Truth Maintenance Systems for Problem Solving

by

Jon Doyle

B. S., University of Houston
(1974)

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

Massachusetts Institute of Technology

May, 1977

Signature of Author .....
Department of Electrical Engineering and Computer Science
May 12, 1977

Certified by .....
Thesis Supervisor

Accepted by .....
Chairman, Departmental Committee on Graduate Students

Truth Maintenance Systems for Problem Solving

by

Jon Doyle

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1977 in partial fulfillment of the requirements
for the Degree of Master of Science.

ABSTRACT

This report describes progress that has been made in the ability of a computer system to understand and reason about its own reasoning faculties. A new method for representing knowledge about beliefs has been developed. This representation, called a non-monotonic dependency system, extends several similar previous representation forms for such knowledge, and has been employed in developing new strategies for representing assumptions, describing hierarchical structures, backtracking, and controlling problem solving systems.

This representation is employed by a set of computer programs called the Truth Maintenance System (TMS), which makes these abilities available as a domain-independent tool for problem solving. The TMS accepts as input justifications for belief in components of program knowledge, and uses these justifications to determine which beliefs endure the discovery of new information and the changing of hypotheses.

The major points of the report are that appropriately recorded justifications for beliefs can be used to efficiently maintain the current set of program beliefs, and can further be used in a variety of tasks, including those of hypothetical reasoning, separating levels of detail in explanations, and dependency-directed backtracking.

Thesis Supervisor: Gerald Jay Sussman
Title: Associate Professor of Electrical Engineering

To my parents

Acknowledgements

I owe much to many for their attention and cooperation during this research. Many of the ideas in this report are due, directly or indirectly, to Gerald Jay Sussman, Guy Lewis Steele Jr., and Richard Matthew Stallman. Much of my progress on these problems has derived from the patient advice, inspiring encouragement, and close cooperation of Gerry Sussman. I particularly wish to thank:

Tomas Lozano-Perez and Drew McDermott for explaining planning to me,

Johan de Kleer, Scott Fahlman, Richard Brown, Kurt VanLehn, Joseph Schatz, and Marvin Minsky for discussion of related problems of problem solving,

Marilyn Matz for her informed perspective on the frame problem, which substantially influenced my own view,

Gerry Sussman, Beth Levin, and Marilyn Matz for several appreciations,

Bob Woodham, Scott Fahlman, Mitch Marcus, Howie Shrobe, Ben Kuipers, Ken Forbus, Bob Sjoberg, Brian Smith and Tom Knight for putting up with me, even when I was obviously out to lunch,

Eric Grimson, Kent Stevens, and Chuck Rich for encouragement,

Johan de Kleer and Marilyn Matz for careful reading,

The Maharal of Prague for several recursions, and

The Fannie and John Hertz Foundation, which supported my research.

CONTENTS

I. Introduction

A. Overview of the Report

This report describes progress that has been made in the ability of a computer system to understand and reason about its own reasoning faculties. A new method for representing knowledge about beliefs has been developed. This representation, called a non-monotonic dependency system, extends and clarifies several similar previous representation forms for such knowledge, and has been employed in developing new strategies for representing assumptions, backtracking, and controlling problem solving systems.

The report begins with a functional description of the TMS (Truth Maintenance System), a set of running programs that embodies the representation and strategies mentioned above. An illustrative example is then presented to lend substance to the following discussion.

The following two chapters discuss in detail the representation for knowledge about belief, efficient techniques for maintaining beliefs in the event of changing information, and techniques for using the representation in several interesting problems of reasoning and control in problem solving systems. Both of these chapters begin with a brief historical perspective on the theme of the chapter. Both chapters end by surveying

relevant current problem solving efforts from the perspective of the ideas developed in the chapter. Readers wishing an overview of the current work are encouraged to read the initial sections of these chapters.

The final chapter provides a summary discussion of the key ideas developed in the report, accompanied by a discussion related problems for future research. Three appendicies provide a glossary of the concepts and terms employed herein, a discussion of related representational systems, and an implementation of a truth maintenance system.

B. A Functional Description of Truth Maintenance Systems

A truth maintenance system is a combination of a representation for recording dependencies between program beliefs and procedures for effecting any updating of beliefs necessary upon the addition of new dependencies. Such a system can easily be used by processes for reasoning about the recorded program reasoning. In particular, processes for non-chronological, dependency-directed backtracking and hypothetical reasoning are particularly straightforward in implementation given the representations of a truth maintenance system.

The basic operation of a truth maintenance system is to attach a justification to a fact. A fact can be linked with any component of program knowledge which is to be connected with other components of program information. Typically, a fact might be

connected with each assertion and rule in a data base, or might be attached, with differing meanings, to various subsystem structures. The TMS decides, on the basis of the justifications attached to facts, which beliefs in the truth of facts are supported by the recorded justifications. From the justifications used in this judgement of beliefs, a number of relations between beliefs are determined, such as the set of beliefs depending on each particular belief or the beliefs upon which a particular belief depends.

The more complex operation of the TMS is required when new justifications change previously existing beliefs. In such cases, the status of all beliefs depending on the changed beliefs must be redetermined. This process is termed truth maintenance, and can be efficiently implemented.

Several useful processes are supported by the above functions and representations. It is a straightforward matter to interrogate the TMS representation for the basic material of explanations of beliefs. More sophisticated uses of the recorded justifications are in hypothetical reasoning, in which the method of conditional proof is supported, in the process of generalization, which is a particular application of hypothetical reasoning, and in dependency-directed backtracking, which employs the recorded dependencies to locate precisely those hypotheses relevant to the failure, and which uses the conditional proof mechanisms to summarize the cause of the contradiction in terms of these hypotheses to prevent its future occurrence.

C. An Example

On the ground,
Sleep sound.
I'll apply
To your eye,
Gentle lover, remedy.
When thou wakest,
Thou takest
True delight
In the sight
Of thy former lady's eye.
And the country proverb known,
That every man should take his own,
In your waking shall be shown.
Jack shall have Jill,
Nought shall go ill,
The man shall have his mare again
and all shall be well.

William Shakespeare, *A Midsummer Night's Dream*

To demonstrate the flavor of the use of recorded justifications in determining belief and in hypothetical reasoning, this section presents a simple example involving the making of assumptions, truth maintenance, and dependency-directed backtracking. Further examples in the body of the report will elaborate some of the finer points of truth maintenance and the separation of levels of detail.

For this example, we set modesty aside and attempt to imitate William Shakespeare in designing the plot of *A Midsummer Night's Dream.* The major problem in

this is to prevent the story from turning into a tragedy. Dependency-directed backtracking provides the tool by which changes in the players' attitudes are determined, such that these changes eventually result in a consistent (happy) set of attitudes. Unfortunately, these methods do not hint at what magic is required to effect these changes.

The problem involves the four individuals Demitrius, Helena, Hermia, and Lysander. Initially, the following beliefs are entertained.

```
(assert (loves Hermia Lysander) (premise))
F-1 (LOVES HERMIA LYSANDER) (PREMISE)

(assert (loves Helena Demitrius) (premise))
F-2 (LOVES HELENA DEMITRIUS) (PREMISE)
```

The information and rules of our example will be framed in a simple rule-based language called SCHPDS, developed by G. J. Sussman and J. de Kleer. Assertions, as above, are of the form (ASSERT <assertion pattern> <justification>) and should be read as "belief in <assertion pattern> is justified by <justification>." The justifications refer to functions which will accept the information transmitted in the justifications and implement the necessary TMS justifications between facts.

The next specification is that of the amatory preferences of the men, who are

creatures easily swayed by flowers and dependency-directed backtracking.

```
(assume (loves Demitrius Hermia) (premise))
F-3 (ASSUMED (LOVES DEMITRIUS HERMIA)) (PREMISE)
F-4 (NOT (LOVES DEMITRIUS HERMIA)) ()
F-5 (LOVES DEMITRIUS HERMIA) (ASSUMPTION F-3 F-4)
```

Assumptions are the fundamental use of non-monotonic justifications in the dependency system. Thus the assumption of F-5 above is accomplished by asserting the reason for the assumption, F-3, and establishing belief in F-5 based on this reason and on, as will be explained further in the next chapter, the lack of belief in F-4. Thus as long as there are no reasons for believing otherwise, F-5 will be believed. At this point, F-4 is not believed, for no reasons exist supporting such belief, and F-5 is believed, since F-3 is believed and F-4 is not.

```
(assume (loves Lysander Hermia) (premise))
F-6 (ASSUMED (LOVES LYSANDER HERMIA)) (PREMISE)
F-7 (NOT (LOVES LYSANDER HERMIA)) ()
F-8 (LOVES LYSANDER HERMIA) (ASSUMPTION F-6 F-7)

(rule (:n (not (loves Demitrius Hermia)))
      (assert (loves Demitrius Helena) (quality-not-quantity :n)))
```

This rule provides for Demitrius' love if he falls from love with Hermia by providing the alternative of Helena. The format of the rule is a pattern, which specifies both a variable (marked by the colon prefix) to be bound to the fact name of the matching assertion, and the pattern which assertions are to be matched by for the body to be executed. If a matching assertion is present, the rule will bind the variables of the pattern to the appropriate values and evaluate each expression of the body. In the rule above, if it becomes known that Demitrius does not love Hermia, the rule will justify the belief that Demitrius loves Helena.

```
(rule (:n (not (loves Lysander Hermia)))
      (assert (loves Lysander Helena) (love-in-idleness :n)))
```

Next, some of the more unfortunate aspects of the world are specified.

```
(assert (jealous Lysander) (premise))
F-9 (JEALOUS LYSANDER) (PREMISE)
```

```
(rule (:j (jealous :x))
      (rule (:l1 (loves :x :y))
            (rule (:l2 (loves :z :y))
                  (if (not (equal :x :z))
```

```
                    (assert (kills :x :z) (jealousy :j :l1 :l2))))))
```

This rule embodies the knowledge that jealous people tend to react unpleasantly against others loving the object of their jealousy. The conditional of the rule body ensures that jealousy is not self-applicable.

```
(rule (:l1 (loves :x :y))
      (rule (:l2 (loves :y :z))
            (if (not (equal :x :z))
                (assert (kills :x :x) (unrequited-love :l1 :l2)))))
```

This rule expresses the depression and consequent action resulting from unrequited love.

The final rule provides the means by which the happy nature of this comedy is ensured. This is accomplished by watching for killings, and a statement of contradiction implying that the set of assumptions about the loves of the characters which lead to such a tragedy must be changed.

```
(rule (:k (kills :x :y))
      (assert (tragedy :k) (contradiction :k)))
```

With these assertions and rules we begin the analysis of the conflicts between the

desires of the four lovers. For this example, we will choose an order for applying the rules to matching assertions which provides for maximal entertainment.

The first derived assertion notes the conflict caused by Lysander's jealousy.

F-10 (KILLS LYSANDER DEMITRIUS) (JEALOUSY F-9 F-8 F-5)

This, however, is noticed to be a tragedy, and so ruled out as a happy state of affairs.

F-11 (TRAGEDY F-10) (CONTRADICTION F-10)

The reaction of the system to contradictions is the invocation of dependency-directed backtracking. This process begins by examining the reasons for the contradiction in order to locate the inconsistent set of assumptions underlying the contradiction. In this case, the contradiction F-11 depends upon F-10, which in turn depends upon F-9, F-8, and F-5. F-8 and F-5 are recognized as assumptions by the system, since the reasons for their beliefs include the lack of belief in the assertions F-7 and F-4 respectively. Beliefs supported by a lack of knowledge in other assertions are suspect, since an inconsistency can be interpreted as indicating that some unbelieved assertion must be believed. Thus the backtracking system will use the support for the contradiction to justify belief in one of these unbelieved facts.

Note at this point one of the efficiencies of dependency-directed backtracking relative to the traditional chronological backtracking schemes. In the above, the set of inconsistent assumptions underlying the contradiction is a subset of all extant assumptions. (I neglected to mention the assumed loves of Theseus, Hippolyta, Oberon, Titania, Bottom, Pyramus and Thisby, which may have been determined after the current choices for Lysander and Demitrius.) Thus where chronological systems for choosing alternatives might search through sets of choices involving these independent assumptions, the dependency-directed system will only consider those assumption actually affecting the discovered contradiction.

The next step in the backtracking procedure is the creation of a nogood, an assertion summarizing the support for the contradiction which is independent of the inconsistent set of assumptions.

```
F-12 (NOGOOD F-11) (CP F-11 (F-8 F-5))
```

This statement of independent support is made by means of a conditional proof justification, stating that F-12 should be believed if when F-8 and F-5 are believed, so is F-11. In the present situation, this reduces to the question of belief in F-9, for all that is necessary to believe in Demitrius' murder is the jealousy of Lysander. In effect, then, belief in F-12 is supported solely by belief in F-9.

To remove the possibility of simultaneously holding beliefs in the inconsistent set of assumptions, the nogood is used to justify beliefs in the previously unbelieved assertions underlying these assumptions. All that is necessary, in general, to accomplish this is to justify belief in one of the unbelieved assertions supporting one of the assumptions. However, to prevent future recurrences of belief in the contradiction due to believing a set of assumptions including the current set of inconsistent assumptions, it is necessary to set up new justifications providing for belief in any of the unbelieved assertions underlying the assumptions, so that the knowledge of the inconsistency will be preserved even if the wrong assumption is temporarily retracted.

```
F-7 (NOT (LOVES LYSANDER HERMIA)) (NOGOOD F-12 F-5)
TRUTH MAINTENANCE PROCESSING DUE TO F-7.
F-4 (NOT (LOVES DEMITRIUS HERMIA)) (NOGOOD F-12 F-8)
```

Note that truth maintenance occurred after the new support for belief in F-7, since this change in belief affected F-8, in which belief depended on a lack of belief in F-7. The invocation of truth maintenance affected only those beliefs determined from the changed belief, namely F-7, F-8, F-10, and F-11. All other facts are known, by means of the recorded dependencies, to be independent of these changes. Following truth maintenance, F-7 is believed, and F-8, F-10, and F-11 are unbelieved. Because of this, the next justification made by the backtracking system, that of F-4 via F-12 and F-8, fails to support belief in F-4 due to the lack of belief in F-8, and so does not cause F-4 to be

believed.

The backtracking concluded, we continue our analysis of the consequences of the rules. The next focus for attention is Lysander's subsequent change of lover, due to the retraction of his love for Hermia, and then the condition of poor Hermia, who has now lost her love and is despondent.

```
F-13 (LOVES LYSANDER HELENA) (LOVE-IN-IDLENESS F-7)
F-14 (KILLS HERMIA HERMIA) (UNREQUITED-LOVE F-1 F-13)
F-15 (TRAGEDY F-14) (CONTRADICTION F-14)
```

Another bout of backtracking is invoked. This time, tracing backwards from the contradiction finds, after passing by F-14, F-13, F-1, F-7, and F-12, only the assumption F-5 of Demitrius' love for Hermia. The nogood mechanism then forces the retraction of this assumption.

```
F-16 (NOGOOD F-15) (CP F-15 (F-5))
F-4 (NOT (LOVES DEMITRIUS HERMIA)) (NOGOOD F-16)
TRUTH MAINTENANCE PROCESSING INVOKED BY F-4.
```

In this situation, the support of the nogood consists of the beliefs F-1 and F-12. This invocation of truth maintenance involves checking the beliefs in F-4, F-5, F-7, F-8, F-13,

F-14, and F-15. The supporting of belief in F-4 now removes the reason for the retraction of Lysander's love for Hermia, and so truth maintenance determines that F-4 and F-8, are believed, and that F-5, F-7, F-13, F-14, and F-15 are not.

The end of the example comes as now Demitrius has seen the error of his ways, with a little help from the backtracking system.

F-17 (LOVES DEMITRIUS HELENA) (QUALITY-NOT-QUANTITY F-4)

This satisfies Helena's love, and since Hermia and Lysander are now happy also by means of truth maintenance, all is well.

II. Truth Maintenance Systems Applied

A. Historical Introduction

Some mistakes we must carry with us.

Larry Niven, *Ringworld*

Problem solving programs have traditionally been plagued by blowups in the size of the computations involved in producing solutions. Such blowups were most clearly visible in the early general problem solving efforts, in which a lack of careful control over generated information required large fruitless searches. The problems inherent in such approaches were so great that a new approach to problem solving was developed, that of procedural knowledge [Moses 1967, Winograd 1972, Hewitt 1972, Sussman 1975, Goldstein 1974]. This approach called for formalizing the knowledge of the domain as a set of programs, in which the uses of the basic knowledge were automatic and compiled. This approach proved some major achivements, notably the SIN symbolic integration program of Moses [1967], the SHRDLU natural language understanding program of Winograd [1972], and Sussman's [1975] HACKER.

The PLANNER problem solving language [Hewitt 1972] was developed as a part of the growing awareness of the power of the procedural knowledge approach. PLANNER (and its implemented subset, MICRO-PLANNER [Sussman, Winograd and Charniak 1971]) embodied a data and control structure which abstracted those employed in previous heuristic problem solvers like SAINT [Slagle 1963] and GPS [Ernst and Newell 1969]. PLANNER provided consequent theorems for specifying methods for proposing solutions and answering questions, antecedent and erasing theorems for deriving and maintaining data, and a pervasive system of automatic backtracking. The automatic backtracking of PLANNER caused many problems, however, making the language unusable for sophisticated problem solving, and soon drew much criticism [Sussman and McDermott 1972].

A principal reason for these problems was the chronological nature of control and backtracking in PLANNER. PLANNER kept track of the chronological order of all choices made. This produced several difficulties. First, the list of failpoints was inaccessible to the programmer, which meant that the only means available for dealing with untenable situations was to FAIL, causing the chronologically most recent decision to be changed and all subsequent computations to be discarded. This was bad because PLANNER assumed that each choice and action possibly affected all subsequent choices and actions. Since this is usually false, this assumption resulted in the discarding of much useful, independently derived information. It is this assumption of extreme chronology which leads (quite literally) to most of the failings of PLANNER.

The CONNIVER language [McDermott and Sussman 1974] was developed to alleviate some of these problems of PLANNER, primarily by separating the control state from the data base. Unfortunately, CONNIVER provided no theory of standard problem solving mechanisms. In particular, the typical uses of contexts (including those of QA4 [Rulifson, Derksen and Waldinger 1973] and the similar mechanisms of situational tags [McCarthy and Hayes 1969]) tended to be reminiscent of PLANNER's use of its data bases. Because of this, CONNIVER suffered problems similar to those experienced by PLANNER in that the context mechanism fostered the assumption that all information in a context (and in its subcontexts) depended upon the reason for creating the context. Thus CONNIVER, like PLANNER, provided mechanisms leading to needless discarding of information and an inability to connect failures with the assumptions underlying the failures. CONNIVER, however, did not produce as many anomalous chronological dependencies as did PLANNER, since contexts could be structured into trees. Other advances of CONNIVER over PLANNER were that information could be extracted from a failure, and that other than the most recent choice could be selected as the choice for retraction.

Many of these problems have been overcome or greatly reduced by the innovations introduced in several recent problem solving programs. The EL-ARS electronics circuit analysis system of Stallman and Sussman [1976] automatically records a justification for each assertion in its data base. By operating via a system of rules of a

local nature, ARS factors most of the chronology out of its beliefs, providing a foundation for the determination of current beliefs by means of these justifications, thus reducing the number of anomalous dependencies even more than CONNIVER. Similar but less powerful dependency-directed contexts are also used in Fikes' [1975] data base system, in the data base of McDermott's NASL system [McDermott 1975, 1976], and in Hayes' [1975] planning system. ARS further demonstrates that these same justifications can be used to effect a particularly efficient form of backtracking known as non-chronological, dependency-directed backtracking. This system of backtracking uses the recorded justifications to locate those assumptions relevant to the failure under consideration. It then summarizes and records the infeasibility of the conjunction of these assumptions. These summarizations are consulted whenever making assumptions. This combined process of dependency-direction and summarization permits a great many fruitless assumptions to be avoided.

Another problem encountered in problem solving is the use of multiple representations of information, and in particular, the representation of multiple levels of detail for use in hierarchical systems. In the past, such representations have involved subroutines, with varying evaluation methods, such as depth first subgoaling as in SHRDLU [Winograd 1972], or NOAH-like [Sacerdoti 1975] hierarchical planning, or have involved some variety of redescription mechanism, such as those presented in KRL [Bobrow and Winograd 1976], OWL [Hawkinson 1975], Merlin [Moore and Newell 1973], and [Fahlman 1975]. The integration of dependency-based methods with these mechanisms

introduces a number of problems stemming from the need for hierarchical descriptive mechanisms to separate levels of dependencies appropriate to the levels of description. The truth maintenance mechanisms developed in this report enable this separation by means of a modified form of hypothetical reasoning, which uses conditional proof mechanisms to restructure arguments in ways which preserve the meaning of the justifications and separate relationships at one level from those at other levels.

This chapter details the basic uses and some important applications of truth maintenance systems. The first section presents the methods for representing standard types of knowledge about beliefs, including premisehood, deductive connections, support by conditional proof and the representation of assumptions. This section also describes the use of these basic techniques as tools in representing several useful relationships among beliefs, such as ordered and unordered sets of alternatives and equivalence classes. The next section discusses the use of truth maintenance system structures in hypothetical reasoning. These mechanisms are then used in the important applications of dependency-directed backtracking, generalization and separation of levels of detail. The chapter concludes with a comparison with other current work.

B. Representing Knowledge About Beliefs

The basic components of a truth maintenance system are facts, representations for justifications of belief in facts, and processes for determining beliefs in facts consistent with these justifications. The details of these representations and processes are not critical, and are discussed in Chapter III. This section instead enumerates several basic forms of justifications for belief and their important properties and uses.

Facts are those components of program knowledge which may be invested with belief and justifications. Typically, facts might be attached to assertions, rules and procedures in a programming system data base. It should be realized, however, that different subsystems of a large system can each use the truth maintenance system for different purposes, and can each attach facts to different types of knowledge. For example, a system employing facts attached to assertions and procedures in a PLANNER-like data base might also employ a subsystem which uses its own representations for efficiency, but which also attaches justifications to items in this representation in terms of its own operation and communication with the supersystem.

Facts are not isolated objects, but are connected to each other by dependencies, the relationships of antecedence and consequence in which belief in a fact is related with belief in other facts by means of a justification. Justifications for belief in a fact are predicates of other facts, predicates whose internal structure is accessible to the truth maintenance

system to allow efficient processing and the determination of various dependency relationships. Belief in a fact may or may not be supported by the existence of a valid (that is, evaluating *true*) justification for the fact. If a fact has at least one valid justification for belief, we say the fact is *in*; otherwise the fact is *out*. The distinction between *in* and *out* is not that of *true* and *false*; indeed, there is no imposed notion of falsity in a truth maintenance system. Instead, a support-status of *in* for a fact denotes the existence of knowledge supporting belief in the truth of the fact, and *out* denotes the lack of such knowledge. The function of the truth maintenance system is to mark each fact *in* or *out* in accordance with the validity of its justifications.

The basic types of justifications for belief in facts are premises, deductions, conditional proofs, and assumptions. Premises are created through giving facts justifications corresponding to the constantly *true* predicate. Facts which are premises are thus always *in* independent of any other beliefs. Premises are useful in expressing the basic knowledge of a program, as well as in hypothetical reasoning.

Deductions are justifications in which belief in a fact is expressed as a monotonic function of belief in other facts. Deductions typically express belief in a fact as following from belief in a set of facts. Deductions are thus the most common form of justification in normal computations.

Conditional Proofs are justifications for supporting belief in a fact on the basis

of the derivability of some fact, called the consequent of the conditional proof, from belief in a set of facts, called the hypotheses of the conditional proof. The support implied by the conditional proof justification is the support of the consequent independent of the hypotheses. The most important applications of such justifications are in summarizations, such as in dependency-directed backtracking, in which the support of an inconsistency is noted independent of the set of hypotheses leading to the inconsistency.

Assumptions are deductions based on a lack of knowledge about belief in facts, that is, deductions based in part on some facts being *out*. Assumptions are uses of non-monotonic dependency relationships, in that unlike other types of justifications, assumption justifications can be invalidated by the addition of new beliefs. Assumptions can effectively model the non-monotonic primitives of previous systems, such as MICRO-PLANNER's THNOT [Sussman, Winograd and Charniak 1971], McCarthy and Hayes' [1969] PRESUMABLY, Sandewall's [1972] UNLESS, etc. The dependency-implemented assumptions have the advantage over these systems in that the nature of the assumption is made explicit and accessible in future deductions. In MICRO-PLANNER, for example, THNOTs were indistinguishable from other forms of deductions once performed, and could not be affected by subsequent discoveries of relevant information. In the non-monotonic dependency system, however, assumptions are easily distinguishable from normal deductions and explicitly indicate which beliefs are subject to change when specific beliefs change. For instance, to assume a fact *f true* unless proven otherwise, *f* may be justified by the predicate (OUT *~f*), where *~f* denotes a fact representing the negation of *f*. This

justification will then support belief in f as long as there are no valid reasons for believing $\sim f$.

These basic forms of justifications for belief can be employed to represent several standard relations among beliefs, such as ordered and unordered sets of alternate beliefs, and equivalence class representatives.

Sets of alternatives between which an arbitrary choice is to be made are easily represented by means of assumptions. There are several dimensions along which the representation can be varied, for instance, whether the set is fixed or is dynamically changing, whether the choice of alternatives might be independently decided, and whether a default order for making the choice is desired.

Perhaps the simplest such structure is that effecting a static unordered set of alternatives $\{F_1, \ldots, F_n\}$. If G is the fact representing the reason for the alternative set, at least one of set will be believed if each F_i is provided with the antecedent

(AND (IN G) (OUT $F_1 \ldots F_{i-1} F_{i+1} \ldots F_n$)).

With this structure, the selected alternative is assumed on the lack of belief in the other alternatives, which will lead the backtracking system to justify belief in one of the other alternatives if the selected alternative fails. This structure, however, does not prevent more than one of the F_i from being *in* via independent means of support. To impose this exclusiveness, a contradiction (see section II.D) must be supported by a justification (IN F_i

F_j) for each distinct pair of alternatives F_i, F_j

To allow dynamic additions to the set of alternatives, or to order the alternatives, somewhat more mechanism is necessary. To represent a dynamically extendable, unordered set of alternatives, the following justifications should be implemented. For each possible alternative A_i, two facts, SA_i ("A_i is the selected alternative") and NSA_i ("A_i is not the selected alternative") should be created and justified as follows:

SA_i : (AND (IN A_i) (OUT NSA_i))

NSA_i: (IN SA_j) {for each j distinct from i}

Those processes wishing to reference the selected alternative should then reference whichever SA_i is currently *in*. New alternatives can be added to the set by collecting all existing alternatives and creating the above justifications for the the new selected-alternative fact and for all of the not-selected-alternative facts.

An ordered, extendable (at the end of the order) set of alternatives can be obtained by creating, for each possible alternative A_i, three facts, SA_i ("A_i is the selected alternative"), NSA_i ("A_i is not the selected alternative"), and ROA_i ("A_i is a ruled-out alternative"), and justifications as follows:

SA_i : (AND (IN A_i NSA_1 ... NSA_{i-1}) (OUT ROA_i))

NSA_i: (OUT A_i) , (IN ROA_i)

As before, those processes wishing to reference the selected alternative should reference whichever SA_i is currently *in*. Processes can independently rule out some of the alternatives by justifying the appropriate ruled-out-alternative fact.

Slight extensions of these mechanisms can be used to represent sets of equivalent objects. Justifications can be arranged so that one of the set will be distinguished as the typical member, and this typical member will not depend upon the choice from the set. This can be done as follows. For each possible member, M_i, create two new facts R_i ("M_i is the set representative") and S_i ("M_i is the selected member"), with justifications as follows:

S_i : (AND (IN M_i) (OUT S_1 ... S_{i-1}))

R_i : (CP of S_i relative to (OUT S_1 ... S_{i-1}))

The alternative mechanism selects one of the members M_i as the representative R_i, and the conditional proof justifications for the representatives remove all dependence of the representative on the choice mechanism, that is, all dependencies of S_i other than M_i.

C. Hypothetical Reasoning

Je m'en vais chercher un grand peut-être.

François Rabelais, *Gargantua*

In one form or another, a major use of truth maintenance systems is in hypothetical reasoning, the process of extracting information from the exploration of the consequences of hypotheses. Such extraction is a necessary component of any problem solving system which attempts to limit the extent of its forays into possible solutions, for such systems must be able to decide a train of thought unfruitful, summarize the result of the exploration, and try new hypotheses, while guiding the subsequent effort in light of the information determined in previous attempts. Such abilities require the facility to discuss with one set of beliefs, what would be true if other hypotheses were entertained. One aspect of truth maintenance systems is the provision of mechanisms to support these forms of behavior and reasoning.

The basic mechanisms employed in hypothetical reasoning are the ability to make hypotheses, to summarize results in terms of hypotheses, and to reconcile existing beliefs with new hypotheses. A truth maintenance system fulfills these requirements by providing the ability to make premises and assumptions as discussed above, the ability to summarize results by means of conditional proofs, and the reconciliation of beliefs and hypotheses by means of truth maintenance and backtracking.

Truth maintenance systems do not directly address some of the problems of hypothetical reasoning. There is a large body of research on knowledge-based reasoning concerned with the proposal of hypotheses and differential diagnosis between them. [Winston 1970, Fahlman 1973, McDermott 1974, Rubin 1975, Kuipers 1975, Brown 1976] These issues are beyond the immediate capabilities of truth maintenance systems because they require knowledge of the semantics of facts, and such knowledge is not available to the domain-independent methods described here.

The making of hypotheses is a straightforward application of the justification forms described in the previous section. The balance of this chapter describes the topics of summarization and reconciliation in discussions of backtracking, generalization, and defining levels of detail.

D. Backtracking

> "I should have more faith," he said; "I ought to know by this time that when a fact appears opposed to a long train of deductions it invariably proves to be capable of bearing some other interpretation."
>
> Sir A. C. Doyle, *A Study in Scarlet*

Systems engaging in hypothetical reasoning require mechanisms for reconciling beliefs upon the introduction of new hypotheses. Two types of hypotheses can be distinguished; speculative hypotheses and counterfactual hypotheses. Speculative hypotheses are those which are consistent with existing beliefs and justifications. Speculative hypotheses are useful when a lack of knowledge forces the making of an assumption for the purpose of exploration. Counterfactual hypotheses, on the other hand, contradict previous beliefs. Such hypotheses are useful in exploring the results of actions and in deriving constraints existing in different worlds.

In the context of the computations modelled by the beliefs and justifications of a truth maintenance system, there is an overlap between these two forms of hypotheticals. This overlap results from the orientation of the truth maintenance system towards apparent consistency. Since there is no imposed notion of negation in the belief system, any set of beliefs is considered consistent. It is thus the responsibility of the external system to indicate contradictory sets of beliefs. Thus what may have originally been speculative

hypotheses may later be discovered to be counterfactual hypotheses requiring special treatment. This treatment is called backtracking.

The general reaction necessary to counterfactual hypotheses and the contradictions they induce is to determine the set of hypotheses so discovered to be inconsistent and to resolve the inconsistency by rejecting belief in one or more of these hypotheses until the inconsistency disappears. It is normally desirable to discard as few hypotheses as possible, so the handling of counterfactuals has been characterized as the selection of a maximal consistent subset of the set of inconsistent hypotheses. [Rescher 1964] This process of selection is not amenable to a purely logical solution, since as far as logic and truth maintenance systems are concerned, premises are independent of all other beliefs. This independence means that there are no inherent relations to other beliefs which can be used in discriminating among premises in consistent subset selection.

Unlike premises, assumptions can be related to each other and to other beliefs. Assumptions can be related to the reasons for their introduction and to the specific lack of information allowing their entertainment. Thus if an inconsistency is discovered involving assumptions, the occurence of the inconsistency is information indicating not only that one of the assumptions must be retracted, but also that belief is justified in one of the unbelieved facts whose lack of valid justifications lead, by means of an assumption, to the inconsistency. Because of this, the recognition of an inconsistency can be used to add information which controls the introduction and consideration of further assumptions.

These mechanisms are embedded in truth maintenance systems in two ways. The making of speculative hypotheticals and the necessary reconcilation of these hypotheses with previous justifications for belief are handled by the normal mechanisms of non-monotonic justifications and truth maintenance processing. The process of backtracking becomes in a truth maintenance system the method of dependency-directed backtracking, for the explicit justifications and dependency relationships provide the raw material for the analysis and summarization of the inconsistent set of hypotheses.

There are several steps to the process of dependency-directed backtracking, the first of which is the signalling of an inconsistency by means of a contradiction, a fact justified by the inconsistent beliefs which the external system indicates to the truth maintenance system as a focus for backtracking. All contradictions have the semantics of *false*, so there need be only one such contradiction fact, with new inconsistencies recorded as new justifications for this fact. As far as the truth maintenance system is concerned, however, there can be several representations for *false*, and so several contradiction facts.

The second step of backtracking is the determination of the inconsistent set of hypotheses underlying the contradiction. Since the wisdom of premises and monotonic justifications is inscrutable to the truth maintenance system, the only hypotheses of interest are those based on incomplete knowledge, that is, assumptions.

At this point one of the strong advantages of dependency-directed backtracking over traditional methods is evident, for the hypotheses underlying a contradiction are found by a simple recursive search of the antecedents of the contradiction. Because of this, beliefs in other hypotheses not involved in the support of the contradiction are irrelevant and are ignored in the choosing of a hypothesis for retraction and in the summarization of the causes of the inconsistency. Because irrelevant hypotheses are ignored, a great many sets of possible assumptions are automatically ruled out, thus preventing combinatorial explosions due to independent assumptions. Stallman and Sussman [1976] illustrate this with the example of transistor state-choices in electronic circuit analysis, where traditional methods of backtracking would consider a number of choice sets the product of the sizes of the independent choice-sets. The use of dependency-direction can reduce this to the sum of the sizes of the independent choice-sets. On one six transistor circuit (which would require $3^6 = 729$ states using traditional backtracking methods), this process reduces the number of contradictions to 2 when heuristic orderings are used on alternate choices. The number of contradictions increases to only 13 when the worst choice is made in each decision. [G. J. Sussman, personal communication]

The third step of backtracking is the summarization of the inconsistency of the set of hypotheses underlying the contradiction. This is accomplished by the creation of a nogood, a fact signifying, if {A, B, ... , Z} is the set of inconsistent assumptions, that

$$A \wedge \ldots \wedge Z \supset \mathit{false},$$

or alternatively, that $\sim (A \wedge \ldots \wedge Z)$. By justifying the nogood with a conditional proof of

the contradiction relative to the set of assumptions, the inconsistency of the set of assumptions is recorded as a fact which will be believed even after the contradiction has been disposed of by the retraction of some hypothesis.

The last step of backtracking uses the summarized causes of the contradiction, the nogood, to at one blow retract one of the inconsistent assumptions and to prevent future contradictions for the same reasons. This is accomplished by deriving new justifications which will provide for belief in at least one of the facts whose disbelief enabled one of the assumptions. This step is reminiscient of the justification of results on the basis of the occurrence of contradictions in reasoning by reductio ad absurdum.

The above description has glossed over two points of some interest, the process by which the inconsistent set of assumptions is derived, and the nature of the nogood-based justifications. These topics are discussed below.

When contradiction strikes, the best cure may not be obvious. Although the set of all assumptions supporting the contradiction is easily calculable, recording the inconsistency of this set directly may be inefficient for several reasons. First of all, there is a definite structure, a partial order, relating these assumptions. The partial order is derived from the graph defined by the well-founded support relationship of one fact being an antecedent-fact of another. When assumptions are examined in light of these relationships, some assumptions will be independent of the other assumptions, and some will be

dependent on lower level assumptions, that is, some assumptions will have others among their foundations. Given that the partial order relating assumptions is the only information available to the backtracking system, it makes sense to only consider assumptions which are maximal in this partial order. Subordinate assumptions, if retracted, result in a greater loss of information than is necessary. This seems to be the only clearly worthwhile criterion for selecting an inconsistent subset of hypotheses. Other discriminations are possible, for instance, on the basis of assumption height in the partial order, or on component size in the partial order, but there does not seem to be any clear justification for the use of either of these criteria.

Finally, the occurrence of a contradiction based on a set of hypotheses can be used to construct new justifications for the *out* facts supporting the selected assumptions. Let the inconsistent assumptions be $A_1, \ldots, A_n$. Let $S_{i1}, \ldots S_{ik}$ be the *out* facts among the antecedent-facts of the assumption A_i. To effect the retraction of one of the assumptions, justify S_{ij} with the predicate

(AND (IN *NG* $A_1 \ldots A_{i-1}\, A_{i+1} \ldots A_n$) (OUT $S_{i1} \ldots S_{i,j-1}\, S_{i,j+1} \ldots S_{ik}$)).

This will ensure that the justification supporting A_i by means of this set of *out* facts will no longer be valid whenever the nogood, *NG*, and the other assumptions are believed. This process is repeated for each assumption in the inconsistent set. If the assumptions and the contradiction, are still believed following this, the backtracking process is repeated.

E. Generalization and Levels of Detail

The lunatic, the lover, and the poet,

Are of imagination all compact.

William Shakespeare, *A Midsummer Night's Dream*

An important characteristic of hypothetical reasoning is that results can be named in terms of hypotheses through the mechanism of conditional proof. Such a naming has two important applications in problem solving, those in generalization and in defining levels of detail. Techniques for generalization are derived by discarding the names of results once summarized, and levels of detail are defined by using the supporting hypotheses as abbreviated names for the computations producing the results.

One major technique of problem solving is to generalize from the solution to a particular instance of a problem to a solution of the problem itself. This technique is typically useful when a result holding for a range of entities is desired, but computations can be performed only on specific entities. In such cases, conditional proofs may be used to support the general result in terms of the justification of the specific result independent of the specific entity used. For example, the EL electronic circuit analysis program requires specific (numerical) input voltages to calculate circuit gains, although the computed gain is valid over the entire linear region of the device. To compute the gain for all input voltages, it suffices to compute the gain using a typical numeric value of the input voltage,

and justify the result on the basis of the support of this particular computed gain independent of the specific input voltage used. (This neglects the problems introduced of the dependence of the computation on inequalities.)

This technique of generalization, however, is a special case of a powerful method for separating levels of detail. This method uses the naming function of conditional proofs to support results in terms of the methods uses to compute them (or, as in generalization, independent of the methods used to compute them), thus summarizing the lower level of detail by the name of the method which produced it. This application of hypothetical reasoning is critically important in hierarchical systems employing truth maintenance, as otherwise explanations of results involve all the detailed results at all levels of detail used in computing the result. This not only produces incredibly long, incomprehensible arguments, but also degrades the effectiveness of backtracking. In many cases, a result may be computed by any of several alternate methods. If levels of detail are not separated, involvement of the result in an inconsistency will cause the choice of method to be suspect, and subject to retraction, even if the result is known to be independent of the choice of the method by which it was computed.

The solution to these problems is to represent objects with substructure as follows. The object is separated into its type, its implementation choice, and a boundary which maps connections to the object to connections to the substructures. To separate the operation of the substructure from that of the object, the boundary maps transfer information from the

outside to the interior, adding the facts representing the object and its implementation choice to the dependencies of this information. The boundary maps transfer information from the inside to the outside by means of a conditional proof. Specifically, the information relevant to the outside boundary point is justified by the conditional proof of the information at the inside boundary point relative to the implementation choice. By this means, the only dependencies possible in the resulting outside information is in terms of the object, the choice of its implementation, and the information at the outside boundary points. All dependencies from the internal structure of the object have been removed.

By this scheme, contradictions involving the outside boundary points of the object can only cause backtracking through the implementation choice (if it is in fact an assumption, or depends upon an assumption), and can never penetrate to the inside of the structure of the object. Contradictions discovered by the internal structure of the object, however, can propagate outwards, possibly transferring the inconsistency to the first level at which an assumption is present.

The following example demonstrates the use of this technique in a hierarchical electronic circuit analysis program patterned after the EL program [Sussman and Stallman 1975]. This new program is currently under development by G. J. Sussman, G. L. Steele Jr., M. Matz, and myself. The example has been simplified somewhat, but preserves the pertinent aspects of the program's operation.

The basis of the example will be the conservation of current by a resistor. Resistors are modelled as devices with two constraints overlayed in parallel on the basic resistor device, the constraints of 2-terminalness (conservation of current) and ohms-lawness.

```
(rule (:t (type :r resistor))
      (assert (type :r 2-terminal) (overlay :t))
      (assert (type :r ohms-law) (overlay :t)))
```

In this example we will ignore the ohms-law part of the resistor. The implementation of the 2-terminal constraint is in terms of a lower level of detail, in which equations are implemented as adder boxes and multiplier boxes connected together. The point of this example will be to show how although the arithmetic box level of devices is used to compute and propagate numerical constraints, the nature of this computation is uninteresting from the view of higher levels dealing with electrical laws and devices, and that the mechanism of conditional proof can be used to isolate the higher level of detail from the lower.

```
(rule (:t (type :d 2-terminal))
      (assert (implementation :d (arithmetic-boxes 2-terminalness)) (method :t))
      (rule (:impl (implementation :d (arithmetic-boxes 2-terminalness)))
            (assert (type (kcl :d) adder) (part :t :impl))
            (assert (map (current (#1 :d)) (a1 (kcl :d)))
                    (make-map :t :impl))
            (assert (map (current (#2 :d)) (a2 (kcl :d)))
                    (make-map :t :impl))
            (assert (value (sum (kcl :d)) !0r)
                    (implementation :t :impl))))
```

Here the 2-terminalness is implemented as an adder box. The constraint that the currents of the terminals of the resistor sum to zero is effected by mapping each of the currents to an addend of the adder box, and declaring the sum of the adder to be zero. These maps will serve as the boundary separating the electrical level of detail, the currents on resistor terminals, from the arithmetical level of detail, the connections of adder boxes.

```
(rule (:m (map :var1 :var2))
      (rule (:f (value :var1 :val))
            (assert (value :var2 :val) (inmap :f :m)))
      (rule (:f (value :var2 :val))
            (assert (value :var1 :val) (outmap :f :m))))
```

This rule sets up the channels by which pieces of information, in this case value assertions, are transmitted across the map boundary. The inmap justification does nothing special; it merely transmits the value across, adding the justification of the map in passage. The OUTMAP justification is the means by which the mapping level is separated from the lower level. The OUTMAP operates by picking up the implementation fact which the MAKE-MAP justification (used in the previous rules) has attached to the map fact. The transmitted value is then asserted on the higher level by means of the conditional proof of the lower level value fact relative to the implementation fact. Since both the mapping structure and the lower level internal structure depend upon this implementation fact, they are absent from the resulting explanations.

As a particular example, we demonstrate the passage of current through a resistor.

```
(assert (type r1 resistor) (premise))
F-1 (TYPE R1 RESISTOR) (PREMISE)
F-2 (TYPE R1 2-TERMINAL) (OVERLAY F-1)
```

F-3 (TYPE R1 OHMS-LAW) (OVERLAY F-1)

This defines a particular resistor, R1. The next level of detail is then implemented.

F-4 (IMPLEMENTATION R1 (ARITHEMTIC-BOXES 2-TERMINALNESS)) (METHOD F-2)

F-5 (TYPE (KCL R1) ADDER) (PART F-2 F-4)

F-6 (MAP (CURRENT (#1 R1)) (A1 (KCL R1))) (MAKE-MAP F-2 F-4)

F-7 (MAP (CURRENT (#2 R1)) (A2 (KCL R1))) (MAKE-MAP F-2 F-4)

F-8 (VALUE (SUM (KCL R1)) 0) (IMPLEMENTATION F-2 F-4)

With the wiring of the resistor completed (the wiring of ohms-law and of the adder have been omitted for brevity), we can specify the current on one side of the resistor and examine the resulting explanations.

(assert (value (current (#1 r1)) 7) (premise))

F-9 (VALUE (CURRENT (#1 R1)) 7) (PREMISE)

F-10 (VALUE (A1 (KCL R1)) 7) (INMAP F-9 F-6)

F-11 (VALUE (A2 (KCL R1)) -7) (SUBTRACTION F-5 F-10 F-8)

F-12 (VALUE (CURRENT (#2 R1)) -7) (OUTMAP F-11 F-6)

A query for the explanation of this last fact produces the following result, in which the level of arithmetic detail is absent by means of the OUTMAP conditional proof of the

computed value, as given in F-11, relative to the implementation fact F-4. The conditional proof is used to derive the set of independently supporting beliefs for the computed value, and these are the facts used in the explanation. (The mechanisms for handling conditional proofs and deriving sets of supporting beliefs from them are described in Chapter III.)

```
(explain 'f-12)

PROOF OF F-12 = (VALUE (CURRENT (#2 R1)) -7) (OUTMAP F-9 F-6)

F-9 (VALUE (CURRENT (#1 R1)) 7) (PREMISE)
F-2 (TYPE R1 2-TERMINAL) (OVERLAY F-1)
F-1 (TYPE R1 RESISTOR) (PREMISE)
```

This example has indicated the usefulness of conditional proofs in clarifying explanations by separating levels of detail. More important benefits are possible in improving the information examined by backtracking systems. Just as concise explanations are more useful to humans, improved explanation structures relating beliefs can ease the task of dependency-directed backtracking, both by reducing the number of fact involved in the support of a contradiction, an in isolating levels of assumptions reflecting the levels of detail in the support of the contradiction. These techniques are still being developed, however.

F. Comparison With Other Current Work

There are three major areas of research relevant to the current effort, the representation of knowledge about belief, hypothetical reasoning, and levels of detail, which we discuss in turn.

Representing knowledge about belief has never been a very strong point in AI programs. Many data base systems dealing with incomplete knowledge have adopted the simple strategy of using three states of knowledge about a particular assertion. In these schemes, an assertion can be explicitly stated to be true or false in the data base, with an absence from the data base indicating an unknown state of knowledge. This scheme is sufficient without the linking of beliefs to their justifications. Our non-monotonic dependency scheme is perhaps the simplest extension of previous systems (particularly the ARS system) which effectively allows beliefs to be explicitly based on a lack of knowledge, that is, upon an unknown truth value.

Perhaps the most sophisticated AI effort at understanding the nature of beliefs was that of McDermott [1974]. Although his TOPLE program used little more than the three-state belief system explicitly, it made heavy use of CONNIVER programs to decide which beliefs could be plausibly held and to construct complicated structures describing the reasons for the necessity of certain sets of beliefs. This might be interpreted as the use of learned, pattern indexed nogood sets, used to organize beliefs to avoid standard types of

contradictions and to construct the necessary beliefs. Unfortunately, CONNIVER programs are not inspectable for causes of failures as are explicitly recorded justifications for belief. In addition to efforts in artificial intelligence, there is also a substantial literature in philosophical logic about the logics of knowledge and belief. Some of the systems, such as that of Hintikka [1962], would seem sufficient to define the semantics of *in* and *out*.

Hypothetical reasoning has also been the source of much work in philosophical logic. Rescher's [1964] classic monograph on the subject develops a theory of hypothetical reasoning involving modal categories of beliefs. Such categories are intended to describe the "strength of attachment" of the reasoner to premises involved in inconsistencies. Modal categories are used to discriminate "preferred" maximal consistent subsets of sets of inconsistent premises, that is, subsets which are reduced by discarding as few of the weakest premises as possible to achieve consistency. Truth maintenance systems provide, of course, the ability to determine those premises underlying a particular belief, but do not possess the (domain specific) knowledge of the semantics of facts required to rank these premises in the event of a contradiction. Assumptions, on the other hand, do admit a measure of "fundamentalness" which the backtracking system utilizes. This is the measure derived from the partial order among assumptions, which interprets the partial order as indicating that non-maximal assumptions are more fundamental than maximal assumptions. This measure is used to retract only the maximal assumptions to remove inconsistencies. This is perhaps analogous to a dynamically computed set of modal categories for ranking the strength of beliefs.

Hypothetical reasoning has been approached from several directions in artificial intelligence research. The earliest approach was that of some mechanical theorem provers, in which no backtracking occurred. The more traditional approach is that of domain-independent automatic backtracking. This form of hypothetical reasoning includes blind MICRO-PLANNER chronological backtracking and several more careful methods, such as the unification-tree method of Cox and Pietrzykowski [1976], memoizing search [Greenblatt, Eastlake and Crocker 1967], and dependency-directed backtracking, which is perhaps the most sophisticated semantics-free automatic backtracking method possible.

An alternative approach to hypothetical reasoning seems to underlie the paradigm advanced by Minsky [1974] for problem solving:

> "The primary purpose in problem solving should be better to understand the problem space, to find representations within which the problems are easier to solve. The purpose of search is to get information for this reformulation, not - as is usually assumed - to find solutions; once the space is adequately understood, solutions to problems will more easily be found."

Several recent efforts have developed techniques with this flavor, including the "debugging" approach of gaining information through failures, applied with some success in salvaging information from hypothetical situations. [Fahlman 1974, Sussman 1973, Goldstein 1974] Other efforts, such as Winston [1970], McDermott [1974], Kuipers [1975], and Rubin [1975], have focused on using information in unsucessful hypothetical explorations in

differential diagnosis between hypotheses.

There has also been a considerable body of research on developing hierarchical structures for representing knowledge. Much of this has focused on recursively embeddable structures, such as frames [Minsky 1974]. The primary concern of many of these efforts, however, seems to be in mainly on the problems of multiple description and recognition, and little effort seems to have been devoted to elaborating mechanisms for controlling reasoning in systems with multiple levels of detail. [Zdonik, Sjoberg, and Matz 1976] With the advent of dependency-based reasoning systems, it has become clear that the interaction of control and description is an important consideration. A major component of this problem is that of structuring explanations to reflect the hierarchy of the computation, so that each level is explained in terms of itself and higher levels, with indicators of the structures at lower levels.

III. Truth Maintenance Mechanisms

A. Historical Introduction

The earliest recording of dependency information by AI programs is found in mechanical theorem provers, in which justifications for beliefs were recorded in the form of proofs. Unfortunately, the traditional preoccupation with consistency of theorem prover designers seems to have led these efforts to ignore any possible uses of the recorded proofs. Since beliefs can never become false in their monotonic systems, no attention was devoted to methods for judging belief rather than truth. The major use of the recorded proofs in theorem proving programs has been in the use of records of variable unifications as answer substitutions [Green 1969]. Recently, somewhat more sophisticated uses of these substitution histories have been used in improving theorem proving backtracking techniques [Cox and Pietrzykowski 1976], but these still suffer from the tree-structured environments produced by unifications.

Non-theorem proving automatic recording of dependency information is a rather recent innovation. Systems such as those of Hayes [1975], Fikes [1975], and McDermott [1975] recorded simple forms of dependency information, but made no sophisticated use of these recorded dependencies beyond a simple scheme for erasing data. The first sophisticated use and recording of dependency information seems to have been in the ARS system of

Stallman and Sussman [1976], in which the method of dependency-directed backtracking was introduced. This system was also the first to realize the importance of multiple justifications, well-founded support and circularities, and the use of primitive truth maintenance. All of these systems will be discussed in more detail at the close of this chapter.

This chapter presents the details of the representation for knowledge about beliefs. The concepts of facts, justifications, *in* and *out* support-statuses, negatives, antecedents, antecedent-sets, and consequences are introduced. Next, the question of well-founded support for belief is discussed, and the related concepts of supporting-antecedent, antecedent-facts, supporting-facts, affected-consequences, foundations, ancestors, and repercussions are introduced. With this structure for reasons for belief, the process of truth maintenance is described. This is the process whereby beliefs are redetermined in circumstances of gaining or losing information about reasons for beliefs. Some considerations involved in the efficiency of this process are discussed, and the chapter concludes with a comparison of this approach to mechanisms employing contexts or situational tags, and to other dependency-based systems.

B. Facts and Dependencies

> And there is no object so soft but it makes
> a hub for the wheel'd universe.
>
> Walt Whitman, *Song of Myself*

The basic components of a truth maintenance system are facts, dependencies connecting the facts, and processes for determining well-founded support for all beliefs in facts. As described in Section II.B, facts are those components of program knowledge which may be invested with belief and justifications.

Facts are not isolated objects, but are connected to each other by dependencies, the relationships of antecedence and consequence in which belief in a fact is related with belief in other facts by means of a justification. Justifications for belief in a fact have a simple structure discussed below, but typically amount to sets of other facts such that belief in the fact is justified if all the facts in at least one of these sets are believed. Belief in a fact may or may not be supported by virtue of its justifications. If a fact is believed to be true by way of its justifications, we say the fact is *in*, otherwise the fact is *out*. The distinction between *in* and *out* is not that of *true* and *false*: rather the former represent the presence or absence of knowledge supporting belief, and the latter actual truth value. For this reason, all facts are intially considered *out*; that is, with no justifications, no beliefs are justified. The *in*ness or *out*ness of a fact is also termed the fact's support-status. IN and OUT are

predicates of facts which are true if and only if their arguments are respectively *in* or *out*.

Justifications for belief in a fact are embodied in the antecedent-set of the fact, a set of antecedents. Each antecedent provides one possible justification for belief in the fact. Recording multiple justifications is important, as it both saves work when hypotheses change, and avoids certain problems (discussed in detail in the following sections) relating to circularities in justifications for beliefs. Because of these circularities, some schemes for retaining a single justification for each belief can lead to permitting unjustified beliefs in some facts.

Two formats for the strucure of antecedents provide simple descriptions of most forms of justifications. These two forms are called support-list antecedents (SL-antecedents) and conditional-proof antecedents (CP-antecedents). SL-antecedents are just a pair of lists, called the *in*list and *out*list, and are interpreted (and will be written) as the predicate (AND (IN *in*list) (OUT *out*list)). SL-antecedents are valid if each fact of the *in*list is *in*, and each fact of the *out*list is *out*. CP-antecedents consist of a fact (the consequent of the conditional proof), and two lists of facts, called the *in*moduli and the *out*moduli. CP-antecedents will be written as the predicate (CP consequent *in*moduli *out*moduli), and are valid if consequent is *in* whenever each *in*moduli is *in* and each *out*moduli is *out*.

Premises, then, are simply SL-antecedents with null *in*lists and *out*lists. Deductions are SL-antecedents with null *out*lists, assumptions are SL-antecedents with non-

null *out*lists, and conditional proof are represented by CP-antecedents in the obvious fashion (normally with null *out*moduli).

The antecedents of facts are also used to construct other sets of dependency relationships. The consequences of a fact are those facts such that the fact occurs in an antecedent of each consequence. The CP-consequent-list of a fact is a list of all those facts possessing a CP-antecedent with the given fact as the consequent of the conditional proof antecedent.

C. Well-Founded Support Relations

Stallman and Sussman [1976] make the important observation that a supporting-antecedent must be distinguished in the antecedent-set of each *in* fact, such that the supporting-antecedent determines a proof of belief in the fact in terms of other facts with well-founded (non-circular) support. This requirement of well-founded support is necessary because circularities in dependency relationships arise in a large variety of circumstances, particularly in situations involving simultaneous constraints on a number of objects. If not handled properly, these circularities lead to unjustified beliefs which both allow unnecessary computations to be performed and confuse dependency-examining processes like backtracking.

Consider, for example, the situation in which the fact *f* represents (= (+ *x y*) 4), *g* represents (= *x* 1), and *h* represents (= *y* 3). If *f* is believed and *g* is supported, it is reasonable to assume that some rule will justify belief in *h* with the antecedent (AND (IN *f g*) (OUT)). This will cause *h* to become *in*. If the support for *g* is now retracted, due to changing hypotheses, and if *h* becomes independently supported, the rule will then justify *g* with the antecedent (AND (IN *f h*) (OUT)). If the independent support for *h* is then retracted, the incautious strategy of reevaluating antecedent-sets one at a time would declare both *g* and *h* as *in*, since their justifications are mutually satisfactory. By distinguishing well-founded support, the situations requiring careful examination of support are highlighted.

One might observe that the supporting-antecedent of a fact could be generalized to a set of antecedents, each of which provides well-founded support for belief in the fact. This, however, only causes unnecessary work. Since the fact will be believed as long as there is at least one supporting-antecedent, there is no point in propagating involvment in truth maintenance processing through the fact until the last supporting-antecedent is invalidated. Recording multiple supporting-antecedents fails to discriminate in cases where only one supporting-antecedent becomes invalid but others are invariant.

To avoid doing more work than necessary, then, only one supporting-antecedent should be distinguished. Furthermore, if several candidates present themselves as providing well-founded support, the best choice is that antecedent which will be valid the longest. This problem, like that of memory paging, is intractable in general, and so heuristics should be employed to guess the best antecedent to use. One simple heuristic, the one used in the programs of Appendix 3, is to choose the chronologically oldest of the possible antecedents, on the theory that chronologically older justifications are likely to be more "fundamental" in some sense, and so are less suceptible to change. Another device is to employ a self-organizing heuristic [Rivest 1976] to order the antecedent-set of the fact. The current research has not included any experimentation to see if benefits accrue from the use of these heuristics, or to see which provides the better performance. One might also imagine schemes by which facts had "certainty factors" attached, with the likelihood of a justification being computed from the certainty factors of the mentioned facts. Except for

those using actual probabilities, however, such schemes have had a reputation for questionable semantics, and so have not been investigated in this research.

For the purpose of tracing through justifications, it is convenient to extract another dependency relationship from the supporting-antecedents of facts. The antecedent-facts of a fact are those facts which currently support belief in the fact. Thus an *out* fact has no antecedent-facts, and the antecedent-facts of an *in* fact are just those in the (necessarily disjoint) union of the *in*list and *out*list of the fact's supporting-antecedent. (As will be explained in more detail later, CP-antecedents never are the supporting-antecedent of facts. Instead, they are used to generate new SL-antecedents summarizing the independent support of the conditional proof, and these SL-antecedents are used instead.)

The antecedent-facts of a fact are not all that is required, however, since when beliefs change the important question is not just which of the believed facts depended on the changed beliefs, but also which other facts might now be believed by virtue of the changes. To make this determination efficient, the set of facts affecting the current support-status of each fact is collected into the fact's set of supporting-facts. The supporting-facts are the same as the antecedent-facts for *in* facts, since any change in one of the antecedent-facts of a fact may cause the fact to become *out*. The supporting-facts of an *out* fact consists of one fact from each antecedent of the fact, such that each selected fact implies the *false* evaluation of the corresponding antecedent. This means that each of the *out* supporting-facts of an *out* fact is in the *in*list of one of the fact's antecedents, and each

in supporting-fact is in the *out*list of some antecedent. While this definition does not specify a unique set of facts as the supporting-facts of an *out* fact, any such set suffices, for the intent is for the supporting-facts of a fact to be a small set of facts which, if unperturbed, forces belief in the supported fact to be invariant. This definition allows, in odd cases, inclusion into the supporting-facts of a fact some facts upon which the support-status of the fact does not really depend. For example, the silly antecedent (AND (IN *f*) (OUT *f*)) might cause *f* to be included in the supporting-facts, even though the antecedent is constantly false independent of the support-status of *f*. Note also that the relationship of one fact being a supporting-fact of another is not well-founded.

From the supporting-facts of a fact, we define additional concepts as follows. The affected-consequences of a fact are those facts whose current support-status rests on the fact; precisely, the affected-consequences of a fact are those consequences of the fact such that the fact is a supporting-fact of each of these consequences. The foundations of a fact are those facts involved in the well-founded support for belief in the fact; precisely, the foundations of a fact are the transitive closure of the antecedent-facts of the fact under the operation of taking antecedent-facts. The ancestors of a fact are those facts involved at some level in determining the current support-status of the fact; precisely, the ancestors of a fact are the transitive closure of the set of supporting-facts of the fact under the operation of taking supporting-facts. The repercussions of a fact are those other facts whose support-statuses are affected at some level by the support-status of the fact; precisely, the repercussions of a fact are the transitive closure of the affected-consequences of the fact

under the operation of taking affected-consequences.

D. Truth Maintenance

And we'll talk of them too,
Who loses, who wins, who's in, who's out,
And take upon's the mystery of things.

William Shakespeare, *King Lear*

Truth maintenance is a process invoked whenever the support-status of facts change, and consists of redetermination of support-statuses for the facts and their repercussions, possibly including backtracking in the event of recalling contradictions. The processing involves an examination of the affected facts for well-founded support. The presence of various types of circularities in the dependency structure complicates this however, requiring a more elaborate mechanism than a simple bottom-up support analysis.

Facts can change their support-status in two ways. The normal reason for change is tht addition of a new valid antecedent to an *out* fact signalling a change of support-status from *out* to *in*. The other way that change can occur is if the justification of a premise is retracted.

New justifications supplied by the external program to the truth maintenance

system require differing actions depending upon the validity of the new justification and the support-status of the justified fact. An antecedent added to an *in* fact only requires the augmentation of the antecedent-set of the fact by the new antecedent, for the new antecedent cannot cause a change of support or support-status. A new antecedent added to the antecedent-set of an *out* fact requires truth maintenance processing on the fact if the new antecedent is valid, for this circumstance signals a change of beliefs. A non-valid antecedent, however, can be added to the antecedent-set without causing truth maintenance. Such a new antecedent requires the additional step of updating the supporting-facts of the fact to include those facts responsible for the invalidity of the new antecedent.

Of course, truth maintenance processing can be trivial in the case of a fact with no affected-consequences, for the support-status of such a fact can be changed without affecting any other beliefs. This special case is important as it routinely occurs after newly created facts are given their first justification.

Alternatively, truth maintenance might be required upon the retraction of a premise. If a fact has been given a premise antecedent, that is, an SL-antecedent with null *in*list and null *out*list, the retraction of this fact as a hypothesis can be effected simply by removing this antecedent from the fact's antecedent-set. If the removed antecedent is not the supporting-antecedent, no further action is necessary. If, however, belief in the fact had actually been supported by the removed antecedent, truth maintenance processing must be invoked.

Truth maintenance processing starts by producing a list comprising the invoking facts and their repercussions, and to mark each fact on this list with a support-status of NIL to denote the state of lacking well-founded support. (Such a support-status exists only during the process of truth maintenance. Except during truth maintenance, all facts have as support-status of either IN or OUT.) This marking is used to ensure that the *in*ness or *out*ness of a fact is based on facts with well-founded support. Next, each of the facts on this list must be examined. This is a recursive procedure taking action only if the fact being examined has a support-status of NIL. The antecedent-set of the fact is evaluated with respect to well-foundedness (as described below), and the process terminates if well-founded support is still not forthcoming. If a well-founded support-status is determined, the support-status of the fact is set appropriately, the supporting-facts installed, and if the fact is now *in*, the supporting-antecedent is installed. Finally, the newly determined status of this fact might now allow determination of the statuses of its consequences, so each of the consequences of the fact is examined in turn.

Evaluation of the well-foundedness of antecedents is done with respect to the three values T, F and NIL. SL-antecedents evaluate to T if each fact of the *in*list is *in* and each fact of the *out*list is *out*; to F if some fact of the *in*list is *out* or some fact of the *out*list is *in*; and to NIL otherwise. CP-antecedents evaluate to T if all *in*moduli are *in*, all *out*moduli are *out*, and the consequent is *in*; to F if the first two conditions hold and the consequent is *out*; and to NIL otherwise. By these evaluation procedures, a fact is

considered *in* if any of its antecedents evaluates to T, *out* if all its antecedents evaluate to F, and otherwise is lacking well-founded support.

The above process will determine well-founded support for the majority of facts, but can leave some facts without well founded support. These are facts which are involved in circularities in the dependency relation, or whose possible support depends on facts involved in circularities. In most cases, such facts would be known to be *out* except for the circular justifications.

There are essentially three different kinds of circularities which can arise. The first and most common is a circularity in which all facts involved can be considered *out*, consistently with their justifications. Such circularities arise routinely through equivalences and simultaneous constraints. An example of this is the circularity presented in the previous section in which an equation produces a circularity between the facts g and h. In this case, however, if neither of g or h is supported by antecedents not involved in the circularity, both g and h should be considered *out*, as this is consistent with their justifications.

The second type of circularity is one in which every assignment of support-statuses to facts consistent with the justifications of the facts must provide support for at least one of the facts involved. For example, such as situation occurs due to the existence of two facts, f and g, such that f has an antecedent of the form (OUT g), and g likewise has

an antecedent of the form (OUT *f*). Certain types of ordered alternative sets can be made to avoid this type of circularity. These circumstances arise typically through the production of unordered alternatives, such as those arising in the backtracking scheme given in section II.D.

The third form of circularity which can arise is the strangest: the unsatisfiable dependency structure, a necessarily circular dependency relation in which no assignment of support-statuses to facts is consistent with their justifications. An example of such is a fact *f* with the antecedent (OUT *f*). This antecedent implies that *f* is *in* if and only if *f* is *out*. Natural examples of unsatisfiable dependency structures are hard to imagine, and are best viewed as bugs in the knowledge or implementation of the problem solver. (It has been my experience that such circularities are most commonly caused by confusing the concepts of *in* and *out* with those of *true* and *false*. For instance, the above example could be produced by this misinterpretation as an attempt to assume the fact *f* by giving it the antecedent (OUT *f*).)

These circularities are handled by the next step of truth maintenance, a special type of examination of the facts not supported during the well-founded support pass of truth maintenance. In this step, the examination procedure still ignores facts possessing a non-NIL support-status. It first checks for well-founded support, as in the previous examination process, and if it finds such support installs it and proceeds to the consequence-processing stage below. If well-founded support is still lacking, however, the

SL-antecedent set is specially evaluated while considering NIL to be equivalent to OUT. That is, the SL-antecedent set is evaluated under the assumption that all unsupported facts are OUT. (The CP-antecedent set is ignored during this evaluation. This will be discussed later.) This evaluation is used to determine the fact *in* or *out*.

Once the examination finds the support-status for the fact, it must check the consequences of the fact. If the fact was determined to be *out*, then a simple recursive examination of the consequences is sufficient. If, however, the fact was brought *in*, more care must be taken. If the fact brought *in* had affected-consequences, then these and their repercussions had been determined on the assumption that the currently examinined fact was *out*, and so must be remarked and reexamined. If the fact had no affected-consequences, then a simple recursive examination of the consequences is sufficient.

This procedure is not completely safe, for it will devolve into an infinite loop if unsatisfiable circularities are present. Such circularities, as previously mentioned, are really erroneous uses of the truth maintenance system. This possibility of an infinite loop can be avoided at the cost of inserting a well-foundedness check whenever a fact is brought *in*. This check operates by checking the ancestors of the *in*ned fact to see if they include the fact itself. If this condition holds, an unsatisfiable dependency structure has been detected.

The above process is incomplete in its treatment of facts justified via conditional proofs. Rather than perform the hypothesizing of beliefs necessary to evaluate the

conditional proof antecedent when the previous independent support of conditional proof vanishes, the above procedure simply ignores the conditional proof antecedent. This is a major incompleteness in the current system, and a problem for exploration and solution by future research.

The current partial solution to this problem is to pass over the examined facts after truth maintenance has decided support-statuses for the lot. If a fact which is the consequent of some conditional proof antecedent is found to be *in*, then new SL-antecedents are derived for the facts possessing the CP-antecedents, where the *in*list and *out*list are derived from a FINDINDEP (see below) of the consequent fact modulo the *in*moduli and the *out*moduli. The fact is then justified with this new antecedent. If this step causes truth maintenance, the scan must be restarted to check for further changed facts.

The procedure for deriving SL-antecedents from CP-antecedents is called FINDINDEP for historical reasons. This is a fairly simple procedure, involving two recursive scans of the foundations of the consequent of a conditional proof. The first step is a recursive search of the foundations of the conditional proof consequent to mark all those facts which are in one of the moduli sets, or are repercussions of one of the moduli. The second step is a similar search, which examines the marks left by the first pass to collect all those foundations of the consequent which are either antecedent-facts of the consequent, or are antecedent-facts of repercussions of the moduli, but which are not themselves repercussions of the moduli. Following this collection, the marks are removed.

The facts collected are separated into a set of *in* facts and a set of *out* facts, and these are then used to create the SL-antecedent summarizing the independent support validating the CP-antecedent.

Another check performed during this scan is that of looking for *in* facts marked as contradictions. If such facts are found, the backtracking mechanisms must be invoked and the scan restarted.

Finally, the truth maintenance system interacts with the external systems through signaling functions attached to each fact. These are simply functions to call on the fact if its support-status changes during truth maintenance. These functions are called signal-recalling functions (*out* to *in*) and signal-forgetting functions (*in* to *out*).

E. Efficiency of Truth Maintenance

> The Mathemagician nodded knowingly and stroked his chin several times. "You'll find," he remarked gently, "that the only thing you can do easily is be wrong, and that's hardly worth the effort."
>
> Norton Juster, *The Phantom Tollbooth*

There are several issues relevant to the efficiency of truth maintenance. Some of these, such as the selection of well-founded support for beliefs, have been previously discussed. This section presents two additional topics, those of possible improvements to the truth maintenance algorithms presented above, and the theoretical difficulty of the general problem of truth maintenance. In the first of these discussions, the issues discussed concern the search procedures employed by the truth maintenance processor in examining facts to determine their well-founded support. The second discussion presents a mapping from a problem related to the NP-complete problem of determining the unsatisfiability of boolean formulas in disjunctive normal form to the problem of choosing support-statuses for facts consistent with their justifications.

Efficient Implementations of Truth Maintenance Systems

As previously mentioned, truth maintenance is an incremental process which

requires examination only of the repercussions of the invoking fact. These repercussions and their dependency structure correspond to a directed graph, and so truth maintenance can be viewed as an algorithm operating on certain types of graphs. As such, many organizational principles and key questions about truth maintenance can be drawn from the known principles for constructing efficient graph algorithms.

There are two basic problems for efficient solution occuring in truth maintenance, the problem of how to determine and examine the repercussions of the invoking fact, and the problem of choosing support-statuses for facts involved in circularities. The first of these problems involves processing a graph with nodes corresponding to the repercussion of the invoking fact and edges corresponding to their dependencies. Actually the repercussions of the invoking fact are not all of the nodes of the graph, since there will in general be facts referenced in the antecedent-sets of the repercussion fact which are not themselves among the repercussions. Since only the repercussions need be examined (as far as evaluating antecedent-sets is concerned), the costs of ignoring these other facts will be neglected.

The basic question to be faced in this process is choosing the order in which the nodes are to be processed. Typically a depth first approach yields the faster algorithm, but here the issue is complicated by the fact that support must be derived from the bottom up, so to speak. If one collects the repercussions of the invoking fact in the correct order, such a bottom up search can be effected. This has been our strategy primarily because it is a

simple one. Other strategies include a breadth first search, which is attractive as it would seem to process the bottommost facts first before moving up to the next level of repercussions. Such a breadth first search is also easy to implement, and it would be interesting to compare the relative efficiency of these approaches in actual use. One further strategy, suggested by the next topic, is discussed below.

The problem of analysis of circularities can also be handled efficiently. The idea here is to actually construct a graph representation of the facts involved in the circularity and the dependencies between them (but excluding those to other facts) and to examine this graph by means of one of the linear-time depth first algorithms for determining strongly connected components of directed graphs. The components so discovered can then be examined in linear time to find the minimal components, followed by examining the facts of the minimal components first. By these steps analysis of circularities can also be done in time proportional to the size of the graph of the circularities.

This method of analyzing circularities suggests another searching strategy for the basic truth maintenance process, that of finding the strongly connected components of the graph of the repercussions of the invoking fact and then sorting these components topologically to find a likely order in which to search the repercussions. This analysis can also be done in time linear in the size of the graph involved which suggest this as a viable alternative to the rather blind methods proposed earlier. Unfortunately two considerations suggest the opposite. First, the process involved here is more complicated, and one might

find simpler algorithms to be faster because of smaller constants being involved. But a more compelling argument follows from the observation that the strongly connected components may involve facts which can be given well-founded support by other antecedents. For many (perhaps most) facts, admitting well-founded support may be independent of involvement in circularities. That is, involvement in circularities is only an approximation to the true state of affairs vis-a-vis well-founded support. The final blow is struck by the empirical observation that circularities seem to occur independent of "level" in the dependency graph, with facts high in the graph being potentially antecedents of facts lower in the graph. Thus it may occur that on the basis of circularities alone, the strategy may well perform the analysis to find all of the repercussions of the invoking fact involved in one massive circularity. Thus the process of analysis of circular dependencies does not seem appropriate to the general truth maintenance strategy, but rather only appears to be useful in considering facts for which well-founded support cannot be determined in any other way.

It would be nice to have a precise theory of how to determine support for facts in the most efficient manner possible, as well as how to deal with circularities as efficiently as possible, but I have no such theory. This question is perhaps worth additional research effort.

The Theoretical Difficulty of Truth Maintenance

As the previous sections have indicated, truth maintenance is an efficient process in general, requiring an effort proportional to the size of the dependency graph involving the invoking fact and its repercussions. This cost does not include the cost of backtracking which may be necessary to choose beliefs consistent with the recorded justifications in those cases in which circularities prevent determination of well-founded support for all facts. The purpose of this section is to relate the theoretical difficulty of truth maintenance to potentially difficult problems.

It is straightforward to reduce questions about the unsatisfiability of boolean formulas in disjunctive normal form to questions about the existence of assignments of support-statuses to facts consistent with their justifications. If this question was of interest, it would be the major part of a proof that the general problem faced by truth maintenance is NP-complete [Cook 1971, Karp 1972], since DNF-UNSAT is well-known to be NP-complete. We have argued, however, that unsatisfiable dependency relationships are manifestations of bugs in the system supplying the justifications for beliefs in facts, and thus are not to be expected in the normal operation of a truth maintenance system. The interesting question is instead the difficulty of finding a consistent assignment of statuses to facts when the existence of such an assignment is guaranteed. I know of no work answering this question, and it would probably be of interest to answer this question.

For completeness, however, we provide the mapping between the structure of

boolean formulas in disjunctive normal form and facts and justifications. Let E be a boolean formula in the variables $X_1, \ldots, X_n$ in disjunctive normal form. We translate the question "Is E unsatisfiable" to the question "Is f *in*" for a fact f with justifications as follows: f will have one justification for each disjunct of E. Each disjunct of E will be translated to a justification of f by letting $f_1, \ldots, f_n$ be auxilliary facts, and letting the justification corresponding to each disjunct be the conjunction of terms of (IN f_i) if $+X_i$ is in the disjunct, and (OUT f_i) if $-X_i$ is in the disjunct. With this translation it is clear that each assignment of statuses to $f_1, \ldots, f_n$ which supports belief in f corresponds in a natural way to an assignment of truth values to $X_1, \ldots, X_n$ which satisfies E. In fact, f can be *in* if and only E is satisfiable.

F. Dependencies and Contexts

Quite a number of important problems with traditional AI data bases can be fruitfully discussed from the standpoint of truth maintenance systems. Many of these problems have traditionally been approached via the mechanisms of contexts and situational tags. The theme of the following discussion is that many difficulties encountered in the use of CONNIVER/QA4 contexts [McDermott and Sussman 1974, Rulifson, Derksen and Waldinger 1973] are simply artifacts of what is in effect an attempt to approximate dependency relationships by means of contexts, and that these problems naturally disappear when the beliefs are determined by means of a truth maintenance system. The relation between the use of a truth maintenance system and the use of situational tags is also considered. The emphasis in this discussion is that situational variables exhibit problems similar to those arising in the use of contexts when used to approximate dependency relationships.

Two classic constructs in artificial intelligence programming languages are antecedent and erasing procedures. Antecedent procedures are pattern invoked procedures triggered by the addition of a matching item of data into the current context; erasing procedures are the complementary form of procedure, invoked whenever a matching datum is removed from the current context. As McDermott [1975] points out, these two classes have found two common uses; as software interrupts, and as what McDermott terms "data-derivers". The use of antecedent and erasing procedures as software interrupts is without

bearing on the current discussion and will be ignored. As data-derivers, however, antecedent procedures have typically been used to produce standardly desired consequences or equivalents of freshly asserted data, while erasing procedures have been used in the related task of cleaning up these derived consequences when the primary data is erased.

Several problems arise in the use of such data-deriving procedures. The first of these is that such data-derivers in general require a duplication of information: essentially the same information is involved in deriving the data as in discarding it. Such forced duplication of information is clearly undesirable, both from the viewpoint of the perspicuity of knowledge representation, and from the standpoint of level of detail. Problem solving program designers have harder problems to deal with than the bookkeeping details of the cleanup of derived data. The necessity of having to explicitly program such a cleanup routine for each data deriver is thus forcing the programmer to deal with an unnecessary level of detail. The use of a truth maintenance system, by which recorded justifications are used to perform such bookkeeping tasks automatically is a natural solution to this problem.

A second problem is that the usual behavior of antecedent procedures is to insert new items into the current context, independent of the contexts containing the antecedent procedure and the triggering data. Because of this placement of derived data, information may be needlessly discarded and rederived if the current context is changed for some reason. Furthermore, since the source of the derived data is not recorded, the program

must rederive the information in each context branch even if the derivation is independent of the context branching. This problem is compounded by the possibility that the antecedent procedure and triggering item are themselves subject to the same misplacement and rederivation problems. Of course, antecedent procedures could be written so as to place their conclusions in the most general context permitted by the contexts of the antecedent procedure and triggering procedure, but this would have to be determined each time the procedure was run and would have to be done in each antecedent procedure. Even such a uniform check, which is best done automatically, is still not the correct solution, as the traditional conception of context is not the correct description of a place into which data may be inserted.

The effect of tree-like contexts is to identify the justification of each item in the context as the sequence of choices which led to that particular context through a sequence of pushing new contexts. The flaw with this is that it produces an approximation to the true dependencies involved. This approximation leads to inefficiencies of the sort described above, which can be avoided only by having each procedure make a careful study to determine exactly where to place each new piece of information and when to remove it. All of these mechanisms are subsumed by the operation of a truth maintenance system, in which new assertions are automatically placed in exactly the correct "context", and are available in any context in which at least one of their justifications is valid.

Another common use of contexts is as a means to implement packets, [Fahlman

1973, McDermott 1975] collections of closely related data and procedures. Packets are normally used as a description of some object or structure, or type of structure, which can be "swapped in" whenever such a structure needs description. Frequently packets are parameterized so that an instantiation of the packet knowledge may be produced for each known instance of the type of structure described by the packet. Such instantiations lead to problems generally labelled as the "symbol mapping" problem, but this is not of interest here. Of interest instead is the problem of context merging which arises when a packet is added to the current collection of knowledge.

The problem of merging contexts can be stated as follows: Take a set of contexts (typically the current context and any packets for inclusion), union their contents, and push a new context onto the result. An innocuous seeming task indeed, except that the desired result is the consistent union of the contents of the merged contexts. Items may be present or cancelled in these contexts, due to deductions or exception specifications. This causes problems when merging contexts in which some item has been cancelled in one context and asserted in another. The context structure alone provides no clues for deciding which items are present and which are not.

If instead contexts are determined by dependencies in a truth maintenance system, the problem becomes clearer: the consistent union of the merged collections of facts is determined by the justifications of the facts involved, and the inconsistencies which remain are the true conflicts requiring explicit attention in any case. In addition, the packet data

can be explicitly dependent upon the identification of the packet itself, thus providing an aid to the separation of levels of detail.

The complementary problem of context excision arises when the need for a packet disappears and the contents of the packet are to be deleted from the current context. A truth maintenance system also performs the desired function here. When the choice leading to the inclusion of the packet is retracted, those facts supported only by this choice are discarded, and those facts with support independent of the choice and packet remain as program beliefs. This solution avoids the problem of discarding all packet information, whether independently useful or not.

The preceding discussions have advanced arguments demonstrating that some failings and difficulties associated with contexts are due to the use of contexts to approximate dependency relationships. The suggested solution is to instead handle dependencies with greater accuracy and to use recorded justifications to determine beliefs. This solution dispenses with those problems that are artifacts of the misuse of contexts, while making the true problem solving issues apparent and subject to attack. We now discuss how these same considerations reflect on one alternative approach to contexts, the use of situational tags. [McCarthy and Hayes 1969]

In current proposals, situational tags are offered as useful mechanisms for creating hypothetical situations, temporal situations, etc. It seems that such proposals are

confusing at least two types of uses of situational tags, those which describe logical dependence and those which distinguish objects. In the former usage, situations suffer the same problems as do contexts.

One set of uses for situational tags is as an alternative method of implementing contexts, in which the context marker is an explicit part of the assertion. For this purpose, situational tags suffer the same failures and difficulties encountered in the use of ordinary contexts. There is little difference in the problems arising when hypotheticals are modelled using contexts or situational tags - the effect is in both cases to approximate the true dependencies by pushing a new context or by creating a new situation. In both cases information must be rederived in other contexts or situations even if the method of derivation is independent of some of the assumptions involved in constructing the context or situation. In such uses of situational tags, phenomena involving the flow of information between situations in two directions [Hewitt 1975] is an indication of the inappropriateness of situations as a description of the information.

Finally, one problem which has not been solved in the development of truth maintenance systems is the ablility to easily name and switch between hypothetical situations. The switching problem is easy in context or situation based systems, for belief is constantly computed from contexts, instead of the opposite as holds in truth maintenance systems. The naming problem is also apparently somewhat simpler in context and situation based systems, for each named context has a specific set of contents. None of these systems,

however, including truth maintenance systems, provide satisfactory methods for determining the equivalence of apparently different contexts, although truth maintenance systems do not share the disadvantage of context based systems in which the name of a context depends upon the chronological order of its construction, and in which some contexts cannot easily be named because contexts can be based on irrelevant choices.

G. Comparison With Other Current Work

There are several precursors of this research into utilizing recorded justifications for belief. The most common use of dependencies in these systems is as an erasing mechanism, using the recorded connections between beliefs to discard only those items possibly affected by an erased item. In addition, these systems are monotonic, and are effectively limited to support-list representations.

Hayes' [1975] robot planning system is organized so that the planning functions record, for each choice made, the set of other choices upon which the new choice depends. These recorded support sets are then used to erase the choices derived from choices retracted or invalidated by unexpected plan execution discoveries. No recording of multiple justifications is made, and no discarded results are saved. Non-dependency mechanisms, apparently, are used to decide when choices are affected by new information, as the support sets represent only monotonic dependency information.

Fikes' [1975] data base system presents a somewhat more systematic treatment of a monotonic support-set dependency system. In this system, the primary use of the recorded support-sets is as an aid when erasing data base entries. Multiple justifications are not kept, nor are discarded results saved. Fikes introduces a mechanism which he employs to partially represent non-monotonic information, the set completeness indicator. This mechanism can be used to indicate the lack of knowledge about a particular query, but does

not enter into the support-sets of derived facts, and so does not cause updating of beliefs upon the discovery of new information. Completeness indicators are instead used strictly in reducing the effort involved in repeated searches.

McDermott [1975] uses a dependency system similar to that of Fikes, although without the mechanism of set completeness indicators. This system also uses the recorded dependencies solely as an erasing mechanism. Problems are inherent in the approach indicated by this system, however, for multiple support-sets are used, and erasing occurs only if all support-sets for an item are removed. As the examples in Section III.C demonstrate, this approach ignores the consequences of circularities in the dependency relationships, and so leads to the retention of unwanted and unsupported beliefs. McDermott has subsequently elaborated on this system [McDermott 1976], but the elaboration seems to be in attempting to encode additional forms of information in the dependencies.

In contrast to the above systems, which employed recorded justifications primarily as erasing mechanisms, the graphical deduction system of Cox and Pietrzykowski [1976] uses the recorded substitution histories produced by unifications to implement a somewhat improved variety of chronological backtracking and erasure. In this system, a failure indicates that an inappropriate unification has been made, and the histories of substitutions are examined to determine a culprit. This method lacks the power of dependency-directed backtracking for two reasons. The substitution histories form an inherently tree-organized

system of environments, thus introducing anomalous chronological dependencies into the sets of unification choices. In addition, the selected unification, once retracted, is not summarized in the form of a nogood to prevent future mistakes, but is instead used, as in the above systems, simply as an aid to erasing all unifications contained in the indicated subtree.

The ARS electronic circuit analysis program [Stallman and Sussman 1976] was the first system to make systematic, sophisticated use of recorded dependencies. Because of the recording of multiple justifications and the use of a simple form of truth maintenance (called fact garbage collection, which was not in their system an incremental process), the designers of this system discovered the effects of circularities on the maintenance of beliefs. This system also presented the method of dependency-directed backtracking, although extra-dependency mechanisms are used to implement assumptions and nogoods, as the monotonic dependency system of ARS cannot support the dependency-based methods described in this report. ARS also employed conditional proofs implicitly in its derivation of support for nogood assertions, but did not make any other use of this form of justification, implicitly or explicitly.

IV. Discussion

A. Summary of the Key Ideas

The major points elaborated in this report are the structure of a non-monotonic dependency system for representing knowledge about beliefs, the use of a truth maintenance system in using this representation of knowledge to maintain beliefs consistent with recorded justifications, the application of dependency relationships in effecting backtracking, and mechanisms for separating levels of detail and their dependencies.

The non-monotonic dependency system formalizes several types of justifications including premises, beliefs which are independent of other beliefs; deductions, beliefs derived from other beliefs; conditional proofs, beliefs summarizing the derivability of one belief from others; and assumptions, the non-monotonic mechanism whereby a belief is based on a lack of contradictory knowledge. These basic representational techniques combine to allow perspicuous implementations of sets of alternatives and selectors of equivalence class representatives.

Beliefs consistent with recorded justifications can be efficiently determined via truth maintenance, a process invoked whenever beliefs change due to the addition of new information or the retraction of hypotheses. Truth maintenance involves an examination

of those beliefs explicitly linked, by means of the dependency system, to the changed beliefs. The truth maintenance system exercises the care required to avoid spurious beliefs produced by circularities among the justifications for beliefs.

Exploiting all the facilities provided by the dependency and truth maintenance systems, dependency-directed backtracking examines the well-founded support recorded for beliefs involved in inconsistencies to determine the set of hypotheses underlying the inconsistency. Retraction of premises supporting an inconsistency is outside the domain of a truth maintenance system, but the dependency relationships involving non-monotonic assumptions can be analyzed to provide a basis for the retraction of assumptions. The causes of the inconsistency can be summarized via a conditional proof, and this summarized cause can then be used to add new justifications which retract one of the underlying assumptions by providing new knowledge which ends the lack of knowledge upon which the assumption was based.

Finally, the mechanism of conditional proof can easily be used to separate hierarchical levels of detail in explanations. This is accomplished by justifying information at one level in terms of the conditional proof of the corresponding information at the lower levels relative to the higher level structures. This separation is important not only in improving the clarity of explanations, but in aiding processes, like dependency-directed backtracking, which must interrogate these explanations to analyze inconsistencies. In the case of dependency-directed backtracking, the separation of levels of detail reduces

the number of assumptions relevant to a given inconsistency, thereby increasing the efficiency of the backtracking process.

B. Future Work

There are many areas needing exploration and elaboration related to the topics discussed in this report. Many of these concern the use of a truth maintenance system in explanation and hypothetical reasoning. Other problems are in improving the technical details of implementing truth maintenance systems. This section discusses these areas for future research.

A major application of the dependency relationships determined by a truth maintenance system is the explanation of computed entities in terms of the knowledge by which they were computed. The recorded justifications for beliefs only form the raw material from which explanations are to be constructed, however, for the explanations produced by simple examinations of the antecedents or foundations of beliefs are often too cluttered with unnecessary information. The mechanism of conditional proof forms an important tool for restructuring arguments to summarize information recognized as known by the querying entity. This report has indicated techniques for using this mechanism in structuring the problem solving behavior of a program into hierarchical levels of detail, but much interesting work can be done in developing techniques by which query routines

can perform goal and listener oriented restructuring of information after the basic computations have been performed. (Cf. [Carr and Goldstein 1977, Shortliffe 1976, Swartout 1977])

Several topics of interest concern domain-independent methods in hypothetical reasoning. As indicated in the discussion of backtracking, there are several possible criteria for analyzing the structure of assumptions involved in inconsistencies. Further exploration of backtracking schemes employing these criteria might provide added efficiencies in backtracking. Related topics are the use of the dependency relationships alone as measures of the strength or stability of arguments, and in estimating the effects of changes in beliefs.

The power of dependency-directed backtracking calls for integration of this method with knowledge-based methods of hypothetical reasoning. Programs with knowledge of the semantics of facts can greatly increase the efficacy of the knowledge-free automatic methods by supplying measures for the soundness of premises and arguments in backtracking and differential diagnosis. Similarly, such knowledge of the meanings of the premises involved might be used in methods for generalizing the information derived from inconsistencies into more widely applicable rules.

On a technical level, there are several algorithmic improvements possible (and necessary) to be made in the basic truth maintenance process itself. Speedups in truth maintenance processing might be gained if a better understanding is developed of the best

order for examination and the analysis of circularities is improved. Efficient (or even correct) methods for evaluating conditional proof antecedents out of context, and for switching contexts need to be developed. The use of multiple supporting-antecedents for beliefs is also a topic for study.

One important problem is the detailing of methods integrating the use of dependencies with the sharing of data. Systems such as Fahlman's [Fahlman 1975] avoid explosions of computation space by employing schemes for dealing with virtual copies of pieces of information. Straightforward schemes for using recorded justifications, however, require explicit copies of the information, and cannot deal with shared structure. It would be very valuable to develop methods for this integration, perhaps of the nature of shared or virtually-copied dependency structures.

Finally, there are many interesting applications of dependencies to problems of control in problem solving systems. Explicit justifications allow the separation of the control and the knowledge embodied by the problem solver, since the problem solver can thus make derived information depend only on non-control information. At the same time, dependencies permit explicit linking of control decisions to the information and decisions which they are based on, thus opening the possibility for careful failure analysis by the problem solving interpreter. (Cf. [Hayes 1973, Pratt 1977, De Kleer, Sussman and Doyle, forthcoming])

References

[Bobrow and Winograd 1976]
Daniel G. Bobrow and Terry Winograd, "An Overview of KRL, a Knowledge Representation Language," Xerox PARC Report CSL-76-4.

[Brown 1976]
Allen Brown, "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures," MIT AI Lab, TR-362, December 1976.

[Carr and Goldstein 1977]
Brian Carr and Ira Goldstein, "Overlays: A Theory of Modelling for Computer-Aided Instruction," MIT AI Lab, Memo 406, February 1977.

[Cook 1971]
S. A. Cook, "The Complexity of Theorem-Proving Procedures," Proc. 3rd Annual ACM Symp. on the Theory of Computing, 1971, pp. 151-158.

[Cox and Pietrzykowski 1976]
Philip T. Cox and T. Pietrzykowski, "A Graphical Deduction System," Department of Computer Science Research Report CS-75-35, University of Waterloo, July 1976.

[Ernst and Newell 1969]
George W. Ernst and Allen Newell, *GPS: A Case Study in Generality and Problem-Solving*, Academic Press, New York, 1969.

[Fahlman 1973]
Scott E. Fahlman, "A Hypothesis-Frame System for Recognition Problems," MIT AI Lab, Working Paper 57, December 1973.

[Fahlman 1974]
Scott E. Fahlman, "A Planning System for Robot Construction Tasks," *Artificial Intelligence, 5* (1974), pp. 1-49.

[Fahlman 1975]
Scott E. Fahlman, "Thesis Progress Report: A System for Representing and Using Real World Knowledge," MIT AI Lab, AI Memo 331, May 1975.

[Fahlman 1977]
Scott E. Fahlman, "A System for Representing and Using Real World Knowledge," forthcoming MIT PHD thesis.

[Fikes 1975]
Richard E. Fikes, "Deductive Retrieval Mechanisms for State Description Models," *IJCAI4*, September 1975, pp. 99-106.

[Goldstein 1974]
Ira P. Goldstein, "Understanding Simple Picture Programs," MIT AI Lab, TR-294, September 1974.

[Green 1969]
C. Cordell Green, "Theorem-Proving by Resolution as a Basis for Question-Answering Systems," in Meltzer and Michie, *Machine Intelligence 4*, pp. 183-205.

[Greenblatt, Eastlake and Crocker 1967]
R. Greenblatt, D. Eastlake, and S. Crocker, "The Greenblatt Chess Program," *Proc. 1967 FJCC*, pp. 801-810.

[Hawkinson 1975]
Lowell Hawkinson, "The Representation of Concepts in OWL," *IJCAI 4*, September 1975, pp. 107-114.

[Hayes 1973]
P. J. Hayes, "Computation and Deduction," *Proc. Symp. MFCS*, 1973.

[Hayes 1975]
Philip J. Hayes, "A Representation for Robot Plans," *IJCAI4*, September 1975, pp. 181-188.

[Hewitt 1972]
Carl Hewitt, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," MIT AI Lab, TR-258, April 1972.

[Hewitt 1975]
Carl Hewitt, "How to Use What You Know," *IJCAI4*, September 1975, pp. 189-198.

[Hintikka 1962]
Jaakko Hintikka, *Knowledge and Belief*, Cornell University Press, Ithica, New York, 1962.

[Karp 1972]
R. M. Karp, "Reducibility Among Combinatorial Problems," in Miller and Thatcher, editors, *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-104.

[Kuipers 1975]

Ben Kuipers, "A Frame for Frames: Representing Knowledge for Recognition," in Bobrow and Collins, editors, *Representation and Understanding*, pp. 151-184.

[McCarthy and Hayes 1969]
J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Meltzer and Michie, *Machine Intelligence 4*, pp. 463-502.

[McDermott 1974]
Drew Vincent McDermott, "Assimilation of New Information by a Natural Language-Understanding System," MIT AI Lab, AI-TR-291, February 1974.

[McDermott 1975]
Drew V. McDermott, "Very Large PLANNER-type Data Bases," MIT AI Lab, AI Memo 339, September 1975.

[McDermott 1976]
Drew V. McDermott, "Flexibility and Efficiency in a Computer Program for Designing Circuits," MIT EECS Department Ph.D. Thesis, September 1976.

[McDermott and Sussman 1974]
Drew V. McDermott and Gerald Jay Sussman, "The CONNIVER Reference Manual," MIT AI Lab, AI Memo 259a, January 1974.

[Minsky 1974]
Marvin Minsky, "A Framework for Representing Knowledge," MIT AI Lab, AI Memo 306, June 1974.

[Moore and Newell 1973]
J. Moore and A. Newell, "How Can Merlin Understand?," CMU Computer Science Department, November 1973.

[Moses 1967]
Joel Moses, "Symbolic Integration," MIT Project MAC TR-47.

[Pratt 1977]
Vaughan R. Pratt, "The Competence/Performance Dichotomy in Programming," MIT AI Lab, Memo 400, January 1977.

[Rescher 1964]
N. Rescher, *Hypothetical Reasoning*, Amsterdam: North Holland 1964.

[Rivest 1976]
Ronald Rivest, "On Self-Organizing Sequential Search Heuristics," *CACM 19, #2,*

(February 1976), pp. 63-67.

[Rubin 1975]
Ann D. Rubin, "Hypothesis Formation and Evaluation in Medical Diagnosis," MIT AI Lab, TR-316, January 1975.

[Rulifson, Derksen and Waldinger 1973]
Johns F. Rulifson, Jan A. Derksen, and Richard J. Waldinger, "QA4: A Procedural Calculus for Intutitive Reasoning," SRI AI Center, Technical Note 73, November 1973.

[Sacerdoti 1975]
Earl D. Sacerdoti, "A Structure for Plans and Behavior," SRI AI Center, TN 109, August 1975.

[Sandewall 1972]
E. Sandewall, "An Approach to the Frame Problem, and its Implementation," *Machine Intelligence 7*, pp. 195-204, 1972.

[Shortliffe 1976]
E. H. Shortliffe, *MYCIN: Computer-Based Medical Consultations*, American Elsevier, 1976.

[Slagle 1963]
James R. Slagle, "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus," in Feigenbaum and Feldman, *Computers and Thought*, pp. 191-203.

[Stallman and Sussman 1976]
Richard M. Stallman and Gerald Jay Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," MIT AI Memo 380, September 1976.

[Sussman 1975]
Gerald Jay Sussman, *A Computer Model of Skill Acquisition*, American Elsevier Publishing Company, New York, 1975.

[Sussman and McDermott 1972]
Gerald Jay Sussman and Drew V. McDermott, "From PLANNER to CONNIVER - A genetic approach," *Proc. AFIPS FJCC*, 1972, pp. 1171-1179.

[Sussman and Stallman 1975]
Gerald Jay Sussman and Richard Matthew Stallman, "Heuristic Techniques in Computer-Aided Circuit Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22,

No. 11, November 1975, pp. 857-865.

[Sussman, Winograd and Charniak 1971]
Gerald Jay Sussman, Terry Winograd and Eugene Charniak, "MICRO-PLANNER Reference Manual," MIT AI Lab, AI Memo 203a, December 1971.

[Swartout 1977]
William R. Swartout, "A Digitalis Therapy Advisor With Explanations," MIT LCS, TR-176, February 1977.

[Waldinger 1975]
Richard Waldinger, "Achieving Several Goals Simultaneously," SRI AI Center, Technical Note 107, July 1975.

[Winograd 1972]
Terry Winograd, *Understanding Natural Language*, Academic Press, New York, 1972.

[Winston 1970]
Patrick Henry Winston, "Learning Structural Descriptions From Examples," MIT AI Lab, TR-231, September 1970.

[Zdonik, Sjoberg, and Matz 1976]
S. Zdonik, R. Sjoberg, and M. Matz, "Levels of Description: Issues and Implementations," MIT 6.860 term paper, December 1976.

Appendix 1. A TMS Glossary

Aldiborontiphoscophornio!

Where left you Chrononhotonthologos?

Henry Carey, *Chrononhotonthologos*

The AFFECTED-CONSEQUENCES of a fact are those facts whose current support-status rests on the fact; precisely, those consequences of the fact such that the fact is a supporting-fact of each of these consequences.

The ANCESTORS of a fact are those facts involved at some level in determining the current support-status of the fact; precisely, the transitive closure of the set of supporting-facts of the fact under the operation of taking supporting-facts.

ANOMALOUS CHRONOLOGICAL DEPENDENCIES are present when beliefs are considered to depend on other beliefs which are logically independent. A good example of such anomalous dependencies are those seen in MICRO-PLANNER, where all assertions depend on all chronologically previous choices, regardless of considerations of the logical relations between the assertions and the choices.

An ANTECEDENT of a fact is a representation for a justification for belief in the fact. There are two basic representations used in the truth maintenance system presented here, called support-list antecedents (SL-ANTECEDENTS) and conditional-proof antecedents (CP-ANTECEDENTS).

The ANTECEDENT-FACTS of a fact are those facts which currently support belief in the fact. The antecedent-facts of a fact are the same as the SUPPORTING-FACTS if the fact is *in*. Facts which are *out* have no antecedent-facts.

The ANTECEDENT-SET of a fact is the set of ANTECEDENTS justifying belief in the fact.

ASSUMPTIONS are non-monotonic justifications, in that the addition of new knowledge can cause the justifications for belief in assumptions to become invalid. Specifically, an assumption is a fact believed on the basis of a lack of knowledge about some other belief. A typical form of an assumption is a fact f whose justification is the predicate (OUT $\sim f$), where $\sim f$ denotes the negation of f. In this case, belief in f will be justified as long as contradictory knowledge is not present.

BACKTRACKING is the process of undoing a failure or inconsistency by undoing some premise or assumption underlying the failure or inconsistency.

BELIEF in a fact results from knowledge of a valid justification.

CHRONOLOGY describes the dependence of actions on the temporal ordering of their execution.

CONDITIONAL PROOFS are justifications which support a belief if a specified belief (the consequent of the conditional proof) is believed when given specified hypotheses (the hypotheses or moduli of the conditional proof).

The CONSEQUENCES of a fact are those facts such that the fact occurs in an antecedent of each consequence.

CONTEXTS are a data base organization introduced in the CONNIVER and QA4 artificial intelligence programming languages. [McDermott and Sussman 1974, Rulifson, Derksen and Waldinger 1973] Contexts allow for a tree-structured set of incrementally differing data bases, such that additions and deletions to a particular context affect only that context and its subcontexts.

A CONTRADICTION is a fact which has been designated as an inconsistency to the truth maintenance system. The backtracking mechanisms will always attempt to remove belief in contradictions.

The CP-CONSEQUENT-LIST of a fact contains those facts possessing a CP-antecedent with the fact as the consequent of the conditional proof.

A DEDUCTION is a type of justification for belief in a fact in which belief in the fact is justified if each fact in a designated set of facts is believed.

DEPENDENCIES are relationships between beliefs. The most generally useful dependencies in a truth maintenance system are the relations of ANTECEDENT-FACTS, CONSEQUENCES, SUPPORTING-FACTS and AFFECTED-CONSEQUENCES.

DEPENDENCY-DIRECTED describes processes, such as DEPENDENCY-DIRECTED BACKTRACKING, which operate on beliefs by searching through the facts related by dependencies to the beliefs.

A FACT is the fundamental entity to which justifications for belief can be attached.

FINDINDEP is a procedure for determining the set of facts which are the independent support of a belief relative to a specified set of beliefs. This procedure is the basis for mechanisms dealing with summarization by conditional proofs.

The FOUNDATIONS of a fact are those facts involved in the well-founded support for belief in the fact; precisely, the transitive closure of the antecedent-facts of the fact under

the operation of taking antecedent-facts.

A HYPOTHESIS is an underived belief, that is, either a PREMISE or an ASSUMPTION.

IN describes the condition of a fact which is believed to be *true* due to knowledge of a valid justification supporting this belief.

The INDEPENDENT SUPPORT of a belief relative to a set of beliefs is a set of facts such that the consequent belief is justified if the specified set of beliefs hold and the facts of the independent support are believed. In a non-monotonic dependency system, both the specified set of beliefs and the independent support each decompose into two sets, separating the *in* and *out* facts of each set.

A JUSTIFICATION is a reason for belief in a fact.

MONOTONIC TRUTH MAINTENANCE SYSTEMS are systems in which belief in a fact cannot be predicated upon a lack of belief in some other fact.

A NOGOOD is a fact summarizing the independent support of an inconsistency relative to the set of assumptions underlying the inconsistency.

A NOGOOD-SET is a set of assumptions designated to be inconsistent by a NOGOOD.

NON-CHRONOLOGICAL describes processes in which the order of actions does not affect the results, so that the relationships between the data produced can be summarized in time-independent, logical terms.

OUT describes the condition of a fact for which no valid justifications are known.

A PREMISE is a belief which depends on no other beliefs.

The REPERCUSSIONS of a fact are those other facts whose support-statuses are affected at some level by the support-status of the fact; precisely, the transitive closure of the affected-consequences of the fact under the operation of taking affected-consequences.

The STATUS of a fact normally refers to the SUPPORT-STATUS of the fact.

The SUPPORT for a belief is a generic term and may refer to the SUPPORTING-ANTECEDENT, ANTECEDENT-FACTS, SUPPORTING-FACTS, FOUNDATIONS or ANCESTORS of the fact, depending on the context of usage.

The SUPPORT-STATUS of a fact is another name for the *in*ness or *out*ness of the fact.

The SUPPORTING-ANTECEDENT determines a proof of belief in the fact in terms of other facts with well-founded (non-circular) support.

The SUPPORTING-FACTS of a fact is the set of facts affecting the current support-status of the fact

TRUTH MAINTENANCE refers to the process by which beliefs are redetermined when other beliefs change due to the addition of new information or the retraction of premises.

TRUTH MAINTENANCE SYSTEMS are systems in which belief in the truth of facts is determined from recorded justifications for belief.

UNSATISFIABLE DEPENDENCY RELATIONSHIPS are cycles of justifications such that no assignment of *in* or *out* to the facts involved is consistent with the justifications recorded for belief in the facts.

VALID JUSTIFICATIONS are justifications which as predicates evaluate *true*.

WELL-FOUNDED SUPPORT for a belief is an argument in terms of the recorded justifications from the basic premises and assumptions of the system upwards with no cycles in the arguments.

Appendix 2. Monotonic Truth Maintenance Systems

Monotonic truth maintenance systems are those in which no fact can depend upon another fact being *out*, that is, the *in*ning of a fact cannot cause the *out*ing of another. Such dependency systems are somewhat simpler to implement than the general system described in the text, admitting some of the same uses and general properties. The basic limitation of such systems is their inability to model assumptions. Since using outside mechanisms to implement assumptions really need to effect non-monotonic dependencies, monotonic systems are subject to all of the problems encountered in the general system, and so any external assumption mechanism must handle the same cases as those affecting the design of a general truth maintenance system.

The representation of dependencies can be simplified somewhat in a monotonic truth maintenances system, so that SL-antecedents are simply lists of facts, and CP-antecedents have only one list of moduli. The ideas of consequences, supporting-antecedent, supporting-facts, ancestors and repercussions are as in the general case, although now the list of supporting-facts is precisely the list of facts in the supporting-antecedent.

The process of truth maintenance is on the surface simplified somewhat in a monotonic system, and is conveniently divided into two processes; *out*ing and un*out*ing. The process of *out*ing occurs whenever a fact is changed from *in* to *out*, and consists of making a list of the repercussions of the invoking fact, setting the support-status of each of

these facts to *out*, setting the supporting-antecedents of these to NIL, and only after these steps have been performed for all repercussions, un*out*ing each of the repercussions.

The process of un*out*ing proceeds by examining the antecedent-set of the fact for an SL-antecedent all of whose facts are *in*. If such an antecedent is found, it is made the supporting-antecedent of the fact, the fact's support-status becomes *in*, and all *out* consequences of the fact are un*out*ed recursively. The handling of CP-antecedents is essentially unchanged from that described previously for non-monotonic systems.

The major effect of the monotonicity is that the mechanisms for making and maintaining assumptions must be done by means external to the truth maintenance system. One method for effecting non-monotonic relationships is that of using forget and recall functions, which are functions attached to the fact. A forget function is run whenever the attached fact is *out*ed, and a recall function is run whenever the attached fact is *in*ned. To implement the assumption of a fact *f* using such functions, a forget function can be attached to ~*f* such that if ~*f* is *out*ed, *f* will be made *in*, along with a recall function on ~*f* which will *out f* if ~*f* is *in*ned. This however is using much more mechanisms than necessary, as functions are being used instead of uniform dependency relationships. Moreover, the same circularities and inconsistencies of assumptions possible in a general dependency system will be present here, but their recognition is obscured by the non-uniform mechanism. Because of the chronological nature of these functions, no guarantee can be made that all feasible selections of statuses can be made in the event of circularities,

and not guarantee can be made that unsatisfiable dependency structures will no cause these functions to inititate and maintain an infinite loop of assuming, un*out*ing, *out*ing, reassuming, and so on.

Many of the applications of the general truth maintenance system are also possible using a monotonic system. The uses of dependencies as automatic erasing mechanisms, in explanation, generalization, and hypothetical reasoning are similar to those in the general system. Backtracking is affected in that assumptions can not be made by way of the dependencies, so all facts representing assumptions must be explicitly marked as such by external methods, and the backtracking procedures must recognize these marks. Similarly, while nogoods are representable as before, the ruling out of certain combinations of choices cannot now be done by means of new justifications, but must operate by external mechanisms.

Appendix 3. An Implementation of a TMS

This appendix presents the current (April 30, 1977, C. F. Gauss' 200th birthday) version of the TMS. A set of descriptions of the functions are provided, followed by the MacLISP programs.

```
The TMS is a set of functions which can be used to maintain
beliefs and justifications for program information components.
The following describes the functions provided by the TMS and
the functions required of the user by the TMS.


The TMS operates on internal data structures called nodes.
The format of these is not of interest to the user.
The user can obtain one of these nodes pointing to user data
structures by asking the TMS to make a new node with a given
name.  The name is the user-supplied data structure, and
should be printable by means of a LISP function.


The entry points to the Truth Maintenance System are as follows:

Function Name                    Arguments

tms-make-dependency-node (external-name)
          ;;Returns a new node with a given name.

tms-justify (node insupporters outsupporters argument)
          ;;Gives a TMS node a new justification,
          ;; which is a list of IN TMS nodes,
          ;; and a list of OUT TMS nodes.
          ;; The argument is an uninterpreted slot
          ;; visible via the tms-antecedent-argument function.
          ;;Returns NIL if no change in status occurred,
          ;; T otherwise.
```

```
tms-cp-justify (node conclusion in-premises out-premises argument)
          ;;Like justify, except specifying a conditional
          ;; proof antecedent of node in terms of the
          ;; support of the conclusion modulo the in premises
          ;; being in, and the out premises being out.

tms-contradiction (type supporters argument)
          ;;Creates a contradiction node in the TMS.
          ;; The type is a mnemonic indicator of the
          ;; type of the contradiction to be printed
          ;; if the *see-contradictions-sw* switch is set.
          ;; TYPE should therefore be prin1-able.
          ;; SUPPORTERS is a list of nodes which are used
          ;;  as the insupporters in justifying the contradiction.
          ;; ARGUMENT is an external form of the argument for
          ;; the contradiction, as in JUSTIFY.

tms-process-contradiction
            (contradiction      ;;the name of the contradiction (prin1-able)
             contradiction-node ;;the node representing the contradiction
             contradiction-type);;the type of contradiction (prin1-able)
          ;; As an alternative to TMS-CONTRADICTION, which
          ;; generates a new TMS-NODE to represent the
          ;; contradiction, this function takes an arbitrary
          ;; TMS node and indicates that it is a contradiction.
          ;;Instructs the TMS to backtrack from this
          ;; contradiction, producing justifications
          ;; for the assumptions underlying the contradiction.
          ;;Returns 'FOUND-A-CULPRIT if some assumption
          ;; was discovered and justified, and
          ;; 'FOUND-NO-CHOICES otherwise.

tms-findindep (node in-moduli-nodes out-moduli-nodes)
          ;;Returns a pair of lists (INLIST . OUTLIST)
          ;; of the independent support of node modulo
          ;; the moduli-nodes.

tms-see-stats ()
          ;;Print out statistics accumulated about TMS operations.

tms-initialize-stats ()
          ;;Initialize the accumulated statistics.
```

```
tms-support-status (node)
          ;;Returns the support-status of a node. ('IN or 'OUT)

tms-antecedent-set (node)
          ;;Returns the list of justifications of the node.

tms-supporting-antecedent (node)
          ;;Returns the current justification of the node.

tms-antecedent-argument (antecedent)
          ;;Returns the external argument associated
          ;; with the given antecedent.

tms-supporting-nodes (node)
          ;;Returns the list of nodes upon whom the given
          ;; node's support-status depends.

tms-antecedents (node)
          ;;Returns the list of nodes providing well-founded
          ;; support for the given node.

tms-consequences (node)
          ;;Returns the list of nodes which have justifications
          ;; mentioning the given node.

tms-external-name (node)
          ;;Returns the user-supplied name of a node.

tms-is-in (node)
          ;;A predicate true iff the node is IN.

tms-is-out (node)
          ;;A predicate true iff the node is OUT.

tms-are-in (nodelist)
          ;;A predicate true iff all nodes in the list are IN.

tms-are-out (nodelist)
          ;;A predicate true iff all nodes in the list are OUT.

tms-affected-consequences (node)
          ;;Returns a list of all consequences of a node which
          ;; are supported by the node.
```

```
tms-believed-consequences (node)
          ;;Returns a list of all IN consequences of a node
          ;; which are supported by the node.

tms-all-antecedents (node)
          ;;Returns a list of all antecedent nodes of a node
          ;; and all their antecedent nodes recursively.

tms-all-consequences (node)
          ;;Returns a list of all believed consequences of a
          ;; node and all their believed consequences recursively.

tms-retract (node)
          ;;Removes premise-support from a node.

tms-premises (node)
          ;;Returns a list of the premises underlying the node.

tms-assumptions (node)
          ;;Returns a list of the assumptions underlying the node.

tms-clobber-name-print-function (node)
          ;;Sets the (LISP) function that the TMS will call
          ;; to print the external names of nodes.
          ;; If no such function is found, PRIN1 is used.

tms-clobber-signal-forgetting-function (node)
          ;;Sets the (LISP) function that the TMS will call
          ;; when changing the status of the node from IN to OUT.

tms-clobber-signal-recalling-function (node)
          ;;Sets the (LISP) function that the TMS will call
          ;; when changing the status of the node from OUT to IN.
```

The TMS also generates new facts internally during backtracking. These will therefore occur in explanations and antecedents of user system facts. The following functions are for dealing with these internal facts.

The internal facts generated by the TMS are symbols (whose interning

is controlled by *tms-intern-sw* as below).
The predicate TMS-FACTP is T if an atom is an internal TMS fact, and NIL otherwise.
The function TMS-FACT-NODE returns the TMS node associated with an internal fact.
The function TMS-FACT-STATEMENT returns the symbolic statement of the meaning of
an internal fact. This statement refers to the external names of the other facts,
such as contradictions and assumptions, which are involved in the making of the fact.

It is thus wise for the user to access the TMS node of a user fact by means of
a function which checks for the representation of the fact (external or TMS internal)
and returns the appropriate thing. Similarly, an access function for the
statement of facts which decides which representation is being used is useful.

The TMS provides the following two functions for debugging purposes:

```
tms-init            ;;Completely flushes all known TMS nodes and state.

tms-intern          ;;Interns all known TMS nodes and sets *tms-intern-sw*
                    ;; (see below) so as to intern future nodes as well.
```

The TMS has the following switches which may be set for wallpaper purposes.

```
Variable (Default value)                    Use

*tms-see-tmp-sw* (nil)                      ;;Announces truth maintenance processing.

*tms-see-tmp-invoker* (t)                   ;;Controls printing of nodes invoking truth
                                            ;; maintenance processing if *see-tms-sw* is set.

*tms-see-justify-sw* (nil)                  ;;Announces the addition of a new justification
                                            ;; for a node.

*tms-see-contradictions-sw* (t)             ;;Announces the processing of a contradiction.

*tms-intern-sw* (nil)                       ;;If non-nil, interns new tms-nodes.
```

(comment General-Purpose Truth Maintenance System)

```
; TMS SPECIAL SYSTEM VARIABLES

(declare (special *tms-noted-in-facts*
                  *tms-noted-out-facts*
                  *tms-process-queue*

                  *tms-contradiction-assumptions*

                  *tms-facts*

                  *tms-see-tmp-sw*
                  *tms-see-tmp-invoker-sw*
                   *tms-see-justify-sw*
                  *tms-see-contradictions-sw*))

(setq *tms-facts* nil)
(setq *tms-see-tmp-sw* nil)
(setq *tms-see-tmp-invoker-sw* t)
(setq *tms-see-justify-sw* nil)
(setq *tms-see-contradictions-sw* t)

(defun tms-init ()
         (setq *tms-process-queue* nil)
         (setq *tms-noted-in-facts* nil)
         (setq *tms-noted-out-facts* nil)
         (setq *tms-contradiction-assumptions* nil)
         (mapc '(lambda (f) (setplist f nil) (makunbound f) (remob f)) *tms-facts*)
         (setq *tms-facts* nil)
         (tms-initialize-stats)
         'done)

(declare (special *tms-gens* *tms-intern-sw*) (fixnum *tms-gens*))
(setq *tms-gens* 0)
(setq *tms-intern-sw* nil)

(defun tms-gens (x)
         (setq *tms-gens* (1+ *tms-gens*))
         ((lambda (name) (and *tms-intern-sw* (intern name)) name)
          (maknam (append '(t m s -) (and x (nconc (explodec x) '(-))) (explodec *tms-gens*)))))

(defun tms-intern () (setq *tms-intern-sw* t) (mapc 'intern *tms-facts*) 'done)

(comment Truth Maintenance Data Structures)
```

```
;Facts have the following attributes:

;findindep-mark                          ;used in FINDINDEP processing
;subordinates-mark

;superiors-mark                          ;used in CONTRADICTION processing

;tms-status                              ;used in truth maintenance processing

;support-status                          ;properties of facts
;si-antecedent-set
;cp-antecedent-set
;supporting-antecedent
;supporting-facts
;consequences
;external-name
;name-print-function
;signal-forgetting-function
;signal-recalling-function

;fact-mark                               ;used in set computations (equality and condensing)
;explain-mark                            ;used in gathering ancestors, etc for explanations

;The support-status, si-antecedent-set, cp-antecedent-set, supporting-antecedent,
; supporting-facts, consequences, external-name, name-print-function,
; signal-forgetting-function, signal-recalling-function, and explain-mark
; of a fact may exist all the time.  A careful study of the
; operation of the tms shows that the fact-mark, and tms-status
; must coexist, as must findindep-mark and subordinates-mark, but that these
; two sets, as well as superiors-mark, are distinct in time, and so can share storage.

;The following is one implementation of a fact.
; Experimentation may prove other organizations to be more efficient.

(defmac accessmacs (slotn propn)
          (let ((cname (implode (nconc (explodec 'clobber-) (explodec slotn))))
                (pn (or propn slotn)))
               `(progn 'compile
                       (defmac ,slotn (fact) `(get ,fact ,'',pn))
                       (defmac ,cname (fact new)
                                  (cond (new `(putprop ,fact ,new ,'',pn))
```

```
                                        (t "(remprop ,fact ,'',pn)))))))

(defmac support-status (fact) "(symeval ,fact))
(defmac clobber-support-status (fact new) "(set ,fact ,new))
(accessmacs supporting-facts)
(defmac antecedents (fact) "(and (eq (support-status ,fact) 'in) (supporting-facts ,fact)))
(accessmacs consequences)
(accessmacs si-antecedent-set)
(accessmacs cp-antecedent-set)
(accessmacs supporting-antecedent)
(accessmacs external-name)
(accessmacs name-print-function)
(accessmacs signal-recalling-function)
(accessmacs signal-forgetting-function)
(accessmacs fact-mark)
(accessmacs tms-status)
(accessmacs tms-noted-mark)
(accessmacs findindep-mark)
(accessmacs subordinates-mark)
(accessmacs superiors-mark)
(accessmacs explain-mark)
(accessmacs cp-consequent-list)
(accessmacs contradiction-name)
(accessmacs contradiction-type)
(accessmacs contradiction-mark)
(accessmacs contradiction-nogoods)
(accessmacs nogood-assumptions)
(accessmacs nogood-contradiction)

(defmac is-in (fact) "(eq (support-status ,fact) 'in))
(defmac is-out (fact) "(eq (support-status ,fact) 'out))

(defun tms-support-status (node) (support-status node))
(defun tms-supporting-nodes (node)  (supporting-facts node))
(defun tms-antecedents (fact) (antecedents fact))
(defun tms-consequences (node) (consequences node))
(defun tms-antecedent-set (node) (append (si-antecedent-set node) (cp-antecedent-set node) nil))
(defun tms-supporting-antecedent (node) (supporting-antecedent node))
(defun tms-external-name (node) (external-name node))
(defun tms-clobber-name-print-function (node fun)
          (cond (fun (clobber (name-print-function node) fun))
            (t (remprop node 'name-print-function))))
(defun tms-clobber-signal-forgetting-function (node fun)
```

```
          (cond (fun (clobber (signal-forgetting-function node) fun)
                     (and (is-out node) (funcall fun (external-name node))))
                (t (remprop node 'signal-forgetting-function))))
(defun tms-install-signal-forgetting-function (node fun)
          (cond (fun (clobber (signal-forgetting-function node) fun))
                (t (remprop node 'signal-forgetting-function))))
(defun tms-clobber-signal-recalling-function (node fun)
          (cond (fun (clobber (signal-recalling-function node) fun)
                     (and (is-in node) (funcall fun (external-name node))))
                (t (remprop node 'signal-recalling-function))))
(defun tms-install-signal-recalling-function (node fun)
          (cond (fun (clobber (signal-recalling-function node) fun))
                (t (remprop node 'signal-recalling-function))))

(defun tms-make-dependency-node (name)
          (let ((node (tms-gens nil)))
                (push node *tms-facts*)
                (clobber (support-status node) 'out)
                (clobber (external-name node) name)
                node))

;The following define the structure of antecedents.

(defmac antecedent-argument (ante) "(cdr ,ante))
(defmac tms-antecedent (ante) "(car ,ante))
(defmac si-antecedent-inlist (ante) "(car (tms-antecedent ,ante)))
(defmac si-antecedent-outlist (ante) "(cdr (tms-antecedent ,ante)))
(defmac cp-antecedent-fact (ante) "(cadr (tms-antecedent ,ante)))
(defmac cp-antecedent-in-moduli (ante) "(caddr (tms-antecedent ,ante)))
(defmac cp-antecedent-out-moduli (ante) "(cdddr (tms-antecedent ,ante)))
(defmac new-antecedent (inlist outlist extarg) "(cons (cons ,inlist ,outlist) ,extarg))
(defmac new-cp-antecedent (cpc inmoduli outmoduli extarg)
           "(cons (cons 'cp (cons ,cpc (cons ,inmoduli ,outmoduli))) ,extarg))

(defun tms-antecedent-argument (ante) (antecedent-argument ante))

; These functions prevent recording duplicate antecedents for facts.

(defun tms-ante-set-member (ante ante-set)
          (do ((as ante-set (cdr as)))
              ((null as))
              (and (tms-equal-fact-sets (si-antecedent-inlist ante)
                                        (si-antecedent-inlist (car as)))
```

```
                    (tms-equal-fact-sets (si-antecedent-outlist ante)
                                         (si-antecedent-outlist (car as)))
                    (return t))))

(defun tms-cp-ante-set-member (ante ante-set)
         (do ((as ante-set (cdr as)))
              ((null as))
              (and (eq (cp-antecedent-fact ante) (cp-antecedent-fact (car as)))
                    (tms-equal-fact-sets (cp-antecedent-in-moduli ante)
                                         (cp-antecedent-in-moduli (car as)))
                    (tms-equal-fact-sets (cp-antecedent-out-moduli ante)
                                         (cp-antecedent-out-moduli (car as))))))

; These define the format of internally generated facts (like nogoods, etc.)


(accessmacs is-internal-tms-fact)
(accessmacs internal-tms-fact-statement)
(accessmacs internal-tms-fact-node)

;Checks whether an atom is an internal TMS fact.
(defun tms-factp (fact) (is-internal-tms-fact fact))

;Gets the statement of an internal TMS fact.
(defun tms-fact-statement (fact) (internal-tms-fact-statement fact))

;Gets the TMS node of an internal TMS fact.
(defun tms-fact-node (fact) (internal-tms-fact-node fact))

;TMS-MAKE-INTERNAL-FACT generates a new internal fact of a given type
; (which is spliced into the name of the fact) and a statement.

(defun tms-make-internal-fact (type statement)
         (let ((fact (tms-gens type)))
               (clobber (is-internal-tms-fact fact) t)
               (clobber (internal-tms-fact-node fact) (tms-make-dependency-node fact))
               (clobber (internal-tms-fact-statement fact) statement)
               fact))

(comment Truth Maintenance Support Functions)

(defun tms-is-in (fact) (is-in fact))
(defun tms-is-out (fact) (is-out fact))
```

```
(defun tms-are-in (facts)
         (do ((fl facts (cdr fl)))
              ((null fl) t)
              (or (is-in (car fl)) (return nil))))

(defun tms-are-out (facts)
         (do ((fl facts (cdr fl)))
              ((null fl) t)
              (or (is-out (car fl)) (return nil))))

(defun tms-add-consequence (fact consequence)
         (or (memq consequence (consequences fact))
              (clobber (consequences fact)
                       (cons consequence (consequences fact)))))

; TMS-AFFECTED-CONSEQUENCES returns a list of just those consequences
;  of a fact which actually mention the fact in their antecedents.

(defun tms-affected-consequences (fact)
         (mapcan '(lambda (cf) (and (memq fact (supporting-facts cf))
                                    (list cf)))
                 (consequences fact)))

(defun tms-affects-facts (fact)
         (do ((c (consequences fact) (cdr c)))
              ((null c))
              (and (memq fact (supporting-facts (car c)))
                   (return t))))

(defun tms-believed-consequences (f)
         (mapcan '(lambda (cf) (and (memq f (antecedents cf)) (list cf)))
                 (consequences f)))

(defun tms-supports-facts (fact)
         (do ((c (consequences fact) (cdr c)))
              ((null c))
              (and (memq fact (antecedents (car c))) (return t))))

;the following functions return all facts of the specified type
;in relationship to the given fact

(defun tms-all-antecedents (fact)
```

```
        (let ((factlist (mapcan 'tms-mark-all-antecedents (antecedents fact))))
              (mapc '(lambda (f) (clobber (explain-mark f) nil))
                    factlist)))

(defun tms-all-consequences (fact)
        (let ((factlist (mapcan 'tms-mark-all-consequences (tms-believed-consequences fact))))
              (mapc '(lambda (f) (clobber (explain-mark f) nil))
                    factlist)))

(defun tms-mark-all-antecedents (fact)
        (cond ((null (explain-mark fact))
               (clobber (explain-mark fact) 'all-marked)
               (cons fact (mapcan 'tms-mark-all-antecedents (antecedents fact))))))

(defun tms-mark-all-consequences (fact)
        (cond ((null (explain-mark fact))
               (clobber (explain-mark fact) 'all-marked)
               (cons fact (mapcan 'tms-mark-all-consequences
                                  (tms-believed-consequences fact))))))

(defun tms-premises (fact)
        (let ((pl (tms-premises1 fact)))
              (tms-premises2 fact)
              pl))

(defun tms-premises1 (fact)
        (cond ((not (explain-mark fact))
               (clobber (explain-mark fact) 'premises)
               (cond ((antecedents fact) (mapcan 'tms-premises1 (antecedents fact)))
                     (t (and (is-in fact) (list fact)))))))

(defun tms-premises2 (fact)
        (cond ((explain-mark fact)
               (clobber (explain-mark fact) nil)
               (mapc 'tms-premises2 (antecedents fact)))))

(defun tms-assumptions (fact)
        (let ((al (tms-assumptions1 fact)))
              (tms-assumptions2 fact)
              al))

(defun tms-assumptions1 (fact)
        (cond ((not (explain-mark fact))
```

```
                (clobber (explain-mark fact) 'assumptions)
                (let ((flag nil))
                     (let ((ans (mapcan '(lambda (a)
                                           (and (is-out a) (setq flag t))
                                           (tms-assumptions1 a))
                                        (antecedents fact))))
                          (cond (flag (cons fact ans))
                                (t ans)))))))

(defun tms-assumptions2 (fact)
       (cond ((explain-mark fact)
              (clobber (explain-mark fact) nil)
              (mapc 'tms-assumptions2 (antecedents fact)))))

;This kludge takes a list of facts and clobbers it to be a non-repetitive list
;  of the same facts.
; The special case of two or fewer facts in the list is handled specially for speed.

(defun tms-fact-set-condense (l)
       (cond ((null l) nil)
             ((cddr l)
              (clobber (fact-mark (car l)) t)
              (do ((cl l (cdr cl))
                   (next (cdr l) (cdr next))
                   (e))
                  ((null next))
                  (setq e (car next))
                  (cond ((fact-mark e) (rplacd cl (cdr next)))
                        (t (clobber (fact-mark e) t))))
              (setq l (nreverse l))
              (cond ((null (car l)) (pop l)))
              (mapc '(lambda (e) (clobber (fact-mark e) nil)) l)
              l)
             ((eq (car l) (cadr l)) (cdr l))
             (t l)))

(defun tms-equal-fact-sets (x y)
       (cond ((null x) (null y))
             ((null y) nil)
             (t (mapc '(lambda (f) (clobber (fact-mark f) t)) x)
                (do ((a y (cdr a)))
                    ((null a)
                     (do ((a x (cdr a)))
```

```
                    ((null a) t)
                    (cond ((fact-mark (car a))
                           (mapc '(lambda (f) (clobber (fact-mark f) nil)) a)
                           (return nil)))))
                (cond ((fact-mark (car a))
                       (clobber (fact-mark (car a)) nil))
                      (t (mapc '(lambda (f) (clobber (fact-mark f) nil)) x)
                         (return nil)))))))

(defun tms-print (text fact)
       (terpri) (princ text) (princ '| |) (tms-prin1-name fact))

(defun tms-prin1-name (fact)
        (let ((pf (name-print-function fact)))
             (cond (pf (funcall pf (external-name fact)))
                   (t (prin1 (external-name fact))))))

(defun tms-signal-status-change (fact oldstatus newstatus)
       (cond ((eq oldstatus newstatus))
             ((eq newstatus 'in)
              (let ((rf (signal-recalling-function fact)))
                   (and rf (funcall rf (external-name fact)))))
             ((eq newstatus 'out)
              (let ((ff (signal-forgetting-function fact)))
                   (and ff (funcall ff (external-name fact)))))
             (t (error 'tms-signal-status-change fact 'wrng-type-arg))))

(defun tms-see-facts ()
       (mapc '(lambda (f)
                (print f)
                (princ '| (|) (prin1 (support-status f)) (princ '|) = |)
                (prin1 (external-name f)))
             *tms-facts*)
       'done)

(comment General Truth Maintenance System)
;TMS-JUSTIFY can be used to supply a new antecedent for
;a fact, and to then determine its support status. If the fact
;lacks well founded support, truth maintenance occurs.

; TMS-JUSTIFY returns NIL if no change in status occurred, T otherwise.

(defun tms-justify (fact ins outs extarg)
```

```
        (let ((*tms-noted-in-facts* nil)
              (*tms-noted-out-facts* nil)
              (oldstatus (support-status fact)))
             (tms-justify1 fact ins outs extarg)
             (tms-tmp-scan)
             (tms-signal-changes)
             (not (eq oldstatus (support-status fact)))))

;TMS-JUSTIFY1 returns T if the justification causes truth maintenance, NIL otherwise.

(defun tms-justify1 (fact ins outs extarg)
       (increment *tms-justify-counts)
       (let ((ante (new-antecedent ins outs extarg)))
            (cond ((let ((as (si-antecedent-set fact)))
                        (cond ((not (tms-ante-set-member ante as))
                               (clobber (si-antecedent-set fact) (nconc as (list ante)))
                               t)))
                   (increment *tms-effective-justify-counts)
                   (mapc '(lambda (f) (tms-add-consequence f fact)) ins)
                   (mapc '(lambda (f) (tms-add-consequence f fact)) outs)
                   (and *tms-see-justify-sw* (tms-print '|Justifying| fact))
                   (eqcase (support-status fact)
                           (in nil)
                           (out (eqcase (tms-eval-antecedent ante)
                                        (yes (tms-tmp (list fact)) t)
                                        (no (tms-install-antecedents fact) nil)))))))))

;TMS-CP-JUSTIFY can be used to provide a conditional-proof antecedent
; for a fact.  The conditional proof is of the form "the support
; of cp-fact modulo the moduli-facts."

;TMS-CP-JUSTIFY returns NIL if no change in status occurred, T otherwise.

(defun tms-cp-justify (fact cpc inmoduli outmoduli extarg)
       (let ((*tms-noted-in-facts* nil)
             (*tms-noted-out-facts* nil)
             (oldstatus (support-status fact)))
            (tms-cp-justify1 fact cpc inmoduli outmoduli extarg)
            (tms-tmp-scan)
            (tms-signal-changes)
            (not (eq oldstatus (support-status fact)))))

(defun tms-cp-justify1 (fact cpc inmoduli outmoduli extarg)
```

```
        (increment *tms-cp-justify-count*)
        (let ((ante (new-cp-antecedent cpc inmoduli outmoduli extarg)))
              (cond ((let ((as (cp-antecedent-set fact)))
                          (cond ((not (tms-cp-ante-set-member ante as))
                                 (clobber (cp-antecedent-set fact) (nconc as (list ante)))
                                 t)))
                     (increment *tms-effective-cp-justify-count*)
                     (clobber (cp-consequent-list cpc)
                              (cons fact (cp-consequent-list cpc)))
                     (tms-add-consequence cpc fact)
                     (mapc '(lambda (f) (tms-add-consequence f fact)) inmoduli)
                     (mapc '(lambda (f) (tms-add-consequence f fact)) outmoduli)
                     (and *tms-see-justify-su* (tms-print '|Justifying| fact))
                     (and (eq (support-status fact) 'out)
                          (eqcase (tms-eval-cp-antecedent ante)
                                  (yes
                                   (let ((support
                                          (tms-findindep
                                           (cp-antecedent-fact ante)
                                           (cp-antecedent-in-moduli ante)
                                           (cp-antecedent-out-moduli ante))))
                                     (tms-justify1 fact (car support) (cdr support)
                                                   (antecedent-argument ante))))
                                  (no (tms-install-antecedents fact))))))))

;TMS-RETRACT removes a premise antecedent from a fact.

(defun tms-retract (fact)
       (let ((*tms-noted-in-facts* nil)
             (*tms-noted-out-facts* nil)
             (oldstatus (support-status fact)))
            (tms-retract1 (list fact))
            (tms-tmp-scan)
            (tms-signal-changes)
            (not (eq oldstatus (support-status fact)))))

(defun tms-retract1 (factlist)
       (do ((fl factlist (cdr factlist))
            (tl nil))
           ((null fl) (and tl (tms-tmp tl)))
           (do ((as (si-antecedent-set (car fl)) (cdr as))
                (cas (cp-antecedent-set (car fl))))
               ((null as))
```

```
                (let ((ante (car as)))
                     (cond ((and (null (si-antecedent-inlist ante))
                                 (null (si-antecedent-outlist ante))
                                 (not (do ((cs cas (cdr cs)))
                                          ((null cs))
                                          (and (eq (antecedent-argument ante)
                                                   (antecedent-argument (car cs)))
                                               (return t)))))
                            (and (eq ante (supporting-antecedent (car fl)))
                                 (push (car fl) tl))
                            (clobber (si-antecedent-set (car fl))
                                     (delq ante (si-antecedent-set (car fl))))))))))

;The Truth Maintenance Processor:
;Truth maintenance processing occurs when the support status of a fact is changed.
;The maintenance processing is initiated by calling TMS-TMP with the list of facts in question.

(defun tms-tmp (factlist)
       (increment *tms-tmp-counts)
       (cond (*tms-see-tmp-s*
              (terpri)
              (princ '|Truth maintenance processing (invocation #|)
              (prin1 *tms-tmp-counts)
              (princ '|)|)
              (cond (*tms-see-tmp-invoker-s*
                     (cond ((null (cdr factlist))                    ;just one invoker
                            (princ '| initiated by |)
                            (tms-prin1-name (car factlist))))))
              (princ '|.|)))
       (setq *tms-process-queues* nil)
       (let ((noted-facts (mapcan 'tms-mark-affected-consequences factlist)))
            (do ((f (tms-dequeue) (tms-dequeue)))
                ((null f))
                (and (null (support-status f)) (tms-examine f)))
            (do ((fl noted-facts (cdr fl)))
                ((null fl))
                (or (support-status (car fl)) (tms-queue (car fl))))
            (do ((f (tms-dequeue) (tms-dequeue)))
                ((null f))
                (and (null (support-status f)) (tms-nwf-examine f)))
            (mapc '(lambda (f)
                           (cond ((null (support-status f))
                                  (print f)
```

```
                                (break | NULL TMS Error |))))
                        noted-facts)
                (cond (*tms-see-tmp-sw*
                        (print (length noted-facts))
                        (princ '|facts examined.|)))))

(defun tms-tmp-scan ()
        (prog ()
                loop                                                    ; sigh...
                (and (do ((fl *tms-noted-in-facts* (cdr fl)))
                        ((null fl))
                        (let ((f (car fl)))
                                (cond ((and (is-in f) (contradiction-mark f))
                                        (tms-process-contradiction1 (contradiction-name f) f
                                                                    (contradiction-type f))
                                        (return t)))))
                        (go loop))
                (and (do ((fl *tms-noted-out-facts* (cdr fl)))
                        ((null fl))
                        (let ((f (car fl)))
                                (cond ((and (is-in f) (contradiction-mark f))
                                        (tms-process-contradiction1 (contradiction-name f) f
                                                                    (contradiction-type f))
                                        (return t)))))
                        (go loop))
                (and (do ((fl *tms-noted-in-facts* (cdr fl)))
                        ((null fl))
                        (let ((f (car fl)))
                                (and (is-in f)
                                        (tms-check-cp-consequences f)
                                        (return t))))
                        (go loop))
                (and (do ((fl *tms-noted-out-facts* (cdr fl)))
                        ((null fl))
                        (let ((f (car fl)))
                                (and (is-in f)
                                        (tms-check-cp-consequences f)
                                        (return t))))
                        (go loop))))

(defun tms-signal-changes ()
        ;It is important that tms-signal-status-change does not cause
        ; further truth maintenance until the following loop is completed.
```

```
          (mapc '(lambda (nf)
                    (tms-signal-status-change nf 'in (support-status nf))
                    (clobber (tms-noted-mark nf) nil))
                *tms-noted-in-facts*)
          (setq *tms-noted-in-facts* nil)
          (mapc '(lambda (nf)
                    (tms-signal-status-change nf 'out (support-status nf))
                    (clobber (tms-noted-mark nf) nil))
                *tms-noted-out-facts*)
          (setq *tms-noted-out-facts* nil))

;The following functions perform the process queue maintenance operations.

;  The possible tms-statuses of a fact are as follows:

;  If the tms-noted-mark is NIL, then the fact has not been examined by the TMS

;  If the tms-status is non-NIL, then the fact is queued for TMS processing.

(defun tms-queue (fact)
          (cond ((not (tms-status fact))
                    (or (tms-noted-mark fact)
                         (error '|Non-noted fact in tms-queue| fact 'wrng-type-arg))
                    (clobber (tms-status fact) 'tms-queued)
                    (push fact *tms-process-queue*))))

(defun tms-dequeue ()
          (let (((fact (pop *tms-process-queue*)))
                (cond (fact (clobber (tms-status fact) nil) fact))))

;This function marks and queues all facts which might be
;affected by the change of support status of the argument fact.
;  The status of the fact before the TMS processing is also noted
;  to allow notification of status changes at the conclusion of TMS processing.

(defun tms-mark-affected-consequences (fact)
          (cond ((not (tms-status fact))
                    (cond ((null (tms-noted-mark fact))
                              (clobber (tms-noted-mark fact) t)
                              (eqcase (support-status fact)
                                        (in (push fact *tms-noted-in-facts*))
                                        (out (push fact *tms-noted-out-facts*))
                                        (else (error '|Statusless fact in tms-mark-affected-consequences|
```

```
                                                fact
                                                'wrng-type-arg)))))
                    (clobber (support-status fact) nil)
                    (clobber (supporting-antecedent fact) nil)
                    (clobber (supporting-facts fact) nil)
                    (tms-queue fact)
                    (cons fact (mapcan 'tms-mark-affected-consequences
                                       (tms-affected-consequences fact))))))

; TMS-CHECK-CP-CONSEQUENCES rederives support for conditionally proven facts
;  whenever the consequent of one of their conditional-proof antecedents
;  comes in.

(defun tms-check-cp-consequences (fact)
         (let ((changed nil))
               (mapc '(lambda (cpc)
                         (mapc '(lambda (ante)
                                   (cond ((eq fact (cp-antecedent-fact ante))
                                          (let ((support
                                                 (tms-findindep fact (cp-antecedent-in-modul ante)
                                                                     (cp-antecedent-out-modul ante))))
                                            (and (tms-justify1 cpc (car support) (cdr support)
                                                               (antecedent-argument ante))
                                                 (setq changed t))))))
                               (cp-antecedent-set cpc)))
                     (cp-consequent-list fact))
               changed))

;TMS-EXAMINE recursively checks facts for well founded support.

(defun tms-examine (fact)
         (increment *tms-examine-count*)
         (let ((newstatus (tms-well-founded-status fact)))
               (cond (newstatus
                        (clobber (support-status fact) newstatus)
                        (tms-install-antecedents fact)
                        (mapc '(lambda (c) (or (support-status c) (tms-queue c)))
                              (consequences fact))))))

(defun tms-nwf-examine (fact)
         (increment *tms-nwf-examine-count*)
         (or (let ((status (tms-well-founded-status fact)))
                  (cond (status
```

```
                        (clobber (support-status fact) status)
                        (tms-install-antecedents fact)
                        (tms-nwf-process-consequences fact status)
                        status)))
              (let ((status (tms-nwf-status fact)))
                    (clobber (support-status fact) status)
                    (tms-nwf-install-antecedents fact)
                    (tms-nwf-process-consequences fact status)
                    status)))

(defun tms-nwf-process-consequences (fact status)
         (cond ((eq status 'in)
                (mapc '(lambda (c)
                         (cond ((null (support-status c)) (tms-queue c))
                               ((memq fact (supporting-facts c))
                                (tms-mark-affected-consequences c))))
                      (consequences fact)))
               (t (mapc '(lambda (c)
                           (or (support-status c) (tms-queue c)))
                        (consequences fact)))))

;TMS-WELL-FOUNDED-STATUS computes the well-founded support status of
; a fact from both its antecedent-sets.

(defun tms-well-founded-status (fact)
        (increment *tms-well-founded-status-count*)
        (eqcase (tms-well-founded-support fact)
                (in 'in)
                (out (tms-well-founded-cp-support fact))
                (else (and (eq (tms-well-founded-cp-support fact) 'in) 'in))))

;TMS-WELL-FOUNDED-SUPPORT computes the well-founded support-status derived
; from its support-list antecedent-set.

(defun tms-well-founded-support (fact)
        (increment *tms-well-founded-support-count*)
        (do ((as (sl-antecedent-set fact) (cdr as))
             (wf t))
            ((null as) (and wf 'out))
            (eqcase (tms-eval-antecedent (car as))
                    (yes (return 'in))
                    (no)
                    (else (setq wf nil)))))
```

```
;TMS-WELL-FOUNDED-CP-SUPPORT computes the well-founded support-status
; derived from its conditional-proof antecedent-set.

(defun tms-well-founded-cp-support (fact)
         (increment *tms-well-founded-cp-support-count*)
         (do ((as (cp-antecedent-set fact) (cdr as))
              (wf t))
             ((null as) (and wf 'out))
             (eqcase (tms-eval-cp-antecedent (car as))
                     (yes (return 'in))
                     (no)
                     (else (setq wf nil)))))

;TMS-EVAL-ANTECEDENT evaluates an support-list antecedent.

(defun tms-eval-antecedent (ante)
         (increment *tms-eval-antecedent-count*)
         (eqcase (tms-in (sl-antecedent-inlist ante))
                 (yes (tms-out (sl-antecedent-outlist ante)))
                 (no 'no)
                 (else nil)))

;TMS-EVAL-CP-ANTECEDENT evaluates a conditional-proof antecedent.

(defun tms-eval-cp-antecedent (ante)
         (increment *tms-eval-cp-antecedent-count*)
         (eqcase (tms-in (cp-antecedent-in-moduli ante))
                 (yes (eqcase (tms-out (cp-antecedent-out-moduli ante))
                              (yes (eqcase (support-status (cp-antecedent-fact ante))
                                           (in 'yes)
                                           (out 'no)
                                           (else nil)))
                              (else nil)))
                 (else nil)))

;TMS-IN checks a list of facts for well-founded inness.

(defun tms-in (factlist)
         (increment *tms-in-count*)
         (do ((fl factlist (cdr fl))
              (wf t))
             ((null fl) (and wf 'yes))
```

```
            (eqcase (support-status (car fl))
                    (in)
                    (out (return 'no))
                    (else (setq wf nil)))))

;TMS-OUT checks a list of facts for well-founded outness.

(defun tms-out (factlist)
       (increment *tms-out-count*)
       (do ((fl factlist (cdr fl))
            (wf t))
           ((null fl) (and wf 'yes))
           (eqcase (support-status (car fl))
                   (in (return 'no))
                   (out)
                   (else (setq wf nil)))))

;TMS-NWF-STATUS computes the (perhaps unfounded) support-status of a fact.

(defun tms-nwf-status (fact)
       (increment *tms-nwf-status-count*)
       (eqcase (tms-nwf-support fact)
               (in 'in)
               (out (eqcase (tms-well-founded-cp-support fact)
                            (in (error 'tms-nwf-status fact 'wrng-type-arg) 'out)
                            (out 'out)
                            (else 'out)))))

;TMS-NWF-SUPPORT computes the support-status of a fact from its
; support-list antecedent-set by equating 'OUT and NIL.

(defun tms-nwf-support (fact)
       (increment *tms-nwf-support-count*)
       (do ((as (sl-antecedent-set fact) (cdr as)))
           ((null as) 'out)
           (and (tms-nwf-eval-antecedent (car as))
                (return 'in))))

(defun tms-nwf-eval-antecedent (ante)
       (increment *tms-nwf-eval-antecedent-count*)
       (and (tms-nwf-in (sl-antecedent-inlist ante))
            (tms-nwf-out (sl-antecedent-outlist ante))))
```

```
(defun tms-nwf-in (factlist)
        (increment *tms-nwf-in-counts*)
        (do ((fl factlist (cdr fl)))
            ((null fl) t)
            (or (is-in (car fl)) (return nil))))

(defun tms-nwf-out (factlist)
        (increment *tms-nwf-out-counts*)
        (do ((fl factlist (cdr fl)))
            ((null fl) t)
            (and (is-in (car fl)) (return nil))))

(defun tms-install-antecedents (fact)
        (increment *tms-install-antecedents-counts*)
        (do ((as (cp-antecedent-set fact) (cdr as)))
            ((null as))
            (eqcase (tms-eval-cp-antecedent (car as))
                    (yes (let ((support (tms-findindep (cp-antecedent-fact (car as))
                                                       (cp-antecedent-in-moduli (car as))
                                                       (cp-antecedent-out-moduli (car as)))))
                               (let ((ante (new-antecedent (car support) (cdr support)
                                                           (antecedent-argument (car as))))
                                     (ante-set (si-antecedent-set fact)))
                                    (and (not (tms-ante-set-member ante ante-set))
                                         (clobber (si-antecedent-set fact)
                                                  (nconc ante-set (list ante)))))))
                    (else)))
        (do ((as (si-antecedent-set fact) (cdr as)))
            ((null as)
             (clobber (supporting-antecedent fact) nil)
             (clobber (supporting-facts fact)
                      (tms-fact-set-condense
                       (nconc (mapcan '(lambda (a) (tms-antecedent-extract a))
                                      (si-antecedent-set fact))
                              (mapcan '(lambda (a) (tms-cp-antecedent-extract a))
                                      (cp-antecedent-set fact))))))
            (eqcase (tms-eval-antecedent (car as))
                    (yes (clobber (supporting-antecedent fact) (car as))
                         (clobber (supporting-facts fact)
                                  (append (si-antecedent-inlist (car as))
                                          (si-antecedent-outlist (car as))
                                          nil))
                         (tms-antecedent-transpose fact (car as))
```

```
                        (return nil))
                    (else))))

(defun tms-nwf-install-antecedents (fact)
       (increment *tms-nwf-install-antecedents-counts*)
       (do ((as (si-antecedent-set fact) (cdr as)))
           ((null as)
            (clobber (supporting-antecedent fact) nil)
            (clobber (supporting-facts fact)
                     (tms-fact-set-condense
                      (nconc
                       (mapcan '(lambda (a) (tms-nwf-antecedent-extract a))
                               (si-antecedent-set fact))
                       (mapcan '(lambda (a) (tms-nwf-cp-antecedent-extract a))
                               (cp-antecedent-set fact))))))
           (cond ((tms-nwf-eval-antecedent (car as))
                  (clobber (supporting-antecedent fact) (car as))
                  (clobber (supporting-facts fact)
                           (tms-fact-set-condense (tms-nwf-antecedent-extract (car as))))
                  (return nil))))))

(defun tms-antecedent-transpose (fact ante)
       (increment *tms-antecedent-transpose-counts*)
       (let ((as (si-antecedent-set fact)))
            (or (eq ante (car as))
                (do ((l as (cdr l))
                     (n (cdr as) (cdr n)))
                    ((eq ante (car n))
                     (rplaca n (car l))
                     (rplaca l ante))))))

(defun tms-antecedent-extract (ante)
       (increment *tms-antecedent-extract-counts*)
       (eqcase (tms-in (si-antecedent-inlist ante))
               (yes (eqcase (tms-out (si-antecedent-outlist ante))
                            (yes (append (si-antecedent-inlist ante)
                                         (si-antecedent-outlist ante)
                                         nil))
                            (no (tms-out-extract (si-antecedent-outlist ante)))
                            (else nil)))
               (no (tms-in-extract (si-antecedent-inlist ante)))
               (else nil)))
```

```
(defun tms-cp-antecedent-extract (ante)
         (eqcase (tms-in (cp-antecedent-in-moduli ante))
                  (yes (eqcase (tms-out (cp-antecedent-out-moduli ante))
                               (yes (eqcase (support-status (cp-antecedent-fact ante))
                                            (in (error '|TMS-cp-antecedent-extract|
                                                       ante 'wrng-type-arg) nil)
                                            (out (list (cp-antecedent-fact ante)))
                                            (else nil)))
                               (no (error '|TMS CP Ante Extract OUTs| ante 'wrng-type-arg) nil)
                               (else nil)))
                  (no (error '|TMS CP Ante Extract INs| ante 'wrng-type-arg) nil)
                  (else nil)))

(defun tms-in-extract (factlist)
         (increment *tms-in-extract-counts*)
         (do ((fl factlist (cdr fl))
              (wf t))
             ((null fl) (and wf (append factlist nil)))
             (eqcase (support-status (car fl))
                     (in)
                     (out (return (list (car fl))))
                     (else (setq wf nil)))))

(defun tms-out-extract (factlist)
         (increment *tms-out-extract-counts*)
         (do ((fl factlist (cdr fl))
              (wf t))
             ((null fl) (and wf (append factlist nil)))
             (eqcase (support-status (car fl))
                     (in (return (list (car fl))))
                     (out)
                     (else (setq wf nil)))))

(defun tms-nwf-antecedent-extract (ante)
         (increment *tms-nwf-antecedent-extract-counts*)
         (cond ((tms-nwf-in (si-antecedent-inlist ante))
                (cond ((tms-nwf-out (si-antecedent-outlist ante))
                       (append (si-antecedent-inlist ante) (si-antecedent-outlist ante) nil))
                      (t (tms-nwf-out-extract (si-antecedent-outlist ante)))))
               (t (tms-nwf-in-extract (si-antecedent-inlist ante)))))

(defun tms-nwf-cp-antecedent-extract (ante)
         (cond ((tms-nwf-in (cp-antecedent-in-moduli ante))
```

```
                (cond ((tms-nwf-out (cp-antecedent-out-moduli ante))
                       (cond ((is-in (cp-antecedent-fact ante))
                              (append (cp-antecedent-in-moduli ante)
                                      (cp-antecedent-out-moduli ante)
                                      (list (cp-antecedent-fact ante))))
                             (t (list (cp-antecedent-fact ante)))))
                      (t (tms-nwf-out-extract (cp-antecedent-out-moduli ante)))))
              (t (tms-nwf-in-extract (cp-antecedent-in-moduli ante)))))

(defun tms-nwf-in-extract (factlist)
       (increment *tms-nwf-in-extract-counts*)
       (do ((fl factlist (cdr fl)))
           ((null fl) (append factlist nil))
           (or (is-in (car fl)) (return (list (car fl))))))

(defun tms-nwf-out-extract (factlist)
       (increment *tms-nwf-out-extract-counts*)
       (do ((fl factlist (cdr fl)))
           ((null fl) (append factlist nil))
           (and (is-in (car fl)) (return (list (car fl))))))

(comment TMS Dependency-Directed Backtracking System)

;TMS-CONTRADICTION is the fundamental method for declaring a set
; a set of facts contradictory.  The arguments are a contradiction
; type, which is a mnemonic symbol, a list of facts to be used as
; the support of the contradiction, and the external argument for
; the contradiction, as in TMS-JUSTIFY.

(defun tms-contradiction (ctype antes extarg)
       (increment *tms-contradiction-counts*)
       (let ((cont (tms-make-internal-fact 'contradiction "(,ctype contradiction))))
            (let ((cfact (internal-tms-fact-node cont)))
                 (tms-justify cfact antes nil extarg)
                 (tms-process-contradiction cont cfact ctype)
                 cfact)))

;TMS-PROCESS-CONTRADICTION directs the backtracking process.
; Its arguments are the contradiction name, the contradiction fact,
;  and the contradiction type.

; The theory of backtracking in this function is as follows:
;  There are 4 flavors of facts as far as the backtracker is concerned:
```

```
;  ASSUMPTIONS -- Superiorless IN facts supported by OUT facts.
;  SUSPECTS    -- OUT facts supporting ASSUMPTIONS.
;  IN-SUPPORT  -- Facts which are IN independent of any SUSPECTS.
;  OUT-SUPPORT -- Facts which are OUT independent of any SUSPECTS.
;  Both types of independent support are collected by calling TMS-FINDINDEP
;   on the contradiction and the list of suspects.

; *tms-contradiction-assumptions* is a list of pairs of assumptions and their suspects.

(defun tms-process-contradiction (cont cfact ctype)
         (let ((*tms-noted-in-facts* nil)
                (*tms-noted-out-facts* nil))
               (tms-process-contradiction1 cont cfact ctype)
               (tms-tmp-scan)
               (tms-signal-changes)))

(defun tms-process-contradiction1 (cont cfact ctype)
         (increment *tms-contradiction-count*)
         (clobber (contradiction-mark cfact) t)
         (clobber (contradiction-name cfact) cont)
         (clobber (contradiction-type cfact) ctype)
         (and (is-in cfact)
               (let ((*tms-contradiction-assumptions* nil))
                     (cond (*tms-see-contradictions-sw*
                               (terpri)
                               (princ '|The Case of the |) (prin1 ctype)
                               (princ '| Contradiction |) (prin1 cont)
                               (princ '|.|) (terpri)))
                     (cond ((tms-findchoices cfact)
                               (cond (*tms-see-contradictions-sw*
                                        (princ '|Careful consideration leads us to the following suspects:|)
                                        (mapc '(lambda (p) (tms-print '|Was it | (car p)) (princ '|?|))
                                              *tms-contradiction-assumptions*)
                                        (terpri)
                                        (princ '|The conclusion is elementary, my dear Watson:|)
                                        (terpri)
                                        (tms-print '|It was | (caar *tms-contradiction-assumptions*))
                                        (princ '|.|) (terpri)))

                               ;The following will cause truth maintenance, and will result
                               ; in one of the assumptions or choices being changed
                               (tms-contradiction-assert-nogood cfact)
                               'found-a-culprit)
```

```
                    (t (cond (*tms-see-contradictions-sw*
                              (terpri)
                              (princ '|The villian has eluded us! Take care!|)
                              (break |An Impeccable Contradiction|)))
                       'found-no-choices)))))

;CONTRADICTION-ASSERT-NOGOOD implements the nogood for a given contradiction.

; Nogoods are implemented as dependency relationships in this program,
;  so that no explict nogood checking is necessary - the truth maintenance
;  system performs that task.  Thus it is known that no former suspects
;  that are OUT at the time of a contradiction (or any other time)
;  are in conflict with a nogood set, for a nogood set would cause an
;  suspect fact to be IN.

(defun tms-contradiction-assert-nogood (cfact)
       (let ((al (mapcar 'car *tms-contradiction-assumptions*))
             (nogoodf (tms-make-internal-fact 'nogood "(nogood for ,(external-name cfact)))))
            (let ((nogood (internal-tms-fact-node nogoodf))
                  (cl (mapcar
                        '(lambda (a)
                           (let ((cname
                                   (tms-make-internal-fact
                                    'culprit "(culprit ,nogoodf ,(external-name a)))))
                             (internal-tms-fact-node cname)))
                        al)))
                 (clobber (nogood-contradiction nogood) cfact)
                 (clobber (nogood-assumptions nogood) al)
                 (clobber (contradiction-nogoods cfact)
                          (cons nogood (contradiction-nogoods cfact)))
                 (tms-cp-justify1 nogood cfact al nil "(nogood ,nogoodf))
                 (mapc '(lambda (c a)
                          (tms-justify1 c "(,nogood @(delq a (append al nil)))
                                        nil "(culprit ,nogoodf)))
                       cl al)
                 (mapc '(lambda (p c)
                          (mapc '(lambda (s)
                                   (tms-justify1 s (list c) (delq s (append (cdr p) nil))
                                                 "(suspect ,nogoodf)))
                                (cdr p)))
                       *tms-contradiction-assumptions*
                       cl))))
```

```
;tms-findchoices marks the antecedent structure of a contradiction to find the relevant choices.

; superiors-mark of a fact is
;     'YES   if some choice depends on the fact
;     'NO    if no choice depends on the fact
;     NIL    if the fact has not been marked yet.

(defun tms-findchoices (fact)
          (tms-findchoices1 fact nil)

          ;The following weeds out any subordinate assumptions spuriously included
          ; in the list due to chronological accidents in tms-findchoices1
          (setq *tms-contradiction-assumptions*
                  (mapcan '(lambda (s) (and (eq (superiors-mark (car s)) 'no) (list s)))
                          *tms-contradiction-assumptions*))

          ;Cleanup the marks made by tms-findchoices1
          (tms-findchoices2 fact)

          ;Return an indication of whether there were any assumptions.
          (not (null *tms-contradiction-assumptions*)))

(defun tms-findchoices1 (fact superiorsp)
          ;superiorsp is whether the fact has choices above
          (eqcase (superiors-mark fact)
                    (yes)
                    (no (cond (superiorsp                                    ;revisions must be propagated downwards
                                (clobber (superiors-mark fact) 'yes)
                                (mapc '(lambda (a) (tms-findchoices1 a superiorsp))
                                      (antecedents fact)))))
                    (else
                     (clobber (superiors-mark fact) (cond (superiorsp 'yes) (t 'no)))
                     (let ((out-antes (mapcan '(lambda (f) (and (is-out f) (list f)))
                                              (antecedents fact))))
                           (and out-antes
                                (cond ((not superiorsp)
                                       ;Only superiorless choices are collected.
                                       ; This may collect assumptions later judged to be subordinate.
                                       (setq superiorsp t)
                                       (push (cons fact out-antes) *tms-contradiction-assumptions*))))
                           (mapc '(lambda (a)
                                    (tms-findchoices1 a superiorsp))
                                 (antecedents fact))))))
```

```
(defun tms-findchoices2 (fact)
        (cond ((superiors-mark fact)
                (clobber (superiors-mark fact) nil)
                (mapc 'tms-findchoices2 (antecedents fact)))))

(comment Dependency-Directed Analysis Functions)

;tms-findindep collects the independent support of a fact modulo some other
; facts.  The support is returned as a pair, (list of in support . list of out support).

(defun tms-findindep (fact inmoduli outmoduli)
        (mapc '(lambda (m)
                  (clobber (findindep-mark m) t)
                  (clobber (subordinates-mark m) t))
               inmoduli)
        (mapc '(lambda (m)
                  (clobber (findindep-mark m) t)
                  (clobber (subordinates-mark m) t))
               outmoduli)
        (tms-findindep0 fact)
        (let ((support (tms-findindep1 fact)))
              (tms-findindep2 fact)
              (mapc '(lambda (m)
                        (clobber (findindep-mark m) nil)
                        (clobber (subordinates-mark m) nil))
                     inmoduli)
              (mapc '(lambda (m)
                        (clobber (findindep-mark m) nil)
                        (clobber (subordinates-mark m) nil))
                     outmoduli)
              support))

; subordinates-mark of a fact is
;    T     if the fact depends on some modulus fact
;    NIL   if the fact depends on no modulus fact

(defun tms-findindep0 (fact)
        ;subsp is whether the fact has choices below
        (cond ((subordinates-mark fact) t)
               (t (let ((subsp nil))
                         (mapc '(lambda (a)
                                   (setq subsp (or (tms-findindep0 a) subsp)))
```

```
                    (antecedents fact))
                  (clobber (subordinates-mark fact) subsp)
                  subsp))))

(defun tms-findindep1 (fact)
       (cond ((findindep-mark fact) '(nil . nil))
             ((null (subordinates-mark fact))
              (clobber (findindep-mark fact) t)
              (eqcase (support-status fact)
                      (in (cons (list fact) nil))
                      (out (cons nil (list fact)))))
             (t (clobber (findindep-mark fact) t)
                (let ((si (mapcar 'tms-findindep1 (antecedents fact))))
                     (cons (mapcan 'car si) (mapcan 'cdr si))))))

(defun tms-findindep2 (fact)
       (cond ((findindep-mark fact)
              (clobber (findindep-mark fact) nil)
              (clobber (subordinates-mark fact) nil)
              (mapc 'tms-findindep2 (antecedents fact)))))
```