860021 on)
(STATUS coupling (LH_coupling m) 2860021 *m)
(STATUS chain (LH_chain alignment) 2860020 *align)
(STATUS rack (UCK1 m) 2860020 *rm)
->
   (= *align load/unload_pos)
   (<> *rm center)
   (= *m on) 
(DEVICE coupling 2860021 LH_coupling)
(STATUS coupling (LH_coupling f) 2860021 off)
(STATUS coupling (LH_coupling m) 2860021 *m)
->
(= *m off) )

; (DEVICE chain *location *name)

; *chain_alignment -- this variable must be "load/unload_pos" if
; the coupling is to engage.

; (STATUS chain (*name alignment) *where load/unload_pos)
->
(= 1 1)

(STATUS chain (*name alignment) *where not-load/unload_pos)
->
(= 1 1)

; (DEVICE switch *location (*name *val))

; *val -- the value m must be for the switch to be on

; Example -- Lower Hoist SIK20

; m -- the mechanical input to the switch apparatus
; oe -- the output electrical line. on means m is in position *val.
; alignment -- alignment is "properly_aligned" if the mechanism
; mechanically works properly.
; charge -- charge is "fully_charged" if the magnet is charged

(DEVICE switch *location (*x *val))
(STATUS switch (*x m) *location *val)
(STATUS switch (*x alignment) *location properly_aligned)
(STATUS switch (*x charge) *location fully_charged)
(STATUS switch (*x oe) *location *oe)
->
(= *oe on)

(DEVICE switch *location (*x *val))

(STATUS switch (*x m) *location *val1)

(<> *val *val1)

(STATUS switch (*x alignment) *location properly_aligned)

(STATUS switch (*x charge) *location fully_charged)

(STATUS switch (*x oe) *location *oe)

->

(= *oe off)

;

(DEVICE piston7 *location *name))

;

Example -- Lower Hoist UVK4

;

c1 -- the control line which reinforces gravity

c2 -- the control line which counters gravity

f -- the direction of force of the piston. down is on, up is off.

m -- the mechanical position of the piston. down is on, up is off.

h1, ..., h5 -- the hydraulic lines pictured on the left of the piston

as pictured from top to bottom on sketch 2860021.

h6 -- the hydraulic line which is pictured on the right of the piston

as seen on sketch 2860021.

;

(DEVICE piston7 *location *x)

(STATUS piston7 (*x m) *location off)

(STATUS piston7 (*x h1) *location *h1)

(STATUS piston7 (*x h2) *location *h2)

->

(= *h1 *h2)

(DEVICE piston7 *location *x)

(STATUS piston7 (*x m) *location off)

(STATUS piston7 (*x h4) *location *h4)

(STATUS piston7 (*x h6) *location *h6)
The drive definitions for all mechanism in this file

(mk_facts 0
  (DRIVE one_side_default piston4)
  (DRIVE one_side_default piston5)
  (DRIVE one_side_default piston6)
  (DRIVE no_default piston7)
  (DRIVE one_side_default rack)
)

pipes, pipes, and more pipes

(mk_device
  (external_pipe 2860021 2860020 (P7j32 P7j322))
(external_pipe (2860020 2860022) (P7j322 P7j3221))
(external_pipe (2860021 2860022) (P5 P3))
(external_pipe (2860021 2860022) (P4 P4))
(external_pipe (2860021 2860022) (P8 P5))
(piston5 2860022 UVK9)
(piston4 2860022 UVK10)
(piston6 2860022 UCK2)
(piston7 2860021 UVK4)
(coupling 2860021 LH_coupling)
(pipe 2860021 ((UVK4 f) (LH_coupling f)))
(pipe 2860021 ((UVK4 m) (LH_coupling m)))
(orifice 2860022 UOK4)
(switch 2860021 (SIK20 off))
(switch 2860020 (SIK17 off))
(switch 2860020 (SIK18 on))
(switch 2860022 (SIK16 on))
(switch 2860020 (SIK7 load/unload_pos))
(pipe 2860021 ((UVK4 m) (SIK20 m)))
(pipe 2860020 ((UCK1 m) (SIK17 m)))
(pipe 2860020 ((UCK1 m) (SIK18 m)))
(pipe 2860022 ((UVK10 m) (SIK16 m)))
(pipe 2860020 ((LH_chain alignment) (SIK7 m)))
(rack 2860020 UCK1)
(chain 2860022 LH_chain)
(pipe 2860022 (P4 P4j2))
(pipe 2860022 (P4j2 (UVK9 c2)))
(pipe 2860022 (P4j2 (UVK9 h10)))
(pipe 2860022 (P3 (UVK9 h5)))
(pipe 2860022 (P1 (UVK9 h2)))
(pipe 2860022 (P2 (UVK9 h3)))
(pipe 2860022 (P7j3221 P7j3221))
(pipe 2860022 (P7j32211 P7j322112))
(pipe 2860022 (P7j322112 P7j3221122))
(pipe 2860022 (P7j322112 (UVK9 h9)))
(pipe 2860022 (P7j3221122 P7j32211222))
(pipe 2860022 (P7j3221122 (UVK9 h7)))
(pipe 2860022 (P7j32211222 P7j322112222))
(pipe 2860022 (P7j32211222 (UVK9 h4)))
(pipe 2860022 (P7j322112222 (UVK9 h1)))
(pipe 2860022 (P7j322112222 (UVK9 c1)))
(pipe 2860022 (UVK10 o) UVK10oj1))
(pipe 2860022 (UVK10oj1 (UVK9 h6)))
(pipe 2860022 (UVK10oj1 (UCK2 c2)))
(pipe 2860022 (UVK9 h8) (UOK4 h1)))
(pipe 2860022 (UOK4 h2) (UCK2 c1)))
(pipe 2860022 (P7j322111 P7j32211))^)
(pipe 2860022 (P7j322111 (UVK10 i1)))
(pipe 2860022 (P7j322111 (UVK10 c1)))
(pipe 2860022 (P5 P5j2))
(pipe 2860022 (P5j2 (UVK10 i2)))
(pipe 2860022 (P5j2 (UVK10 c2)))
(external_pipe (2860021 2860020) (P9 P2))
(external_pipe (2860021 2860020) (P3 P1))
(pipe 2860020 (P2 (UCK1 c1)))
(pipe 2860020 (P1 (UCK1 c2)))
(pipe 2860022 ((UVK10 f)(UCK2 t1f)))
(pipe 2860022 ((UVK10 m)(UCK2 t1m)))
(pipe 2860022 ((UVK9 f) (UCK2 t2f)))
(pipe 2860022 ((UVK9 m) (UCK2 t2m)))
(external_pipe (2860022 2860020) ((UCK2 alignment)(LH_chain alignment)))
)

/*************************************************************************/
*  filename:  bo
*  This file sets up the constants for the cycles of the lower hoist
*  
*************************************************************************/
(assert

; This file contains the information which is constant for various
; cycles of the lower hoist.

; initial_truth is a set of external observable points whose values
; are always known.

; the routine (init) propagates values from these lines to the inputs
; of all the pistons and defines this expanded set as (always_true *x)
; This represents all assumed values in the system.

((initial_truth ( (STATUS point P11 2860021 on)
    (STATUS point P2 2860021 on)
    (STATUS point P1 2860021 off)
    (STATUS point P7 2860021 off)
    (STATUS point P2 2860022 off) )))

((init_system ( (STATUS piston7 (UVK4 m) 2860021 off)
    (STATUS piston6 (UCK2 m) 2860022 off)
    (STATUS chain (LH_chain alignment) 2860020 load/unload_pos)
    (STATUS solenoid (LHK1 c1) 2860021 off)
    (STATUS solenoid (LHK1 c2) 2860021 off)
    (STATUS solenoid (LHK2 c1) 2860021 off)
    (STATUS solenoid (LHK2 c2) 2860021 off) )))

((engage_coupling ( (STATUS solenoid (LHK1 c1) 2860021 off)
    (STATUS solenoid (LHK1 c2) 2860021 off)
    (STATUS solenoid (LHK2 c1) 2860021 off)
    (STATUS solenoid (LHK2 c2) 2860021 on) )))

((track_extend ( (STATUS solenoid (LHK1 c1) 2860021 on)
    (STATUS solenoid (LHK1 c2) 2860021 off)
((drop_engage_coupling (STATUS solenoid (LHK1 c1) 2860021 on)
  (STATUS solenoid (LHK1 c2) 2860021 off)
  (STATUS solenoid (LHK2 c1) 2860021 off)
  (STATUS solenoid (LHK2 c2) 2860021 off) ) ))

((disengage_coupling ( (STATUS solenoid (LHK1 c1) 2860021 on)
  (STATUS solenoid (LHK1 c2) 2860021 off)
  (STATUS solenoid (LHK2 cl) 2860021 on)
  (STATUS solenoid (LHK2 c2) 2860021 off) ) ))

((rack_retract ( (STATUS solenoid (LHK1 cl) 2860021 off)
  (STATUS solenoid (LHK1 c2) 2860021 off)
  (STATUS solenoid (LHK2 cl) 2860021 on)
  (STATUS solenoid (LHK2 c2) 2860021 off) ) ))

((drop_disengage_coupling ( (STATUS solenoid (LHK1 c1) 2860021 off)
  (STATUS solenoid (LHK1 c2) 2860021 off)
  (STATUS solenoid (LHK2 c1) 2860021 off)
  (STATUS solenoid (LHK2 c2) 2860021 off) ) ))

/)*********************************/
* filename:  lhsupport.hc
* This file defines the modal categories for the facts.
* It's main purpose is to preprocess a list of devices and place them
* in the form WIF expects. For example, the name of a pipe defines
* its endpoints. So just from the name, the device PIPE can be asserted
* (using WIF's mk_facts), and two status points can also be asserted.
* This routine effectively cuts down on the amount of busy work necessary
* in setting up the HOIST in the file lhoist and lhmore.
(load bo)
(assert
  ((get_modal_cat (DEVICE .*any) 4))
((get_modal_cat (STATUS chain.*any) 4))
((get_modal_cat (STATUS rack (*x debris).*y) 4))
((get_modal_cat (STATUS .*any) 5)))

; This file uses prolog as a preprocessor to set up all of the various devices
; A pipe, for example, has two end points. The pipe itself must be put in W0
; along with it's end points. More complicated devices have more i/o variables
;
;
;    mk_device
;
((mk_device))
((mk_device *h.*t) if (mk_dev *h)
  (mk_device_set_up (DEVICE . *h))
  (mk_device . *t))

((mk_dev (pipe.*y) ) if      (mk_facts 3 (DEVICE pipe . *y)))
((mk_dev (external_pipe.*y)) if (mk_facts 3 (DEVICE external_pipe . *y)))
((mk_dev *y) if             (mk_facts 4 (DEVICE . *y)))

((mk_device_set_up *x) if (mk_dev_su *x)
  (fail))
((mk_device_set_up *x))

((mk_dev_su (DEVICE pipe *loc (*a *b)) ) if
  (init2 *a *loc)
(init2 *b *loc)
  (cut)
((mk_dev_su (DEVICE external_pipe (*loc1 *loc2) (*a *b)) ) if
  (init2 *a *loc1)
  (init2 *b *loc2)
  (cut))
((mk_dev_su (DEVICE solenoid *loc *name)) if
  (init1 solenoid (*name c1) *loc)
  (init1 solenoid (*name c2) *loc)
  (init1 solenoid (*name ih1) *loc)
  (init1 solenoid (*name ih2) *loc)
  (init1 solenoid (*name oh1) *loc)
  (init1 solenoid (*name oh2) *loc)
  (init1 solenoid (*name f) *loc)
)((mk_dev_su (DEVICE switch *loc (*name *val)) ) if
  (init1 switch (*name m) *loc)
  (initval switch (*name alignment) *loc properly_aligned 5)
  (initval switch (*name charge) *loc fully_charged 5)
  (init1 switch (*name oe) *loc)
)((mk_dev_su (DEVICE orifice *loc *name)) if
  (init1 orifice (*name h1) *loc)
  (init1 orifice (*name h2) *loc)
)((mk_dev_su (DEVICE *what *loc *name)) if
  (fact (DRIVE no_default *what) *any1)
  (init1 *what (*name c1) *loc)
  (init1 *what (*name c2) *loc)
  (init1 *what (*name f) *loc)
  (init1 *what (*name m) *loc)
)((mk_dev_su (DEVICE *what *loc *name)) if
  (fact (DRIVE one_side_default *what) *any1)
  (init1 *what (*name c1) *loc)
  (init1 *what (*name c2) *loc)
  (init1 *what (*name f) *loc)
)((mk_dev_su (DEVICE *what *loc *name)) if
(fact (DRIVE center_default *what) *any1)
(initl *what (*name c1) *loc)
(initl *what (*name c2) *loc)
(initl *what (*name f) *loc center)

((mk_dev_su (DEVICE piston2 *loc *name) ) if
 (initl piston2 (*name ih1) *loc)
 (initl piston2 (*name ih2) *loc)
 (initl piston2 (*name ih3) *loc)
 (initl piston2 (*name f) *loc)
 (initl piston2 (*name oh1) *loc)
 (initl piston2 (*name oh2) *loc) )

((mk_dev_su (DEVICE piston3 *loc *name) ) if
 (initl piston3 (*name ih1) *loc)
 (initl piston3 (*name ih2) *loc)
 (initl piston3 (*name ih3) *loc)
 (initl piston3 (*name f) *loc)
 (initl piston3 (*name oh1) *loc)
 (initl piston3 (*name oh2) *loc) )

((mk_dev_su (DEVICE piston4 *loc *name) ) if
 (initl piston4 (*name m) *loc)
 (initl piston4 (*name o) *loc)
 (initl piston4 (*name il) *loc)
 (initl piston4 (*name i2) *loc) )

((mk_dev_su (DEVICE piston5 *loc *name) ) if
 (initl piston5 (*name m) *loc)
 (initl piston5 (*name h1) *loc)
 (initl piston5 (*name h2) *loc)
 (initl piston5 (*name h3) *loc)
 (initl piston5 (*name h4) *loc)
 (initl piston5 (*name h5) *loc)
 (initl piston5 (*name h6) *loc)
 (initl piston5 (*name h7) *loc)
 (initl piston5 (*name h8) *loc)
 (initl piston5 (*name h9) *loc) )
(initl piston5 (*name h10) *loc)
(initl piston5 (*name m) *loc)
((mk_dev_su (DEVICE piston6 *loc *name) ) if
  (initl piston6 (*name m) *loc)
    (initl piston6 (*name t1f) *loc)
    (initl piston6 (*name t1m) *loc)
    (initl piston6 (*name t2f) *loc)
    (initl piston6 (*name t2m) *loc)
    (initl piston6 (*name chain_alignment) *loc))

((mk_dev_su (DEVICE piston7 *loc *name) ) if
  (initl piston7 (*name m) *loc)
  (initl piston7 (*name h1) *loc)
  (initl piston7 (*name h2) *loc)
  (initl piston7 (*name h3) *loc)
  (initl piston7 (*name h4) *loc)
  (initl piston7 (*name h5) *loc)
  (initl piston7 (*name h6) *loc))

((mk_dev_su (DEVICE rack *loc *name) ) if
  (initl rack (*name m) *loc)
  (initl rack (*name h10) *loc)
  (initval rack (*name debris) *loc no_debris 4))

((mk_dev_su (DEVICE coupling *loc *name) ) if
  (initl coupling (*name m) *loc)

((mk_dev_su (DEVICE chain *loc *name) ) if
  (initval chain (*name alignment) *loc load/unload_pos 4))

((initl *what *x *location) if
  (fact (STATUS *what *x *location *any) *modal)
  (cut))

((initl *what *x *location) if
  (mk_facts 5 (STATUS *what *x *location unknown))
  (cut))
((initval *what *x *location *val *mod) if
  (fact (STATUS *what *x *location *any) *modal)
  (retract ((fact (STATUS *what *x *location *any) *modal)))
  (fail))

((initval *what *x *location *val *mod) if
  (mk_facts *mod (STATUS *what *x *location *val))
  (cut))

((init2 (*x *y) *loc) if (cut))
((init2 *x *loc) if (initl point *x *loc) (cut))

((filter nil nil 0) if (cut))
((filter (not *x) nil 0) if (cut))
((filter *x nil 0) if
  (:- *y graphics_only)
  (== t *y)
  (not_observable *x)
  (cut))
((filter (or. *orlist) *orans 1) if
  (cut)
  (filter_or *orlist *orans1)
  (filter_fix_up (or. *orans1) *orans))
((filter (*a.*b) (*a.*b) 0) if
  (atomp *a)
  (cut))
((filter (*h.*t) *ans *val) if
  (filter_and (*h.*t) *rslt *val)
  (filter_fix_up *rslt *ans))

((filter_or nil nil) if (cut))
((filter_or (*h.*t) (*a.*sofar)) if
  (filter *h *a *any)
  (cut)
  (filter_or *t *sofar))
((filter_and nil nil 0) if (cut))
((filter_and (=> . *t) (*anst) 0) if
 (filter_and *t *rslt *what))
; (filter_and1 *what *rslt *ans)
 (filter_fix_up *rslt *anst)
 (cut))
((filter_and (*h.*t) *rslt *val3) if
 (filter *h *a *val1)
 (cut)
 (filter_and *t *ans *val2)
 (filter_comb1 *val1 *val2 *val3)
 (filter_comb2 *a *ans *rslt))

((filter_and1 1 *in (=> . *in)))
((filter_and1 0 *in *in))

((filter_comb1 0 0 0))
((filter_comb1 *x *y 1))

((filter_comb2 nil *x *x))
((filter_comb2 *h *t (*h.*t)))

((filter_fix_up (or) nil))
((filter_fix_up (or *x) *ans) if (filter_fix_up *x *ans))
((filter_fix_up (*x nil) *ans) if (filter_fix_up (*x) *ans))
((filter_fix_up ((*a.*b)) *ans) if
 (filter_fix_up (*a.*b) *ans))
((filter_fix_up (nil) nil))
((filter_fix_up *x *x))

((not_observable (STATUS rack (*a f).*b)))
((not_observable (STATUS piston4 (*a f).*b)))
((not_observable (STATUS piston5 (*a f).*b)))
(not_observable (STATUS piston7 (*a f).*b))
(\not_observable (STATUS coupling (*a f).*b))
\not_observable (STATUS piston6 (*a f).*b))
(\not_observable (STATUS piston6 (*a t1f).*b))
(\not_observable (STATUS piston6 (*a t2f).*b))

((add_loop *list) if
  (counterfactual *list 4 *ans)
  (cut)
  (filter *ans *rslt *any)
    (pprint *rslt)
  (cut)
    (choose_one *list *ans *rslt1 *any1)
  (update_facts *rslt1))

((add_loop1 *list *rslt) if
  (counterfactual *list 4 *ans)
  (cut)
  (filter *ans *rslt *any)
  (cut)
    (choose_one *list *ans *rslt1 *any1)
  (update_facts *rslt1))

/**
*  filename:  test.hc
*/
; deterministic machine) as it would be if the hoist were working correctly.

; 

((test *x *where *modval) if 
  (test_startup *x *where) 
  (test_loop *x *where *modval) 
  (test_shutdown *x *where))

((autotest *x *where *m) if 
  (test_startup *x *where) 
  (write "INITIAL VALUES:")
  (print "")
  (for_all ((fact (STATUS *dev (*x *part) *where *val) *ml))
    ((write "")
     (write "(STATUS " *dev " (*x " " *part ") " where " " *val ")")
     (print "") )
  (print "")
  (for_all ((fact (STATUS *dev (*x *part) *where *val) *m) *ml))
  (counterfactual ((STATUS " *dev " (*x " " *part ") " where " " *val ")")
   (pprint *ans1)
   (print "")
   (counterfactual ((STATUS " *dev " (*x *part) *where off)) *m *ans2)
   (pprint *ans2)
   (print "")
   (print ")
   (test_shutdown *x *where))

((test_startup *x *where) if 
  (for_all ((fact (DEVICE *what *where ((*x *y) *z)) *modal))
    ((retract ((fact (DEVICE *what *where ((*x *y) *z)) *modal)))
    (assert ((fact_store *x *where
      (DEVICE *what *where ((*x *y) *z)) *modal)))))
  (for_all ((fact (DEVICE *what *where (*z (*x *y))) *modal))
    ((retract ((fact (DEVICE *what *where (*z (*x *y))) *modal))))
(assert ((fact_store *x *where
  (DEVICE *what *where (*z (*x *y))) *modal)))))

(test_shutdown *x *where) if
(for_all ((fact_store (DEVICE *what *where ((*x *y) *2)) *modal))
  (retract ((fact_store (DEVICE *what *where ((*x *y) *2)) *modal)))
  (assert ((fact *x *where
    (DEVICE *what *where ((*x *y) *2)) *modal))))
)

(test_loop *x *where *modal) if
(write "TEST " *x "? (y/n): ")
(:= *line (getline))
(:= *response (strsub *line 1 1))
(== *response "n")
(test_loop *x *where *modalval) if
(ans_init *tok)
(for_all ((fact (STATUS *any (*x *y) *where *val) *modal))
  (ans_store *tok (fact (STATUS *any (*x *y) *where *val) *modal)))
(ans_retrieve *tok *bag)
(test_group *bag *small_bag)
(print "")
(write "Items changed for run:" )
(print "")
(print *small_bag)
(cf *small_bag *modalval)
(print "---------------------------------------------")
(test_loop *x *where *modalval))
((test_group nil nil))
((test_group ((fact (STATUS *d *w *where *val) *modal).*t)
  (STATUS *d *w *where *newval).*sofar)) if
  (write "(STATUS *d "*w "*where " is now "*val ",")
  (print ")")
  (write "Enter new value (<cr> means keep same value): ")
 (:= *newval (getline))
 (!= *newval ")")
  (test_group *t *sofar))
((test_group (*h.*t) *sofar) if
  (test_group *t *sofar))

((flatten nil nil))
((flatten (*h.*t) (*h.*sofar)) if
  (is_a_fact *h)
  (flatten *t *sofar))
((flatten (*h.*t) *rslt) if
  (flatten *h *ans1)
  (flatten *t *ans2)
  (append *ans1 *ans2 *rslt))

((is_a_fact (*h.*t)) if (atomp *h))

((initialize) if (initial_truth *x)
  (counterfactual *x 4 *ans)
  (filter *ans *rslt *any)
  (flatten *rslt *all_true)
  (assert ((always_true1 *all_true)))
  (choose_one *x *ans *rslt1 *any1)
  (update_facts *rslt1)
  (init_system *list)
  (append *list *all_true *list1)
  (set vanilla_lsp *list1)
  (add_loop *list1))
((always_true *rest) if
  (always_true *rest)
  (for_all (((fact (STATUS piston7 (*x1 p) *y1 *val1) *any1))
    ((retract ((fact (STATUS piston7 (*x1 p) *y1 *val1) *any1)))))
  (for_all (((fact (STATUS piston7 (*x2 m) *y2 *val2) *any2))
    ((assert0 ((fact (STATUS piston7 (*x2 p) *y2 *val2) 0))))))
  (phase1) if (always_true *list1)
    (engage_coupling *list2)
    (append1 *list1 *list2 *list)
    (add_loop1 *list *out)  
    (set phase1_lsp *out) )
  (phase2) if (always_true *list1)
    (rack_extend *list2)
    (append1 *list1 *list2 *list)
    (add_loop1 *list *out)  
    (set phase2_lsp *out))
  (phase3) if (always_true *list1)
    (drop_engage_coupling *list2)
    (append1 *list1 *list2 *list)
    (add_loop1 *list *out)  
    (set phase3_lsp *out))
  (phase4) if (always_true *list1)
    (disengage_coupling *list2)
    (append1 *list1 *list2 *list)
    (add_loop1 *list *out)  
    (set phase4_lsp *out))
  (phase5) if (always_true *list1)
    (rack_retract *list2)
    (append1 *list1 *list2 *list)
    (add_loop1 *list *out)  
    (set phase5_lsp *out))
  (phase6) if (always_true *list1)
(drop_disengage_coupling *list2)
(append1 *list1 *list2 *list)
   (add_loop1 *list *out)
   (set phase6_lsp *out))
)
(load "lhstates.hc")
(set graphics_only t)
The vita has been removed from the scanned document.