TR-I-0074

# The ATMS Manual

Version 1.1

ATMS 説明書

1.1

John K. Myers

真龍主·星音

February 23, 1989

*Abstract*

This manual presents user documentation for the ATR Interpreting Telephony Research Laboratories LISP ATMS. The ATMS, or "Assumption-based Truth Maintenance System", is a data-base that is able to represent and store concepts (*atms-nodes)* and constraints between these concepts (*implications*). Some concepts are *assumed* by the user. Depending upon whether these *assumptions* are BELIEVED or NOT BELIEVED, different multiple *possible worlds* known as *environments* are set up. In addition to storing data and implications, the ATMS performs "*truth maintenance*"--if any concept or set of concepts becomes inconsistent, the effects of this change are propagated through implications to other concepts that might become inconsistent, thus "maintaining the truth" of the system as a whole. Also, the ATMS offers *explanations* as to why concepts are believed in a particular possible world.

The ATMS was developed to enable ATR to have modifiable LISP source code to an assumption-based TMS. This was required for performing plan recognition using an ATMS.

The ATMS is a general-purpose system that can work with any application that requires an assumption-based truth maintenance system; it is not limited to plan-recognition applications. The current version of the ATMS supports various assertion, query, explanation, and output functions for the user.

# Acknowledgment

This research was supported by ATR Interpreting Telephony Research Laboratories. I would like to express my gratitude to Dr. Akira Kurematsu, President of the Interpreting Telephony Research Laboratories, for the invitation to come to ATR, for providing the support that enabled this research to be done, and for the interest shown in this work. I am also grateful to Mr. Teruaki Aizawa, Head of the Natural Language Understanding Department, for efficient management and encouragement, and for providing the time necessary to explore some of the important, more basic research areas. In addition, I am grateful to Mr. Kiyoshi Kogure, who helped significantly in many of the small details in bringing me to ATR and getting me acclimated to a new culture, besides providing useful research discussions. Thanks are also due to Mr. Hitoshi Iida, who provided helpful suggestions as to the technical direction of this work. I would also like to acknowledge the technical contributions of Mr. Remi Zajac, Mr. Martin Emele, and Dr. Gayle Sato, who provided useful discussions and helpful ideas for improvement. Finally, I would like to acknowledge the friendliness and the helpfulness of the rest of the people in the Natural Language Understanding Department.

# Contents

# List of Figures

# 1 Introduction

This manual describes the ATR Interpreting Telephony Research Laboratories' ATMS (*A*ssumption-based *T*ruth-*M*aintenance *S*ystem), version 1.1. The ATMS, or "Assumption-based Truth Maintenance System", is a database that is able to represent and store concepts (*atms-nodes*) and constraints between these concepts (*implications*) that occur in different possible situations at the same time, known as *multiple simultaneous possible worlds.* Worlds are set up by "assuming" a concept–if the assumed concept is believed, this contributes to forming one possible world, whereas if the assumption is disregarded (not believed), this forms a different possible world.

In addition to storing data and implications, the ATMS performs "*truth maintenance*"—if any concept or set of concepts becomes inconsistent, the effects of this change are propagated through implications to other concepts that might become inconsistent, thus "maintaining the truth" of the system as a whole. Although a valid possible world can only contain consistent concepts within itself, there is nothing preventing the set of concepts in one possible world from being inconsistent with the concepts in another world.

This manual starts out with a glossary, which defines the technical terms that are used. Next is a command explanation section that gives a breakdown of all the commands used in the system, grouped by function. After this, the manual starts with an introduction to what an ATMS does, a brief non-technical look at truth values, and a discussion of the types of data structure objects that the system works with to represent problems. These objects can be put together to form relationships and logical operations; the method of representing the basic logic operations is discussed next. Following this, the slightly more complex data structures required for interpretation are discussed.

The next section gives a technical discussion of the meaning of various truth values used by the system. Following this, there is a theoretical discussion about different types of knowledge, and how the ATMS can represent these various types.

This is required to understand the next section, which is a discussion of the actual, practical use of the ATMS system, referring to specific commands. The methods of setting up the system, setting up the problem, exploring the problem, and interpreting the results are discussed. After this, concrete examples that demonstrate different aspects of the use of the system are

discussed in detail.

The manual concludes with the appendicies. Included here is a discussion of the implementation of the system, and an alphabetical index of the commands used by the system.

The first-time reader should probably briefly glance at the glossary and the command explanation section, before going immediately to the introductory explanations and reading them in order. After reading the technical discussion of the different types of truth values, the reader can go back to the command explanation section and read it again in depth, to get a good understanding of the system. The types of knowledge section should be read before the sections on working with the ATMS and the examples. The implementation appendix, although useful, is not required to understand how to run the system. The manual contains an alphabetical glossary and the command explanation system at the front, and the command dictionary at the back, for easy reference.

This manual is intended for the naive user who has never worked with an ATMS before. The user should be able to read the manual, run the examples, and afterwards understand how to use the ATMS. However, some familiarity with basic computer science concepts would be helpful. Also, it is assumed that the reader is familiar with the LISP computer language's syntax. The manual is designed to be self-contained; no other reference source is needed to run the system.

# 2 Glossary

In the definitions in this section, *italics* represent terms that are defined elsewhere under other definitions; **bold face** represents the term itself. Underlining is occasionally used for emphasis.

**Antecedent** The IF part of an IF-THEN concept. Each *implication* can have one or more antecedents.

**Assertion** A concept. A "fact", that will either be believed or not believed. Assertions are represented by ATMS-nodes. Assertions can be sentences or data-structures in the *user system*, but they are treated as atomic by the ATMS.

**Assume** The action of augmenting an ATMS-node by turning it into an assumption.

**Assumption** A concept that the user system thinks is basic or influential. **Assumptions** are concepts on which other concepts depend. Also, the data-structure that represents this concept. **Assumptions** are ATMS-nodes that have been specially marked, by *assuming* them. Typically, assumptions will *justify* a network of ATMS-nodes. A single assumption can be BELIEVED or NOT BELIEVED. In fact, it takes on both of these values simultaneously; this serves to split the knowledge base into two different [sets of] *possible worlds.*

**ATMS-node** The basic atomic data structure for the ATMS system. An ATMS-node stores a single concept (or *assertion*).

**Believed** A truth value for a concept (ATMS-*node*) in a particular *possible world* (*context*). BELIEVED corresponds to TRUE in a trinary TRUE/FALSE/UNKNOWN logic. See *not believed.*

**Characterizing Environment** A **characterizing environment** is a *consistent*, complete, *minimal* environment that characterizes (uniquely represents) a context. Since all valid environments that are not created by the user are always characterizing environments, this concept may be ignored. See *environment* instead.

**Concept** An idea about something, represented by an ATMS-*node* or an *implication.*

**Conjunction** A logical AND. If all of the items in a conjunction are believed, then the conjunction as a whole is believed.

**Consequent** The THEN part of an IF-THEN concept. Each *implication* has one consequent.

**Consistent** A *context* is **consistent** if it is not *inconsistent.* Conceptually, a possible world is consistent if all the things that are believed in that possible world can all be believed at the same time.

**Context** The set of all BELIEVED nodes that are implied by an environment's assumptions. An environment is only a set of assumptions, whereas a context consists of those assumptions plus *all* ATMS-nodes that are directly or indirectly implied by those assumptions (including all premises), following all active implication chains forward as far as possible. A context is an entire possible world, including all the concepts implied by it.

If a context includes the `*nogood-node*`, that context is inconsistent.

**Constraint** A concept that rules out the possibility of something happening, i.e. several specific *concepts* occurring at the same time. That is, it states that these concepts taken together are *inconsistent.* **Constraints** are implemented in the ATMS system by *implications.*

**Contradiction** A contradiction is a set of concepts that cannot all be BELIEVED at the same time. See *inconsistent.*

**Deletion** Physically removing an *item* from the *knowledge base.* The curent system cannot individually delete items; it can only *retract* them. See *retraction.*

**Disjunction** A logical OR. If any one or more of the items in a disjunction is believed, then the disjunction as a whole is believed.

**Disregarded** This means, Not used by the system. Another name for *Not Believed.*

**Environment** A data structure that stores a list of believed assumptions. An environment represents and is the symbol for a *possible world*. An environment implicitly implies a *context*. An environment can be *consistent* or *inconsistent*.

**Implication** A logical form, consisting of the *conjunction* of a number of *antecedents*, and a single *consequent*. If, in any one possible world, all of the antecedents are BELIEVED, then this implies that the consequent must be BELIEVED as well. The antecedents imply the consequent. An "implication" is both this concept, and the name of a data structure that represents this concept.

Implications can have associated data attached to them that explain (to the user system) why this implication is valid. This can simply be the name of the implication, or a user system representation of the rule that this implication represents, etc.

**In** A truth value for a *concept* (ATMS-*node*) taken over the set of all known *possible worlds* (*contexts*). If the ATMS-node is BELIEVED in at least one known, consistent context, then it is IN. See OUT.

**Inconsistent** A *context* is **inconsistent** if it includes the `*nogood-node*`. Conceptually, a possible world is inconsistent if it has a thing that cannot be believed, or if there are things in that possible world that cannot be believed together. Inconsistencies (*contradictions*) are asserted into the ATMS by the *user system* by using the `(nogood)` or the `(nogood-set)` commands.

The system only uses the inconsistencies that it is told about; there are no implicit inconsistencies. In particular, all negatives have to be expressed explicitly.

**Invalid** *Inconsistent.*

**Item** An instantiation of any data structure, including an environment, an ATMS-node, an implication, etc.

**Justification** A justification is actually the same as an *implication*, but the conceptualization is different. A believed ATMS-node that is not an assumption must have at least one implication that **justifies** why this

node is believed. The node is the *consequent* of the justification, and the node is justified by the *antecedent* nodes. All of the antecedent nodes must be believed in order for the nodes to "actually justify" the consequent; otherwise, they simply "potentially justify" the consequent. The justification is the link between the antecedents and the consequents. A justification is both this concept, and an alternative name for the implication data structure that represents this concept.

A justification can have associated data attached to it that explains the reason behind that justification. This could be a name, or some other concept relevant to the user system.

**Knowledge Base** The sum total of assertions that have been made to the system. The contents of the ATMS system, looked upon as a data-base that represents knowledge.

**Label** A set of *environments* attatched to a node. Each environment is *consistent*, and the node is BELIEVED in each environment. The set is complete but *minimal*; thus, larger (subsumed) environments having no new information will not be listed.

**Minimal** A label is **minimal** if it contains the smallest possible significant environments. Technically, a set of environments is minimal when no environment in the set is subsumed by another environment in the set. Because label environments consist of sets of assumptions that justify a node's concept, maintaining a minimal label stores only the assumptions that are truly relevant.

**Node** An ATMS-node, Assumption, or Premise.

**Nogood** A loose term that technically means *inconsistent* when applied to an environment, but can also mean *OUT* (or even sometimes, incorrectly, *not believed*) when applied to a node. When an environment becomes **nogood**, there is no way to reverse this change.

**Nogood-Node** A special *node* used by the system to embody and represent the concept of *nogood* or *inconsistency*.

**Not Believed** A truth value for a concept (ATMS-*node*) in a particular *possible world* (*context*). NOT BELIEVED corresponds to UNKNOWN in

a trinary TRUE/FALSE/UNKNOWN logic. See *believed*. Other ways of thinking about NOT BELIEVED include DISREGARDED, or NO OPINION. Note that NOT BELIEVED is not the same as FALSE; there is no way to explicitly represent FALSE using an ATMS.

**No Opinion** NOT BELIEVED.

**Out** A truth value for a concept (ATMS-*node*) taken over the set of all known *possible worlds* (*contexts*). If the ATMS-node is NOT BELIEVED in all known, consistent contexts, then it is OUT. See IN.

**Possible World** Something that could be happening. An intuitive conceptualization of an *environment* and its *context*. A self-consistent set of assertions that are all *believed*.

**Premise** A concept that is considered to be always true, no matter what. Technically, a premise is BELIEVED in all possible worlds. A premise cannot be retracted.

**Retraction** Taking an assertion back; no longer believing it. Retraction essentially consists of making an assertion NOT BELIEVED in all considered possible worlds. This can be done permanently by setting the node representing the assertion to directly imply NOGOOD; or, it can be done conditionally by having the node, and an assumption that the node is really retracted, together imply NOGOOD. Alternatively, retraction can be accomplished by not considering any possible worlds in which the node is BELIEVED. Retraction differs from deletion in that deletion physically removes the node, whereas retraction simply removes the use of the node by the system. Items cannot be deleted in the current system.

**Subsumed** An environment is **subsumed** by another environment if it is a larger *superset* of the beliefs of that environment. For instance, environment 1 contains believed concept A, "The computer has crashed", while environment 2 contains believed concept A plus believed concept B, "There is a pen on the table". Environment 2 is **subsumed** by environment 1. To obtain a *minimal* representation, subsumed environments are eliminated from labels.

**Truth Maintenance** The problem of maintaining the correct truth value of assertions that are based on the truth value of other assertions. Since there can be long chains of truth dependencies, a particular truth value typically propagates through many nodes.

**Truth Maintenance System (TMS)** A computer system that performs truth maintenance. There are several kinds. An *Assumption-based Truth Maintenance System* allows the representation of multiple possible worlds simultaneously, whereas most other kinds can only represent a single possible world.

**Unknown** See NOT BELIEVED.

**User System** The **user system** is a computer system outside of the ATMS, that uses the ATMS to help solve its problems. The user system will have data structures and information that the ATMS knows nothing about. The ATMS stores data for the user system, and reports answers to it.

**Valid** Not *inconsistent.*

**World** See *possible world.*

# 3 Data and Command Explanation

This section presents a description of the system's commands. These are arranged by the type of command.

## 3.1 System Data Types

There are five explicit major kinds of data in the ATMS system. These are:

**ATMS-node** A node. Otherwise known as a Concept, a Statement, or (sometimes, depending upon the usage) an Assumption.

**premise** A node that is always true. It does not have its own kind of data structure. Premises have the empty environment (#0) as their label.

**assumption** A fundamental node that is used to justify other concepts. Assumptions are both BELIEVED and NOT BELIEVED. They are used for environments.

**implication** An AND GATE structure between nodes. Takes many antecedents and one consequent. If all the antecedents are IN, then the consequent is IN. Also known as a Justification, a Constraint, or an Inference.

**environment** A set of assumptions. Each assumption in the environment is BELIEVED under that environment. Also known as a Possible World, Assumption Set, or Consistency Set.

## 3.2 Creation Commands

These are the basic commands. They are the ones used most often by the user system.

**(reset-atms)** Clears the system out. Expunges all previously-defined ATMS-nodes, assumptions, premises, implications, and environments. Automatically initializes Node# 0 as the NOGOOD-NODE, and Environment# 0 as the Truth Environment.

**(atms-node data)** Constructs and returns an ATMS node representing the given information. The nodes are numbered serially. Note: Node 0 is always the NOGOOD-NODE.

**(premise data)** Constructs and returns a Premise node storing the given information.

**(assumption data)** Constructs and returns an Assumption node storing the given information.

**(implication consequent-node data antecedent-node1 A2 ...)** Constructs and returns an implication.

**(justification consequent-node data antecedent-node1 A2 ...)** Same as implication.

**(inference consequent-node data antecedents)** Same as implication. The "inference" terminology is supported but not encouraged; use "implication" or "justification" instead.

**(nogood node1)** Builds a justification from the node to `*nogood-node*`. This is the standard method of entering contradictions, or in other words permanently making the node's data false. This function can also be called with a sequence of nodes, in which case each node in the sequence is set to NOGOOD.

**(nogood-set node1 node2 ...)** Builds a justification to `*nogood-node*` based on the *conjunction* of the given nodes. Standard method of entering contradictions. Note carefully that `(nogood-set )` of a set of nodes, which contradicts the AND of the set, is not the same as `(nogood )` of each of the members of the set, which contradicts the OR of the set.

## 3.3 Modification Commands

There is no way to modify an implication once it has been created. There is no way to retract the action of turning a node into a premise or an assumption.

All user *data* that the system stores can be modified using the `setf` function called on the data accessor function.

**(presume-this-node node)** Turns an ATMS-node into a premise. Technically, overwrites the label with the single, empty environment `*truth-env*`.

**(premise-this-node node)** Turns an ATMS-node into a premise. Same as `(presume-this-node)`.

**(assume-this-node node)** Turns an ATMS-node into an assumption. (Technically, justifies the node with a new assumption-tag whose data contains the node.) Returns the node. Typically used only for effect. Of course, the user should not call this on nodes that are already assumptions or premises.

## 3.4 Deletion Commands

There are no individual deletion commands for the system. Concepts can be retracted, but they cannot be deleted without resetting the entire system.

**(reset-atms)** Clears the system out. Expunges all previously-defined ATMS-nodes, assumptions, premises, implications, and environments. Automatically initializes Node# 0 as the NOGOOD-NODE, and Environment# 0 as the Truth Environment.

## 3.5 User Query Commands

**(explain-nodes)** Runs explain-node on all the nodes.

**(explain-node node)** Prints out environments in which node is IN.

**(env-nogood-p env)** Tests whether env is nogood.

**(IN-p node)** Tests whether node is IN. Returns a list of consistent environments entailing the node (the label) if the node is IN; returns nil if the node is OUT. This is the recommended function to use when tracing a node with a user-program.

**(OUT-p node)** Tests whether node is OUT. Returns T if OUT, NIL otherwise.

**(atms-node-p node)** Tests whether object is an ATMS-node or not. Note: assumptions and premises are also ATMS-nodes.

**(premise-p node)** Tests whether object is a premise or not.

**(assumption-p node)** Tests whether object is an assumption or not.

**(implication-p imp)** Tests whether object is an implication or not.

## 3.6 User Output Commands

**(print-nodes)** Prints a list of all the nodes, and their data.

**(print-assums)** Prints a list of all the assumptions, and the corresponding nodes.

**(print-implics)** Prints a list of all the implications, including assumption justifications.

**(print-envs)** Prints a list of all the environments.

**(print-atms)** Dumps everything. Use this to get used to the system.

**(print-node node)** Individual item printing functions.

**(print-assum assum)** Prints a single assumption.

**(print-implic implic)** Prints a single implication.

**(print-env env)** Prints a single environment.

**(print-significant-envs env-list)** Prints the significant (non-subset, valid) environments from a given list. Defaults to all the known environments if given no argument.

**(print-sig-envs env-list)** Prints the significant (non-subset, valid) environments from a given list. Defaults to all the known environments if given no argument.

## 3.7 User Access Commands

### 3.7.1 Number Accessor Functions

Each object is given an ID number to distinguish it. Calling these functions with the number returns the object.

**(Node# n)** Accessor functions for ATMS-nodes. Given its ID number, these functions return the node.

**(ATMS-Node# n)** Same as `(Node# n)`.

**(Premise# n)** Accessor function for premises. Since premises are really ATMS-nodes, this is the same as Node#.

**(Assum# n)** Accessor function for assumptions.

**(Assumption# n)** Accessor function for assumptions.

**(Implic# n)** Accessor function for implications.

**(Implication# n)** Accessor function for implications.

**(Just# n)** Accessor function for implications.

**(Justification# n)** Accessor function for implications.

**(Env# n)** Accessor function for environments.

**(Environment# n)** Accessor function for environments.

### 3.7.2 ID Accessor Functions

These functions return the ID number for the given object.

**(atms-node-ID node)** ID number function for nodes.

**(premise-ID node)** ID number function for premises. Same as `(atms-node-ID)`.

**(assumption-ID assump)** ID number function for assumptions. Returns NIL if not an assumption.

**(implication-ID implic)** ID number function for implications.

**(justification-ID just)** ID number function for implications.

**(environment-ID env)** ID number function for environments.

### 3.7.3 Data Accessor Functions

These functions return the user data contained in the given object.

All user data that the system stores can be modified by using the `setf` function called on the data accessor function.

**(atms-node-data node)** Returns the data stored in a node.

**(premise-data node)** Returns the data stored in a premise.

**(assumption-data assum)** Returns the data stored in an assumption.

**(implication-data impl)** Returns the data stored in an implication.

**(justification-data just)** Returns the data stored in an implication.

## 3.8 Context Commands

**(context env)** Returns a list of the nodes in an environment's context, including the ATMS-nodes, the assumptions, and the premises. Works even if the context is invalid. This is an expensive function to call.

**(in-context-p node env)** If the given node is in the given environment's context, returns a (usually smaller) characterizing environment describing why that node is believed. Otherwise, returns nil.

**(in-world-p node env)** Same as `in-context-p`.

## 3.9 Environment Commands

### 3.9.1 General Environment Functions

**(env-assums env)** Returns a list consisting of the assumptions that are BELIEVED in a given environment. Does not check whether environment is inconsistent or not. Note that more, derived ATMS-nodes will be believed under this environment (in the environment's context), than are returned in this function.

**(nogood-p env)** Returns T if given environment is NOGOOD (INCONSISTENT), nil otherwise. An environment is NOGOOD if the `*nogood-node*` is BELIEVED because of it (i.e., in its context). Same as `inconsistent-p`.

**(inconsistent-p env)** Returns T if given environment is NOGOOD (INCONSISTENT), nil otherwise. An environment is NOGOOD if the `*nogood-node*` is BELIEVED because of it (i.e., in its context). Same as `nogood-p`.

**(nogood-env env)** Forces the given environment (and all of its supersets) to become NOGOOD. Calls `nogood-set` on the (conjunction of the) set of assumptions composing the environment. In general, this should be used only because of higher-level knowledge not part of the knowledge represented in the ATMS.

### 3.9.2 System Environment Functions

**(node-label node)** Returns a list of the minimal environments under which the given node is believed.

**(node-envs node)** Returns a list of the minimal environments under which the given node is believed.

**(all-node-envs node)** Returns a list of *all* of the known consistent environments under which a given node is believed. This function is slightly expensive.

**(OR-env env1 env2)** Returns an environment consisting of the union of the assumption sets from the two given environments. This may be inconsistent, even if both of the previous two are not. Such an environment might not be a characterizing environment.

**(significant-envs env-list)** Returns a list of environments where subset and inconsistent environments have been eliminated. Defaults to using `*environments*`, all of the known environments, as input if no argument is given.

**(sig-envs env-list)** Returns a list of environments where subset and inconsistent environments have been eliminated. Defaults to using `*environments*`, all of the known environments, as input if no argument is given.

**(dont-use assum-list env-list)** Returns a list of environments where environments containing any of the given assumptions have been deleted.

**(dont-use-nodes nodes envs)** Returns a list of environments where environments whose context contains any of the given nodes have been deleted. A rather expensive function.

### 3.9.3 User Environment Functions

**(create-env assum-list)** Creates a new environment for the system to keep track of and follow, consisting of the set of all the assumptions in the given assumption-list. Returns the environment. Returns the old environment instead of creating it if previously there. Currently returns nil if new environment is nogood. If an ATMS-node in the assumption list was not in fact previously an assumption, it is *assumed* by this function. Note that this side-effect should be used with care.

**(find-env assum-list)** Finds and returns an existing environment. Returns `nil` if it did not exist previously. Does not create any new environments. This is a fast function.

**(add-assums-to-env old-env assumptions ...)** Creates (if necessary) and returns a new environment consisting of the assumptions of the old environment plus the new series of assumptions. Currently returns nil if new environment is nogood. Does not affect the old environment.

**(subsumed-by-p larger-env smaller-env)** Tests to see whether larger-env is subsumed by (is a superset of) smaller-env. Returns `T` if subsumed, `nil` otherwise. Extremely fast.

**(characterizing-env env)** Returns the characterizing environment of the given environment (possibly itself). Returns nil if inconsistent.

## 3.10 Explanation Commands

**(why-envs node)** Returns a list of the consistent environments under which (in whose context) this node is BELIEVED.

**(why-env-assums node)** Explains the different assumption sets that this node is BELIEVED in. Instead of returning a list of environments justifying this node, like `why-envs`, this function returns the environments' assumption sets, in the form of a list of lists of assumptions.

**(why-nodes node env)** Explains the contributing immediately preceding nodes that make the given node believed under the given environment. Returns a list of all the believed nodes that *directly* justify the given node in the given environment's context.

**(why-implications node env)** Explains the contributing immediate implications that make the given node believed under the given environment. Returns a list of all the active implications that *directly* actually justify the given node in the given environment's context. Does not return implications that indirectly justify the node, or potentially justify the node but are inactive. Returns the system-generated justification for an assumption.

**(why-assumptions node env)** Explains the assumptions that directly or indirectly contribute to the given node under the given environment. Returns a list of all the BELIEVED assumptions that justify the node in the environment's context.

**(why-nogood-nodes env)** Explains the immediately preceding nodes that contribute to making the `*nogood-node*` believed under the given environment. The environment should be inconsistent.

**(why-nogood-implications env)** Explains the implications that immediately contribute to the `*nogood-node*` under the given environment. The environment should be inconsistent. Returns a list of the active implications that actually justify the `*nogood-node*` in the environment's context.

**(why-nogood-assumptions env)** Explains the assumptions that directly or indirectly contribute to NOGOOD under the given environment. The environment should be inconsistent. This is a very useful function, as it returns only the mutually conflicting assumptions that are causing the problem with an inconsistent environment.

## 3.11 System Activity Commands

**(install-action node action)** Installs the command (`action`) into the given node. If the given node becomes IN, (i.e., believed in *any* valid context), the given action command is executed.

## 3.12 Significant Variables

**OS** This variable holds the Output Stream for the print functions. Default is T, meaning standard screen output stream.

**use-uniquification** This flag tells whether ATMS data is treated as being unique (under `equal`) or whether it can be duplicated. If unique, (`atms-node data`) and similar functions will return a previously created node instead of creating a new one. Default is T.

***environments*** This variable stores a list of all (both valid and inconsistent) of the environments known to the system.

***nogood-node*** This variable stores the special NOGOOD node. This node is allocated on reset. Note that (`Node# 0`) also returns this node.

***truth-env*** This variable stores the empty environment. This environment's context contains all the premise nodes; it is always true.

***atms-nodes*** This variable stores a list of all the ATMS-nodes known to the system. This includes the assumptions and the premises.

***assumptions*** This variable stores a list of all the assumptions known to the system.

***premises*** This variable stores a list of all the premises known to the system.

***implications*** This variable stores a list of all the implications known to the system. Each assumption internally generates an implication; these are included as well.

***atms-node-count*** The number of ATMS-nodes, including those that have been turned into assumptions or premises, known to the system.

***assumption-count*** The number of assumptions known to the system.

***environment-count*** The number of environments known to the system.

***premise-count*** The number of premises known to the system.

***implication-count*** The number of implications known to the system.

***initial-assumption-limit*** This number gives a soft limit on the number of *assumptions* that the system can store. It is used to determine the initial size of the assumption-bit-vector assigned to each environment. It must be set before calling `(reset-atms)`. Set this to the reasonable maximum number of assumptions expected to be handled by the system. This number affects memory allocation, paging, and performance. Default is 200.

***incremental-assumption-size*** This number tells how much the system's bit-vector size is increased during the next growth cycle. See `*initial-assumption-limit*`. This number indirectly affects memory allocation, paging, and performance. Default is 50.

**geometric-limit-increase** This flag tells whether *incremental-assumption-limit* doubles after every expansion (geometric increase) or stays constant (arithmetic increase). This number indirectly affects memory allocation, paging, and performance. Default is T.

## 3.13 System Flag Variables

***watch-atms*** This flag makes the system print out a notification each time an item is created. Default is T.

***debug-atms*** This flag makes the system print out debugging information. Default is nil.

***watch-enlarge*** This flag makes the system print out a message when the system enlarges the bit-vector arrays for assumptions. Default is T.

***print-data*** When this flag is T, the print functions print out the data inside nodes and assumptions. When it is nil, the print functions only print out a numbered node. Set this to nil when very long data is stored in nodes. Default is T.

# 4 What does an ATMS do?

An *A*ssumption–based *T*ruth–*M*aintenance *S*ystem, or ATMS, is basically a special kind of data-base that stores pieces of data. This data is special in that each piece has a truth-value, *believed*, or *not believed*. The ATMS remembers whether each piece of data, or *ATMS-node*, is believed or not. However, the truth-value of some ATMS-nodes may depend upon whether other ATMS-nodes are believed or not. So, in this case, if one node becomes believed, it could affect other nodes in the data-base. Similarly, if one node becomes disbelieved, perhaps other nodes will become disbelieved also. Naturally, in a network of interrelated data, such changes will propagate on to still other nodes and could be far-reaching. Remembering which nodes are believed and which are disbelieved at the current time is the job of a *T*ruth–*M*aintenance *S*ystem (TMS).

An Assumption-based TMS is a special kind of TMS that does not deal with only one possibility, but represents several possibilities at the same time. That is, in one possibility, some nodes could be believed, while in another possibility, they might not be believed. These different possibilities are called *possible worlds*.

A possible world is set up by "assuming" a concept. *Assuming* a concept states that the user is willing to possibly believe the concept by itself, without regard to any further justifications of the concept. Thus, assumptions are basic, and are normally not justified by other concepts. Rather, assumptions are used *to* justify other concepts, and to imply conclusions.

However, an assumption is not an absolute declaration of truth. The system maintains the possibility that the assumption could be true, or it could be disregarded and not believed. In fact, the system represents both possibilities simultaneously. Thus, assuming a node creates two different possible worlds–one in which the node's concept is believed, and one in which it is not believed.

In fact, since there are usually many assumptions, at the time a new node is assumed there are already many different possible worlds in existence. So, actually, a new assumption acts to create two different *sets* of possible worlds–one set consisting of all of the current possible worlds as they stand (NOT BELIEVING the new assumptions), and one set consisting of all the current possible worlds plus the new assumption (BELIEVING the new assumption).

The set of possible worlds can therefore theoretically be the power set

of the set of assumptions. However, in practice, the ATMS only represents those possible worlds that are significant, i.e. different from each other (not *subsumed*). This typically tends to be a much smaller number.

A single possible world has many concepts, represented by nodes. The entire set of nodes that are believed in a given possible world is called a *context*. However, since this is very large, a possible world is represented by a "characterizing environment" (or *environment* for short), consisting of a set of all the believed *assumptions* in that environment. Since the assumptions are basic, the assumptions determine all of the nodes that are believed in that possible world. Thus, the context can be uniquely derived from the characterizing environment.

All the assumptions listed in an environment are believed by that environment. All the assumptions not listed in an environment are disregarded. As will be discussed later, disregarding (not believing) an assumption is not the same as believing it to be false; this is important to remember.

The ATMS can be run directly from a terminal. However, typically another computer program, i.e. a *user system*, will interact with the ATMS by setting up the problem and interpreting the results. A general-purpose inference-engine user system, known as FLAIL, has been implemented at ATR and can be used for this.

The designer of the user system must determine the type of data that the ATMS is going to represent, and the relationships between different concepts. The user system sets up the problem by entering concepts into the ATMS, in the form of ATMS-nodes, premises, and assumptions. In addition, relationships between concepts are entered in the form of networks of implications, and problem constraints are entered by specifying mutual inconsistencies ("nogood" sets). It is the responsibility of the user system to determine these concepts, relationships, and constraints. Once a problem has been set up, the user system can query the ATMS system and interpret the results, using the explanation facilities. Typically, the user system will use the ATMS in an interactive manner, incrementally adding new nodes and implications and then querying the results.

The ATMS thus acts as a large truth-maintaining data-base that permits representation and exploration of multiple possible worlds at the same time. The system reports what concepts are currently believed or not believed in any particular world. However, the ATMS must work with a finite space of alternatives. Also, the data given to the ATMS by the user system should be

definite concepts–that is, constants and instantiated variables. The system can then perform implicit searches among the given alternatives, representing which sets of alternatives are possible and which are inconsistent under the given constraints.

# 5 An Introduction to Truth Values

Some truth maintenance systems use the truth values TRUE and FALSE. Because an ATMS is based on assumptions, an ATMS uses the truth values BELIEVED and NOT BELIEVED (or, "NO OPINION").[1] The user assumes that something could be believed, but this is just an assumption–the possibility that it is not believed is explored as well.

Besides these truth values, another important set is the pair IN and OUT, which are basically slightly stronger versions of (sometimes) BELIEVED and (always) NOT BELIEVED, respectively.[2]

This brief introduction provides a working definition only; truth values will be examined in more depth later on in this manual.

# 6 What's Inside an ATMS?

In an ATMS, there are two main kinds of objects–*nodes*, and *implications*. A node is a basic unit of data, representing one item to be remembered. Implications are the connections between nodes. There are a few different kinds of nodes that have slightly different meanings, but there is only one kind of implication. Together, the nodes and the implications are used to build the data structure that represents the problem.

In addition to these objects, there is also *environments*, which is a way of thinking about nodes and grouping them together. Nodes and implications are discussed in the following subsections; environments will be discussed later.

In general, because ATMSs were originally developed at many different institutions, there are a number of different names for the same concept. Therefore, all of the different names for the same type of object will be

[1] In any one possible world.

[2] For all currently known possible worlds taken together.

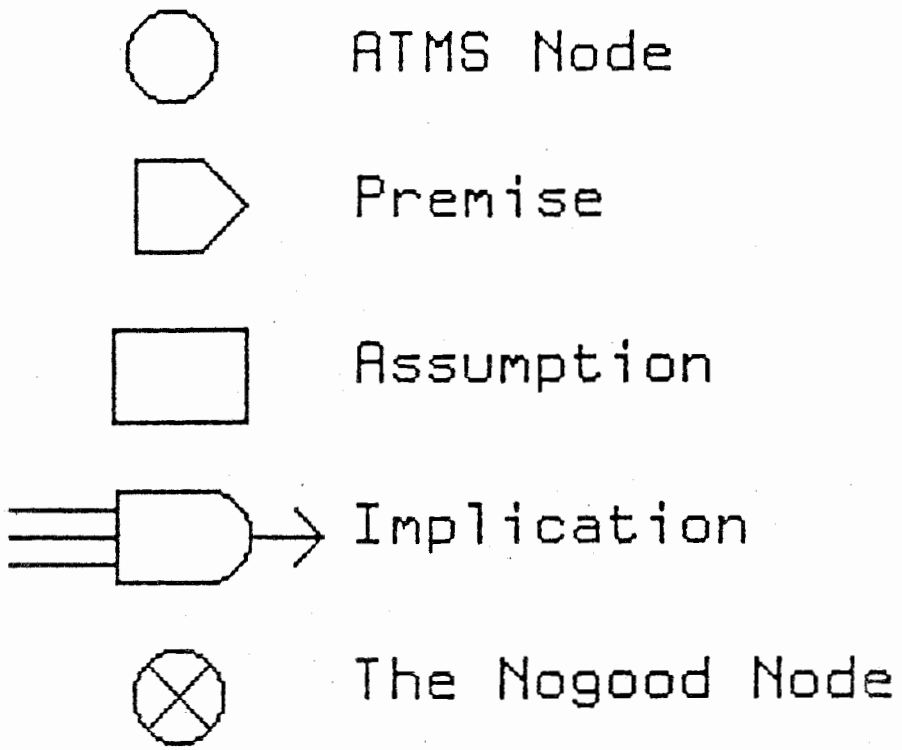



Figure 1: Graphical Conventions for depicting the ATMS.

presented here, so that the reader can understand this system in relation to other systems in the literature.

## 6.1 ATMS Nodes ○

An ATMS node is the basic unit of data of the system. It stores a particular statement that the system will either believe as true, or disregard as unknown. The system never actually uses this data; the system only uses what it is told are the constraints between this node and other nodes. The data is treated as atomic. The system only reports this data to the user when asked, making no other use. Thus, the user is free to store absolutely anything that he or she desires into the data in a node. In typical usages, this will actually be a list, an entire tree, or a sentential fact.

An ATMS node is created by using the function `(atms-node` *data*`)`. The ATMS system ensures that the data stored in a node is unique; thus if this function is called a second time with the same data, the same node is conveniently returned, instead of creating a new node.[3]

A node can be made special by either turning it into a *premise*, or turning it into an *assumption*, as will be explained next.

A node will take on one of the truth values BELIEVED or UNKNOWN (NOT BELIEVED) inside any one possible world. A node by itself, one that isn't an assumption, or isn't justified by other nodes, has the default value UNKNOWN. Thus, it is necessary to introduce assumptions or premises into a network in order to have nodes that are believed.

Once a node is created, it can never be deleted, without resetting the entire ATMS. (However, it can be "effectively deleted" or *retracted*, by creating an implication from it to the NOGOOD node. Note that a direct implication with no other antecedents constitutes a permanent retraction, whereas if other nodes are used for antecedents, this constitutes a conditional retraction. Also, if the node is used as an assumption it can be disregarded by the user, by not using any environments containing that node.)

Nodes are sometimes called *concepts*, *statements*, or *facts*. De Kleer also sometimes calls these *derived nodes*.

[3]This feature can be turned off by `(setq use-uniquification NIL)`.

## 6.2 Premises 

A premise is a special kind of node that is always true. That is, the user knows (from higher knowledge) that this particular node will always be BELIEVED in all valid possible worlds. The system uses this knowledge to propagate belief and inconsistency to other nodes that are not premises.

Premises are usually specified beforehand, at the time of their creation, using the command `premise`. However, it is possible to take an existing ATMS node and turn it into a premise, using the command `premise-this-node`. It is not possible to retract a premise once it has been made, or to turn it back into an ordinary node.

It is a system error to set up a network where a premise is forced to derive the NOGOOD node. Since a premise normally is used in conjunction with other nodes or assumptions as the antecedents to an implication, this does not present a problem; those nodes can become inconsistent, while the premise remains true.

It is a conceptual error to have a premise that is justified by other nodes, that is, to have a premise that is the consequent of an implication. Premises by definition need no justification.

As a side note, an ATMS node that is justified by the conjunction of nodes that are all premises effectively becomes a premise itself.

Premises have no other known name.

## 6.3 Assumptions 

An assumption is a special kind of node which has data that is conceptually basic to other nodes in the system. Assumptions are significant; they are typically used to justify other nodes. The system uses the assumptions to compose its environments.

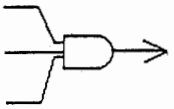
Assumptions are usually specified beforehand, at the time of their creation, using the command `assumption`. However, it is possible to take an existing ATMS node and turn it into an assumption, using the command `assume-this-node`. It is not possible to delete an assumption once it has been made, or to turn it back into an ordinary node. However, assumptions can be permanently retracted by using them to directly imply the NOGOOD node; they can be conditionally retracted, by using them to imply the NOGOOD node in conjunction with other nodes. By ignoring all environments

that contain them, they can also be ignored by the user.

An assumption automatically takes on both the values BELIEVED (assumed) and UNKNOWN (not assumed) at the same time. This in effect creates two possible worlds, one in which the assumption is present, and one in which the assumption does not exist (which is the same as the current world). In fact, if there are currently multiple possible worlds (environments), creating a new assumption has the potential of doubling the number of environments. However, the system only creates new environments that are significant, and some of the new environments may be invalid (inconsistent). Thus, in practice a new assumption usually adds only a few extra environments (possible worlds).

It is usually a weak conceptual error to have an assumption justified by other nodes. Assumptions are basic, and other nodes depend on them, not vice versa.

Assumptions are also known elsewhere as *assumed nodes*. [4]

## 6.4 Implications 

The implication is the second main entity of the system. Implications tie nodes together. An implication has one or more *antecedents* and a single *consequent*, all of which are some type of node. Like nodes, implications can store data; this is usually the name of a reason or rule that this implication represents, although it could be a list of variable bindings, a pointer to a rule in an inference engine, or some other useful data, etc. Also like nodes, the data is purely for the use of the user system, and is not referenced by the ATMS.

Implications are created using the `implication` command. Once the implication is created, it is fixed in place; antecedents and the consequent cannot be added, deleted, or changed, and the implication itself cannot be destroyed. Implications can be retracted by adding in an additional assumption node

[4] (There is a technical difference between an assumption and an assumed node that does not matter to the user. An assumption is implemented as the conjunction of an ATMS node (the assumed node) and a system-only special kind of node (the assumption-tag), along with a justification from the assumption-tag to the assumed node. De Kleer, the original ATMS author, basically uses "assumption" to refer to only the assumption-tag itself in his papers, while using "assumed node" to refer to the atms-node. To make matters conceptually simpler, this implementation detail has been hidden from the user.

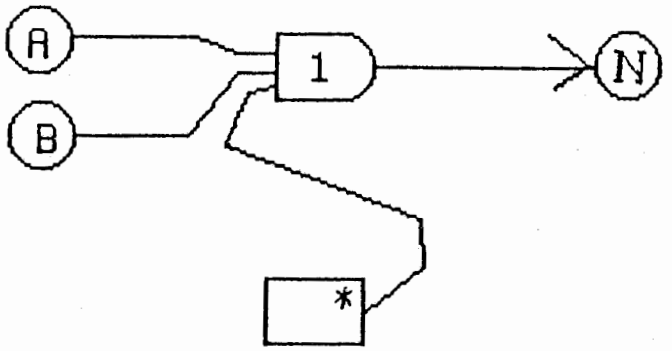



Figure 2: Creating a Retractable Implication.

to the antecedents, on creation, that has the value "This implication is not retracted". If this assumption is BELIEVED, then the implication is active; if the assumption is UNKNOWN, then the implication is effectively retracted. See Figure 2.

An implication acts like a high-impedance "AND" gate. If, in a particular possible world, all the nodes on the antecedent side of the implication are BELIEVED, then the consequent node must be BELIEVED also. However, unlike TRUE/FALSE logic, if any of the antecedent nodes of an implication are UNKNOWN, then the consequent's value is undefined–it is UNKNOWN by default, but it is also possible for it to be BELIEVED if some other implication also has that node for a consequent (or if the consequent happens to be an assumption). Many implications connecting to a node thus act like a "wired OR" in electronics. See Figure 3. If the implied (consequent) node is the NOGOOD node, the implication has a slightly different effect, as the NOGOOD node cannot be believed; see the next subsection.

An implication is a simple construction. It is not possible to have one of the antecedents to an implication be the negation of a node, or a function of a node–implications only take nodes themselves as antecedents and

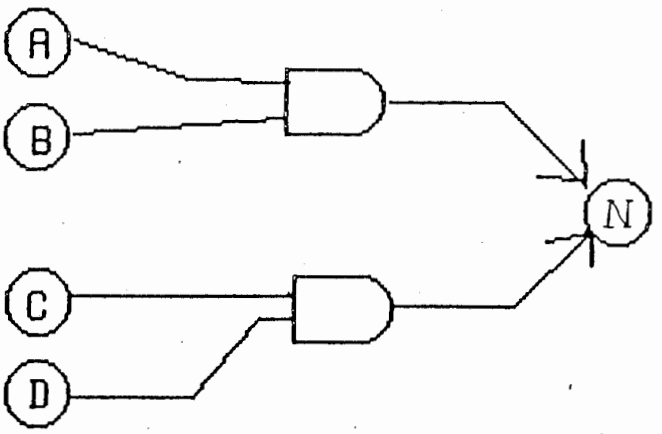



Figure 3: Many Implications Justifying a Single Node.

consequents.[5] A simple implication can only represent a single conjunction; it takes multiple implications to represent a disjunction.

It is a system error to use an implication as another implication's antecedent or consequent. An intermediate node should be introduced instead.

Implications have many conceptual interpretations. One is that belief in the set of the antecedents, when taken together, *implies* belief in the consequent as well. A second interpretation is that the belief in the consequent, if it exists, is *justified* by belief in the set of antecedents. A third interpretation is that belief in the consequent, when taken together with belief in the antecedents, offers a *constraint* on the allowable states of the problem. Indeed, it is usually possible to directly translate constraint-based reasoning problems into an ATMS representation. Implications can also be thought of as representing *productions*, or representing instantiations of *inferences* from an "expert system".

For these reasons, implications are also known as *justifications*, *constraints*, or, occasionally, *inferences*. The "justification" terminology is quite

[5]The way around this is to create another node that is defined by the user system to represent the negation, or the function, of the desired node or expression.

common in the literature, and will be used occasionally here. The implication itself is called a *justification*; one says that the antecedents *justify* the consequent node, and the consequent *is justified by* the antecedents.

### 6.5 The Nogood Node ⊗

The NOGOOD Node is a special node that is always automatically supplied by the system when the system resets. It can be found using the expression `(Node# 0)`, or the variable `*nogood-node*`. It embodies the concept of inconsistency. If any set of one or more nodes implies the NOGOOD node, the conjunction of that set is *inconsistent*–it does not make sense to have all of them believed true at the same time. It is easy to specify this using the `nogood-set` command, which creates such an implication for the user. If the user wants a particular node set to NOGOOD so that it will never be BELIEVED again in any possible world, it is possible to use the `nogood` command. This command will also work on a series of nodes, which sets *each* node to NOGOOD–note that this disjunction is a much stronger condition than the corresponding `nogood-set` command, which still allows the nodes to be believed in other possible worlds that don't contain that particular set. See Figure 4.

The NOGOOD node is sometimes also called *False* or *Falsity*, *Inconsistent*, or *The Contradiction*.

## 7 Logical Operations with an ATMS

It is possible to represent logical operations on nodes using the ATMS's implications. The basic operations are presented here; naturally, since all of the basic operations can be represented, any logical expression composed of these operations can be represented as well. See Figure 5.

### 7.1 The AND operation

The AND operation (conjunction) on one or more nodes is represented by using all of the nodes in the set as antecedents to a single implication. The single consequent is the result. If the antecendents are all BELIEVED, then the

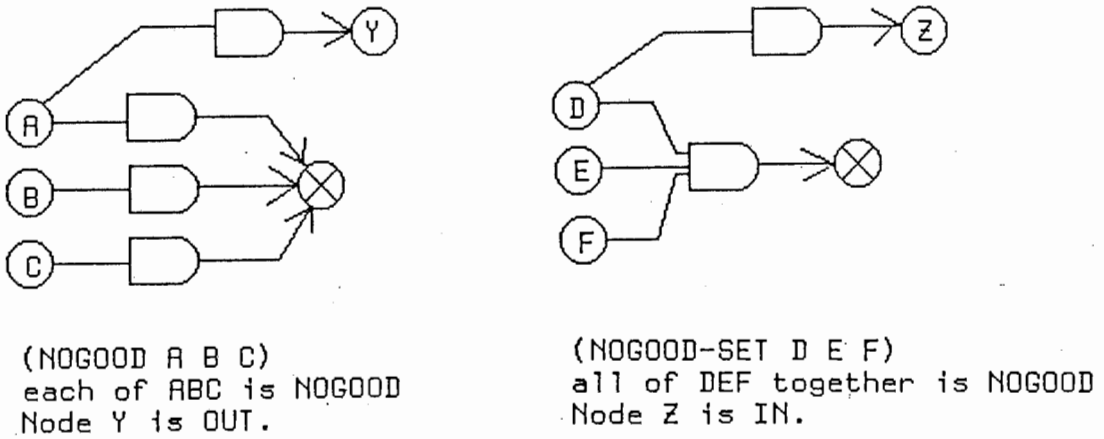



Figure 4: The nogood and nogood-set commands.

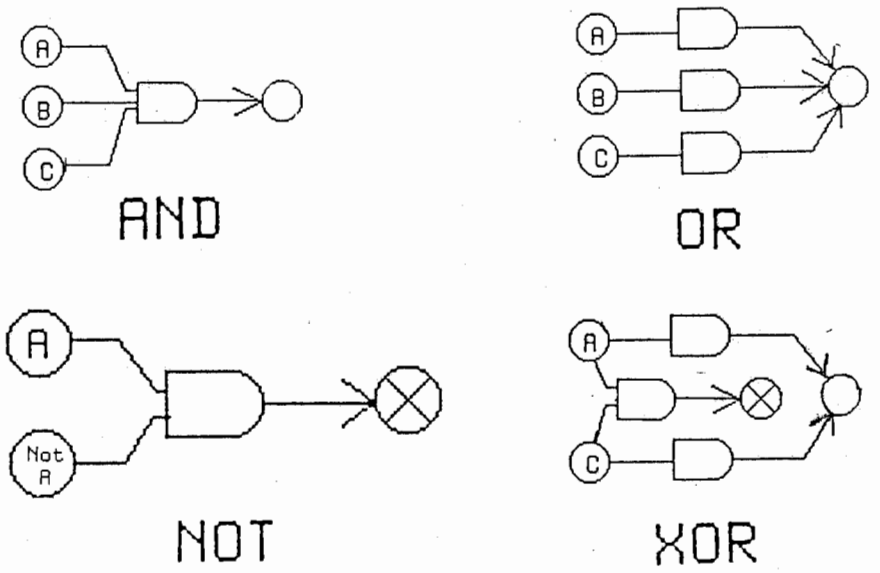



Figure 5: Logical Operations Using an ATMS.

consequent must be BELIEVED as well; otherwise, it is undefined, defaulting to UNKNOWN.

## 7.2 The OR operation

The OR operation (disjunction) on one or more nodes is represented by using each of the nodes in the set an an antecedent to multiple single-antecedent implications, each of which have the same consequent, which is the result of the disjunction. If any of the nodes is BELIEVED, the result must be BELIEVED as well; otherwise, it is undefined, defaulting to UNKNOWN.

## 7.3 The NOT operation

The NOT operation is a little harder to represent in an ATMS, as there is no inherent way to represent the negation of a node. Thus, another node must be explicitly asserted by the user system, with the contents labeled "NOT (the first node)". The occurence of both of these nodes at the same time is inconsistent; this is represented by implying the NOGOOD-NODE with their conjunction. Thus, if the main node is BELIEVED, then the NOT node must be NOT BELIEVED; if the NOT node is BELIEVED, then the main node must be NOT BELIEVED. If either node is UNKNOWN (NOT BELIEVED), then the other node is undefined, defaulting to UNKNOWN.

## 7.4 The XOR operation

The XOR operation (exclusive disjunction) on two nodes is represented by combining the representations for OR and NOT. If either one or the other nodes is BELIEVED, then the result is BELIEVED. However, if both of the nodes are BELIEVED, then this is inconsistent, and will not occur in any valid world.

# 8 More Data Types

Besides ATMS-nodes, premises, assumptions, implications, and the nogood-node, the system also works with environments, labels, and the truth environment.

## 8.1 Environments

Environments represent possible worlds. An environment is a set consisting of all the *assumptions* that are BELIEVED in this possible world. However, since premises and the regular ATMS-nodes that are not assumptions can be derived from the assumptions, they are not mentioned. Environments are as small as possible while still being significant. For this reason, they are known technically as *characterizing environments*. The name of the possible world consisting of the environment plus all of the nodes that can be derived as BELIEVED from that environment, is called the *context*.

The system automatically creates all significant environments, given the set of assumptions. There are various functions (e.g., `(Env# n)`) to reference these environments. The function `(env-assums env)` returns the assumptions of an environment. As a new feature, the user system can create environments itself for its own convenience; note however that these will in general not be minimal, and therefore will not be characterizing environments.

## 8.2 Labels

Whereas environments store a set of ATMS-nodes (assumptions) that are believed in any given possible world, labels store a set of possible worlds (environments) that entail the belief of any given ATMS-node. Each ATMS-node has a label that stores a list of characterizing environments. Each environment in the node's label is consistent, and has that node in its context.

Note that labels are *minimal*. Thus, both the environments and the list of environments are as small as possible. If an environment is a superset of another environment, the larger environment is *subsumed* and not represented in the label.

As a special convenience, all inconsistent environments are stored in the label of the `*nogood-node*`.

## 8.3 The Truth Environment

There is a special environment that has no assumptions at all. If any node is BELIEVED in this environment, it is BELIEVED in *every* environment, because nothing has to be assumed in order for the node to be true. Such

nodes are called premises. The environment is therefore called the Truth Environment; it is initialized by the system on reset, and stored in the variable `*truth-env*`. Note that if a node is BELIEVED in the Truth Environment, it does not matter if it has any other explicit justifications; the Truth Environment subsumes all other environments.

# 9 Truth Values Revisited

The precise definitions of the truth values used in the system can now be discussed. The ATMS operates using a rather unusual truth system. It is important to understand this properly, in order to be able to understand the operation of the ATMS and obtain predictable results.

Most common logic-based systems use a binary logic with the values TRUE and FALSE. Some systems use a trinary logic with the values TRUE, FALSE, and UNKNOWN.

The ATMS, however, is based on assumptions. Therefore, each node takes on one of the binary values BELIEVED (basically, True), or NOT BELIEVED (basically, Unknown, or No Opinion). (It is important to note that Not Believed is quite different from False.)[6] These values hold for any node in any *single* possible world. In addition, these values are only valid for the possible world as it currently stands; it is possible that they may change if the possible world is expanded. Remember that nodes represent ATMS-nodes, assumptions, and premises.

There are other truth values that are derived from these. Since there are multiple possible worlds, it is convenient to define concepts that describe the overall behavior of the node in all of the possible worlds currently known to the system.

The first distinction is the value set IN and OUT. If a node is currently NOT BELIEVED in *all* currently known possible worlds, then it is designated *OUT*. If a node is currently BELIEVED in *at least one* possible world, then it is designated *IN*. A single node can change from IN to OUT and vice versa as the system's set of possible worlds progresses.[7]

[6]The difference is that under a binary logic, ([either] X OR NOT X) is always true. Under the logic used by the ATMS, this can be NOT BELIEVED. This difference has many consequences.

[7]This definition only holds for consistent possible worlds, as inconsistent worlds are

Note that therefore it does not make sense to talk about *the* truth value of a node, whether it is BELIEVED or NOT BELIEVED. Unlike other kinds of truth maintenance systems such as a JTMS (that only represent one possible world at a time), in an ATMS a node can be both BELIEVED AND NOT BELIEVED– because it occurs in many different possible worlds simultaneously. One can only talk about the belief value relative to a particular possible world, or about the global truth values of IN and OUT.

The second distinction is between PREMISED TRUE and NOGOOD. Nodes that are PREMISED TRUE are automatically always BELIEVED in all possible worlds. Nodes that are NOGOOD are always NOT BELIEVED in all possible worlds. Once a node is designated PREMISED TRUE or NOGOOD, it does not change out of this category. A node becomes PREMISED TRUE if it is a premise node, or directly justified by only premise nodes. A node becomes NOGOOD if it directly solely justifies a NOGOOD node.

As was discussed before, there is a special node, called The NoGood Node, that is always included in the system. Like other nodes, this node can theoretically take the value BELIEVED or NOT BELIEVED. However, since The Nogood Node is NOGOOD, it must always be NOT BELIEVED in all (valid) possible worlds (which, of course, implies that it is always OUT).

These truth values apply to nodes. There are also truth values that apply to possible worlds. These are the values CONSISTENT and INCONSISTENT. If a possible world does contain a NOGOOD node with a BELIEVED value, then this world is called INCONSISTENT. If a possible world does not contain a NOGOOD node that is BELIEVED, then the world is CONSISTENT (or VALID). Inconsistency is stronger than implication and thus propagated belief; it does not make much sense to talk about which nodes are believed in an inconsistent world (unless perhaps the user system is performing debugging), because the beliefs are inconsistent.

The classification NOGOOD also applies to worlds, and means the same thing as an INCONSISTENT world. The usage can be determined from the context.

---

considered to be not possible.

# 10 Theory–Types of Knowledge

## 10.1 Theory

This section of the manual briefly digresses into an overview of different theoretical types of knowledge. This is important in understanding the application of the ATMS to actual problems.

When talking about a state, an action or some other kind of concept, there are at least three important attitudes that can be taken towards that concept, or conversely, three ways of knowing that concept. The first is theoretical or *hypothetical* knowledge. This is used to talk about concepts in the abstract, without any commitment as to whether the concepts actually exist or not. An example is, "People who are asking questions" (or, more formally, "Hypothetically, there might exist such a thing as a person who is asking a question"). Another example is, "People who are expecting answers" (or, more formally, "In theory, there might exist such a thing as a person who is expecting an answer.")

Hypothetical concepts can be linked with hypothetical rules. An example of a hypothetical rule is: "People that ask questions expect answers", or, more formally, "In theory, if a question is being asked, then always an answer is expected."

The second kind of knowledge is uncertain, potential, or *possible* knowledge. This is used to talk about a concept that is suspected of existing, but the question of its actual existence is unclear or could be challenged later. An example is, "This person might be asking a question", or, more formally, "It is possible that right now a question is being asked".

Note that possible concepts, when combined with hypothetical rules about hypothetical concepts, produce further possible concepts. Thus, using the previous hypothetical example, the new possible knowledge "It is possible that right now an answer is expected" is now known.

Note that if a concept is possible knowledge, it usually implies the consideration that it is also possible that that knowledge could be not true.

The third attitude that can be taken towards a concept is taking it as *actual* knowledge. This is used to talk about a concept when it is clear that the concept actually exists, and when there is no possibility that that concept could be challenged later. An example is, "This person is asking a question", or, more formally, "It is actually true that right now a person is asking a

question".

Actual concepts can also combine with hypothetical rules to produce further actual concepts. Again, using the previous hypothetical example, the actual concept "An answer is expected" is produced (more formally, "It is actually true that right now an answer is expected"). [8]

## 10.2 The ATMS Representation

The ATMS represents the different kinds of knowledge in different ways. Hypothetical knowledge is represented by the ATMS-nodes. But, unless there is a reason to BELIEVE the knowledge, it remains hypothetical, and is not used by the system. Possible knowledge is represented by the BELIEVED/NOT BELIEVED paradigm. If a node is an assumption, then it represents possible knowledge; ATMS-nodes that are implied by assumptions and therefore also come to be BELIEVED in some contexts also represent possible knowledge. Actual knowledge is represented by premises, which are always believed in all possible worlds.

## 10.3 What's Been Left Out?

One thing that has not been addressed in this section is the question of multiple possible or actual instantiations of a single theoretical concept. The representation of this is more difficult, and is beyond the scope of this discussion.

Another thing that has specifically not been addressed is the question of time. All of the theoretical and possible knowledge that has been discussed in this section is believed in a timeless sense. That is, all of the hypothetical rules are true for all time. This does not allow for the representation of actions, where something can be changed or removed. All of the possible worlds that are generated are different possibilities that may be true "right now", there is no way to represent the changes required to reason about possible progressions in the future. The representation of this is also more difficult, and beyond the scope of this discussion.

[8] In addition, actual concepts can combine with other possible concepts when hypothetical rules have multiple antecedents. However, in this case another possibility is produced, *not* another actuality.

# 11 Working with the ATMS

This section will discuss the theory and practice of using the ATMS. Various aspects of storing data, remembering concepts, and represent the implications of those concepts will be discussed. The method of representing the justification of a concept, along with constraints, conflicts, and inconsistencies, will be examined.

## 11.1 Setting Up the System

The current official version of the system is stored in the Lisp machine file server under `LM01:>myers>golden-atms`. To load the system, the user should type

```
(load "LM01:>myers>golden-atms")
```

The system resets itself upon loading, and will come back with the value `"ATMS cleared out"`.

In order to reset the system, type the command `(reset-atms)`. This erases all the previously entered ATMS-nodes, assumptions, premises, and implications; it resets all the counts to zero, and initializes the system variables `*nogood-node*` and `*truth-env*`.

## 11.2 Setting Up the Problem

The problem must first be examined to determine what the relevant concepts are, and to understand whether they are hypothetical, possible, or actual knowledge. For each individual concept, a node must be created. This should be an ATMS-node for hypothetical concepts, an assumption for possible concepts, or a premise for actual concepts. An implication should be created for each rule, or justification. See section 7 for explanations on how to represent logical operations such as AND, OR, and NOT using the ATMS's implications.

Since in most cases at the start of a problem only hypothetical knowledge exists, and there is no actual knowledge, usually the user system will start out by building a network consisting almost entirely of ATMS-nodes. The few ground truths that the problem has will be represented by premises.

An ATMS only works with constant data. Thus, if there are any variables in the concepts, these variables should be instantiated; a distinct node should be created for each possible instantiation. If it is the case that a variable can take on only one instantiation at a time, the additional pairwise constraint that simultaneous belief in `value-1` and `value-2` is inconsistent should be added, using the function `(nogood-set v-1-node v-2-node)`. This should be called pairwise on all possible pairs of value instantiations for this particular variable.

The actual mechanics of network creation are discussed in the following subsections.

### 11.2.1 Creating an ATMS-node

Suppose we want to create an ATMS-node to represent the data `A`. This is done with the command `(atms-node 'A)`, which returns the created node. Naturally, we will probably want to use this node again, so it is a good idea to store it in a user variable: `(setq node-A (atms-node 'A))`. There are two other ways to reference this node, once it has been created. Since this is the first node, `(atms-node# 1)` will return the node. Also, the next time `(atms-node 'A)` is called, the old node will be returned instead of creating a new node with the same data.[9] Note that this conveniently solves the forward-reference problem when creating large networks–the user does not have to determine whether a node has been previously created or not.

Simply creating a node informs the system that the given concept exists, but does not say anything about whether it is believed or not. Thus, without any further justification, the node must be assumed by default to be NOT BELIEVED. Indeed, since it is NOT BELIEVED in all possible worlds, it is OUT. We can check this with `(explain-node node-A)` or `(OUT-p (atms-node 'A))`.

[9]This is assuming the `use-uniquification` system flag is set to `T`, which is the current default. "The same data" is tested by using the `equal` function; thus strings work properly. This flag must be set to `nil` if multiple copies of the same data are required to have distinct identities. In this case, the user is responsible for keeping track of the different nodes.

### 11.2.2 Creating a Premise

There are two ways to create a premise. Either we can create a new premise node directly, e.g. with `(setq node-B (premise 'B))`, or we can turn an old node that is not already a premise or an assumption into a premise, using the function `(premise-this-node node-A)`.

Since premises are by definition BELIEVED in *all* possible worlds, they are always IN, and they do not contribute towards any information distinguishing the possible worlds. Thus, they are not included in the environments. The INness can be checked using `(explain-node (atms-node 'B))` or `(IN-p (atms-node 'B))`.

### 11.2.3 Creating an Assumption

There are also two ways to create an assumption–directly, using `(setq node-C (assumption 'C))`, or by using an old node that is not already an assumption or a premise, `(setq node-D (atms-node 'D))` and then `(assume-this-node node-D)`.

An assumption splits the universe into two sets of possible worlds–those in which the assumption is BELIEVED, and those in which the assumption is NOT BELIEVED. Possible worlds are represented by environments, which can be listed using `(print-envs)`.

### 11.2.4 Creating an Implication

Implications have one consequent, some user data, and at least one antecedent. The consequent and the antecedents must be nodes, they cannot be other implications. It is convenient to create ATMS-nodes and assumptions at the same time that an implication is being created. Thus, if it is desired to imply ATMS-node C from the conjunction of premise A and assumption B, it is convenient to use the command:

```
(implication (atms-node 'C) "Because A and B -> C"
                            (premise 'A)(assumption 'B))
```

This creates the desired implication. It also does the right thing with respect to the nodes: if nodes A, B, or C don't exist, they are created; if they do

exist, the system does not create another copy, but references the previously created version.[10]

In general, there is no reason to save pointers to implications, as there are basically no significant operations that can be performed on them. Implications, once created, cannot be modified or deleted.

### 11.2.5 Referencing Data

In general, as was discussed, there are at least three different ways of referencing data. The easiest is for the user system to store pointers to the nodes and implications themselves. Since each creation function returns the object created, that object can be stored inside a convenient user variable at the time of creation. For example, `(setq my-assumption-A (assumption "Data for Assumption A"))`. The second method is to reference the object using its ID number. Each object has an ID number that distinguishes it; these may be found by the ID accessor functions (section 3.7.2–e.g. `(premise-ID` *node*`)`) or by printing the object (section 3.6). The number accessor functions (section 3.7.1) (e.g., `(Assum#` *n*`)`), given the ID number, will return the object. This is perhaps more convenient for the interactive user. The third, and easiest method, uses the user system data as an access key. As long as all data is unique, simply calling `(atms-node` *data*`)` will return the previously cerated node (or create a new one, if it wasn't there.) Note that this works for premises and assumptions as well.

Finally, once an ATMS object itself is located, the user system data that is stored inside the object can be returned by the data accessor functions (section 3.7.3).

## 11.3 Exploring the Problem

The ATMS is designed to be a system that is used by the user system *interactively*. As the user system learns more about the problem, new nodes and justifications should be entered. If a concept is recognized as being no longer hypothetical but possible, the node representing that concept should be assumed, using `(assume-this-node` *node*`)`. If a concept is recognized as being no longer hypothetical but actual, the node representing that concept

[10] Again, this behaviour is dependent upon the system variable `use-uniquification` not having been changed from its default value of `T`.

should be premised, using (`premise-this-node` *node*). If a possible concept is recognized as actual, it can be justified with a premise.

If reasons are recognized as to why a hypothetical node could be true, that node should be justified with (possibly new) nodes representing those reasons. If new relationships between concepts or new implications are realized, appropriate implications should be created.

### 11.3.1 Constraints

If a constraint on the problem is recognized, that constraint must be entered into the system using a nogood command. If it is realized that a concept (say, `node-A`) will never be believed, no matter what, the system can be notified of this fact by using the command (`nogood node-A`). This is known as retracting the node. Note that this is a permanent action. It is an error to call `nogood` on a premise; this leads to *all* premises being NOT BELIEVED.

The second kind of nogood command deals with sets. If it is realized that all of the nodes in a set can not *together* be true at the same time, then the command (`nogood-set node-A node-B ...`) should be called. Note that this does not say that any *particular* node is nogood, only that the *simultaneous belief* of all the nodes is nogood. `Nogood-set` thus makes the AND of a set of nodes inconsistent; `nogood` called on each member of a set of nodes makes the OR of the set inconsistent and is therefore a much stronger condition.

It is possible to nonmonotonically retract a single node by creating a new assumption (containing the data, "This other node is really nogood" in a manner similar to conditional retraction of implications (Figure 2)) and then calling `nogood-set` with both the node and the assumption. When the assumption is believed, the node is retracted; when the assumption is not believed, the node is not retracted. Note that the capacity to revoke the retraction must be specified at the time of retraction; if the user system simply calls `nogood` on the node itself, the system cannot later change its mind.

## 11.4 Interpreting the Results

Once the problem has been set up, the user system needs to obtain the required results from the ATMS, and be able to interpret them.

There are two basic operations that can be performed on the system once the problem has been set up. The first one is asking what concepts are (or whether a particular concept is) BELIEVED in a particular possible world. The second operation is the dual to this: asking in what possible worlds a particular concept is believed.

Several functions, including the explanation facilities, are built up around these operations. How the user actually uses these functions depends upon details of the problem and the approach.

### 11.4.1 System Environments vs. User Environments

There are two main approaches that can be taken towards data interpretation. One is to let the ATMS system handle the environments (possible worlds). The system only creates and reports environments that are significant, i.e. necessary and minimal. The user system can then work with these environments, in a data-driven fashion. The system performs this function in any case; thus this approach is faster than the second approach. However, these environments can be smaller that what the user system has in mind.

The other approach is for the user to handle the environments. The user system specifies the environments it is interested in, and has the system follow the course of those environments. The user system has more control in this case, but in general the particular environments specified by the user will probably not be used by the system–only various subsets of them. The results thus require more interpretation.

In the following section, general functions that are used under either approach are discussed. Then, the system environment approach is described. Finally, functions supporting the user environment approach are discussed.

### 11.4.2 General Functions

Remember that a "possible world" can be thought of either as the *environment* (a set of BELIEVED assumptions that specify the set of concepts or nodes that are believed), or the *context* (the set of *all* nodes that are believed, given a particular environment). Each context has a *characterizing environment* that represents it. Although the ATMS system itself only deals with characterizing environments, in general when the user system specifies an environment to the ATMS this will not be a characterizing environment.

Whether the user system depends on the ATMS system or specifies its own environments, it is important to realize when an environment has become nogood (inconsistent). This is tested for by the function `(nogood-p env)`, or equivalently, `(inconsistent-p env)`. Having an environment become nogood is a monotonic process; once it is inconsistent, it stays inconsistent. The inconsistency of an environment is a global property, unlike the NOT BELIEVED value of a node, which is local relative to a particular environment (e.g., see `nogood-set`).

A list of the assumptions that make up an environment can always be returned using the command `(env-assums env)`. A list of all of the concepts that are BELIEVED under a particular environment, including believed ATMS-nodes, assumptions, and premises, can be returned by using the command `(context env)`; however, this is an expensive function. If a simple test for a particular node is desired, it is much faster to use the function `(in-context-p node env)`, or equivalently, `(in-world-p node env)`. If it is known that the node is an assumption, it is even faster to use the function `(in-env-p assum env)`; this is the function that should be used most of the time.

Occasionally it may be the case that the user knows that an entire environment (and all of its supersets thereafter) is nogood, even though the system may not believe this yet. To inform the system of this decision, the command `(nogood-env env)` should be used. This essentially calls `nogood-set` on (the conjunction of) all of the assumptions in the environment, so that that particular combination of assumptions can never be valid again. In general, however, this should be used only because of some higher-level knowledge or decision that is not part of the knowledge represented in the ATMS; otherwise, the correct method is to `nogood` the particular node, nodes, or set of nodes in the environment that were responsible for this decision.

### 11.4.3 System Environments

Under the system environment method, the user system makes use of the result environments returned by the system. These will be as small as possible. The user system must then interpret the results.

The function `(env-assums env)` was already discussed to determine the assumptions that an unfamiliar environment posesses. In order to get a list of the (consistent) minimal environments in which a particular node is believed,

use `(node-label node)`, or equivalently, `(node-envs node)` or `(why-envs node)`.[11] If a list of *all* of the known (consistent) environments under which a given node is believed is desired, the function `(all-node-envs node)` should be called; however, this function is slightly expensive, as the ATMS is designed to work with minimal environments.

Suppose that two particular environments are interesting (say, from two different nodes). It is possible to create the concatenation of these two environments by using the function `(OR-env env1 env2)`. This returns an environment composed of the union of the two environment's assumption sets. The user system can then keep track of this single environment; the system will mark it as NOGOOD if it ever becomes inconsistent.

One of the problems with using the environments given by the system is that they are always minimal. In addition, the system tends to create small pieces of power sets of the assumptions. Although the environments from a single node are guaranteed to be minimal, that is, they are as small as possible and any one environment is not a subset of any other environment, when the environments from several nodes are combined, some of the environments in the set will be non-interesting, "insignificant" subsets of other environments. To eliminate these insignificant environments, the function `(significant-envs env-list)` (or, `(sig-envs env-list)`) should be called. This returns a list of the remaining environments, after the insignificant environments have been dropped from the given list. The environments are checked for validity at the same time, so only the consistent environments are returned. If this function is called with no argument, it defaults to examining `*environments*`, a list of all the (valid and invalid) environments known to the system. Thus, `(sig-envs)` returns the largest significant environments known to the system. For the interactive user, these two functions also have a printing output version, `(print-significant-envs env-list)` or `(print-sig-envs env-list)`, which again defaults to the entire system's known environments without an argument.

There are two ways to retract a concept. One way is done inside the ATMS by setting the node representing the concept to NOGOOD, as has been previously discussed. The second method is for the user system to disregard all environments containing that concept. The second method costs more in terms of ATMS computation, but it has the advantage that concepts can

[11] This is actually the same as `IN-p`.

be temporarily retracted or can be easily un-retracted without specification ahead of time. The best function to use to follow this method is `(dont-use assum-list env-list)`, which returns a list of environments consisting of the given environment list, from which all environments containing any of the assumptions in the given assumption list have been deleted. Disregarding all environments in which a particular assumption is used has the result of effectively setting that concept to NOT BELIEVED in all used possible worlds. There is an analogous function that takes any kind of ATMS node as its input, instead of just assumptions, `(dont-use-nodes nodes envs)`; this function disregards an environment if one of the given nodes is in that environment's context. Thus, this is a more general function, but it is slightly computationally expensive.

### 11.4.4 User Environments

Under the user environment method, the user system creates environments consisting of sets of assumptions that the user system is specifically interested in. The system then follows this new environment as well, noticing what nodes are it its context and when it becomes nogood.

This is a relatively easy approach; it is also computationally inexpensive. However, there is a danger that, for any particular concept or small set of concepts, the set of relevant assumptions picked by the user could be larger than necessary, and therefore could even include some irrelevant assumptions that unnecessarily force the node to be NOT BELIEVED in that environment. (The previously discussed `(node-envs node)` function returns the system's record of the minimal environments for that node, in which that node is believed, in case the system's opinion is desired.)

The advantage of this method is that the user system can specify exactly which environments it thinks are significant; there is then no trouble interpreting the results when the system returns a smaller subset environment using the previous approach. Specifying a new environment is done using the command `(create-env assum-list)`, which creates and returns a new environment if necessary, or returns the old one if the system had created it already. If an ATMS-node in the assumption list was not previously an assumption, it is assumed by this function; however, this side-effect should be used with care. If the resulting environment is NOGOOD, this function currently returns `nil` instead of an environment.

If the user system simply wants to find out whether a particular environment exists or not, but does not want to create it if it does not exist, the function `(find-env assum-list)` can be used. This function returns `nil` if the environment is not already there. Find-env is a fast function.

If there is an existing interesting environment and the user system wants to enlarge it with one or more extra assumptions, the function `(add-assums-to-env old-env assumptions ...)` will create (if necessary) and return the new environment (or, currently, `nil` if the environment is NOGOOD). The old environment is untouched.

As was previously discussed, if there are two interesting environments, and the user system wants to combine them to form an environment consisting of their union, the function `(OR-env e1 e2)` is used.

The function `(subsumed-by-p larger-env smaller-env)` can be used to test whether a user system environment is subsumed by (is a superset of) a particular system environment. In this case, if the smaller system environment becomes NOGOOD, the larger user environment will, too. This function is extremely fast.

When the user system specifies a particular environment, it is useful to know whether any of the assumptions in the environment are redundant (implied by some of the other assumptions) or not. The function `(characterizing-env env)` will return an environment having a set of assumptions as small as possible. This function returns the given environment if it is a characterizing env, or `nil` if the given environment is inconsistent.

## 11.5 Examples: Demonstration of Capabilities

### 11.5.1 Truth Maintenance: Consistency Management

One of the main tasks of any truth maintenance system is to manage the consistency of the data-base it stores. If a node or set of nodes becomes inconsistent, the implications of this fact should automatically propagate throughout the network.

To demonstrate this capability, a test network, shown in Figure 6, is created using the following commands:

```
(implication (atms-node 'D) '1 (assumption 'A))
(implication (atms-node 'G) '2 (atms-node 'D))
```

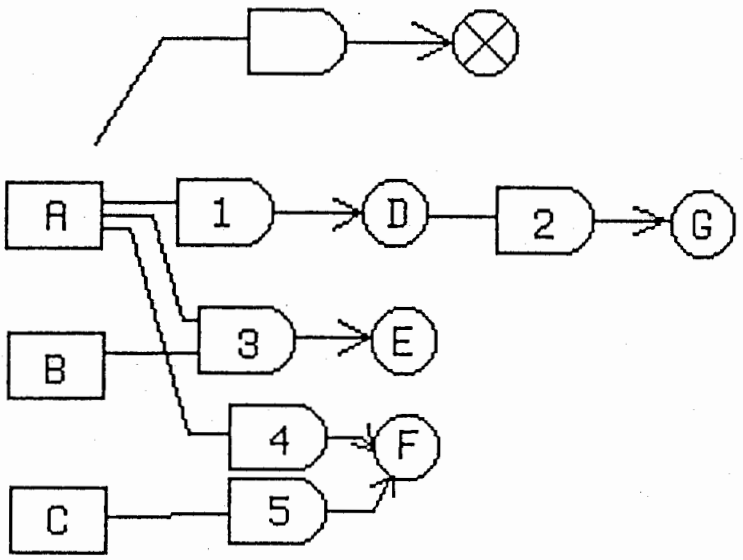



Figure 6: Network for Truth Maintenance Example.

```
(implication (atms-node 'E) '3 (assumption 'A) (assumption 'B))
(implication (atms-node 'F) '4 (assumption 'A))
(implication (atms-node 'F) '5 (assumption 'C))
```

As can be seen, node G is based on node D, which in turn is based on the assumption that A is believed true. Node A is also used to justify node E with an AND connection, that is, node A **and** node B imply node E. In addition, node A is used to justify node F with an OR connection, i.e. node A **or** node C implies node F. The basic assumptions in this case are nodes A, B, and C.

In the network as it stands, all the nodes are IN for one reason or another. Both nodes D and G have the sole environment {A}; node E has the environment {A,B}, and node F has two environments in its label, {A} and {C}. Remember that each environment corresponds to a possible world in which this node is believed. Of course, all the assumptions have one environment apiece, i.e. {A}, {B}, and {C}, respectively. This information can be summarized by using the command `(explain-nodes)`.

```
NOGOOD-NODE is out.
A is in, under   #<Env 1>: {A}.
```

```
B is in, under   #<Env 2>: {B}.
C is in, under   #<Env 4>: {C}.
D is in, under   #<Env 1>: {A}.
E is in, under   #<Env 3>: {A, B}.
F is in, under   #<Env 1>: {A}  #<Env 4>: {C}.
G is in, under   #<Env 1>: {A}.
```

We now decide to permanently disregard/retract node A by making it NOGOOD, using the `(nogood (atms-node 'A))` command. The `nogood` command sets up a direct implication to the `*nogood-node*`. Thus, node A will be NOT BELIEVED in all possible worlds.

By this command alone, the results of making node A inconsistent are automatically propagated throughout the network. Node D and therefore node G both become OUT, because they had no other justification. Node E also becomes OUT, because it depends on both A and B being believed. However, node F is still IN, because it is still believed in at least one environment. It now has a label with only one environment, {C}. Naturally, assumptions B and C are unaffected by A becoming NOGOOD, and are still IN. Again, using `(explain-nodes)` gives a summary of this information:

```
NOGOOD-NODE is in, under XX#<Env 1>: {A}.
A is out.
B is in, under   #<Env 2>: {B}.
C is in, under   #<Env 4>: {C}.
D is out.
E is out.
F is in, under   #<Env 4>: {C}.
G is out.
```

Note that a computer program can get the same information for a particular node (rather than simply a printed explanation) by calling `(why-envs (atms-node 'A))`.

### 11.5.2 Multiple Contexts: Parallel Contradictory Representations

An ATMS is capable of representing multiple worlds at the same time. In addition, these multiple worlds can be *mutually contradictory*–things can be BELIEVED in one world that contradict things that are BELIEVED in another

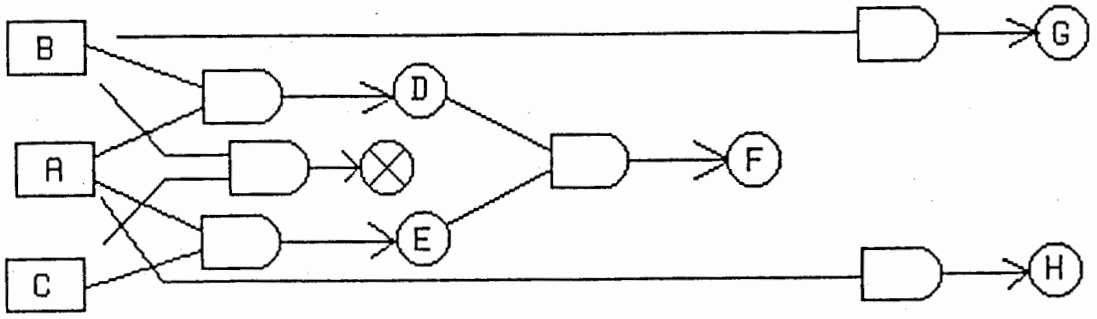



Figure 7: Network for Parallel Representations Example.

world. Both halves of a contradiction can be explored at the same time; there is no need to switch back and forth between contexts, as the system maintains what is true in any one possible world. The ATMS sets up the worlds for the user, and derives them automatically, based on the assumptions and the nogood contradictions that are asserted.

This example starts out with a consistent single world, in which nodes A and B together imply node D, and nodes A and C together imply node E. Nodes D and E imply node F. The world can be found by the function `(print-significant-envs)`, which prints out `#<Env 6>: {A, B, C}`. In addition, this is returned by the function `(significant-envs)` or equivalently `(sig-envs)`. This environment implies all of the nodes (i.e., A through F).

Next, it is learned that C is actually the same as NOT B. Thus, to have B and C be in the same world together is inconsistent. To represent this inconsistency in a permanent fashion, the function `(nogood-set (atms-node 'B)(atms-node 'C))` is called. (This is implemented by implying the `*nogood-node*` with the conjunction of B and C.)

This has the effect of splitting the previous single world into two (mutually inconsistent) parallel worlds. In one world, A and B are believed, with the

result that D is believed. However, C is not believed, and thus both E and F are not believed either. The second world is a mirror of the first—in this possibility, A and C are believed, and thus E as well, while B, D, and F are not believed. Notice that node F is no longer believed in either world. These possibilities are again returned as a list by `(sig-envs)`, and printed out by `(print-sig-envs)`:

```
#<Env 3>: {A,B}
#<Env 5>: {A,C}
```

It is worthwhile to note that under most logics, when both B and not B are represented, the logic collapses and any assumption can be proved or disproved. However, under the BELIEVED/UNKNOWN(NOT BELIEVED) logic used by the ATMS, when the knowledge base represents something that is inconsistent, anything that depends exclusively on that inconsistency (e.g., node F) simply becomes NOT BELIEVED. All other derivations (e.g., D from B, or E from C) are left intact–they do not collapse simply because another part of the knowledge base is inconsistent. Finally, an inconsistency splits the knowledge base into two [sets of] possible worlds–one in which only one half of the inconsistency is believed, and one in which only the other half of the inconsistency is believed.

One of the powerful features of an ATMS is that now both possible worlds can be explored and expanded simply by performing actions on the one ATMS data-base; there is no need to switch back and forth between parallel representations in multiple worlds. For instance, say that it is now learned that B implies G. If this justification is entered, G becomes believed in the {A, B} world, but it is *not* believed in the {A, C} world. It was only necessary to enter the one implication into the data-base, and the appropriate possible world was expanded. Now, say that it is learned that A implies H. When this implication gets added, *both* possible worlds are updated. Node H is BELIEVED in both worlds {A,B} and {A, C}. Note that under normal parallel representations of multiple worlds not using an ATMS, node H would have had to have been added *twice*, once for world {A,B}, and once for {A, C}. The ATMS allows exploration and appropriate update of multiple contradictory worlds at the same time, using single operations.

### 11.5.3 Explanation: Justifying Results

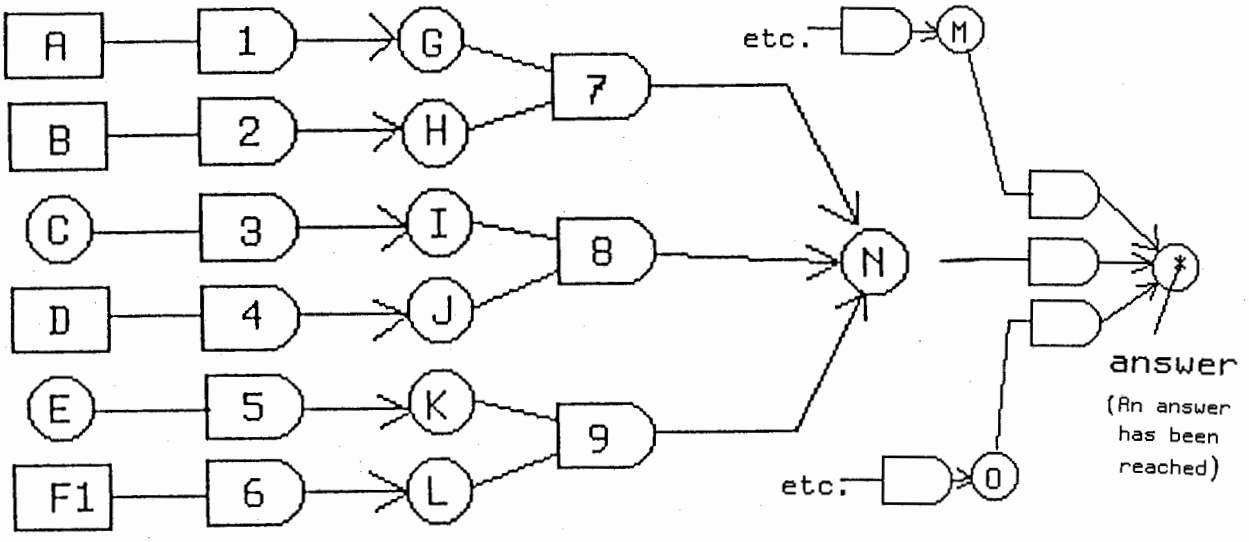



Figure 8: Network for Explanation Example.

In this example, we have a complex network with some basic justifying concepts on the left (nodes A through F), and some possible derivations on the right (nodes M through O). We are willing to assume that nodes A, B, D, and F could be believed. Through monitoring, we have discovered that N is an answer.[12] Now, we want the ATMS system to explain why node N is believed.

The first step is to find out in what possible worlds (environments) node N is believed. This is done using the function `(why-envs (atms-node 'N))`, which returns all the environments in which N is BELIEVED: `#<Env 5>: {A, B}`. (Note that this is actually the same function as `IN-p`). Ideally, there will be only one solution, but it is possible that there can be more than one–the ATMS is powerful enough to represent this. These will be minimal (as small as possible) and non-redundant. Solutions must be explored one

[12]Actually, the monitoring process is just one more level of explanation, reached by calling `(why-envs (atms-node 'ANSWER))` and then `(why-nodes (atms-node 'ANSWER) env)` to get node N, as shown in the illustration. Also, if an interrupt-driven instead of a polled system is desired, it is possible to install a user processing routine into the ANSWER node that gets triggered when the node becomes IN, e.g. to report the answer. This is done with the command `(install-action 'ANSWER 'user-routine)`.

at a time when using the following functions, as it only makes sense to talk about whether a node is believed or not in a particular environment.

The environment itself returns a list of exactly all the assumptions that are used to *originally* justify that node–that is, if the used justifications are recursively followed all the way back, what are the assumptions at the beginning of the justification network. However, say that we want to get an explanation of the *immediate* justifications of the node.

There are two types of answers for immediate back-chaining explanations. The first is concerned with the justification (or implication) itself. For instance, in our example there are three justifications leading to node N: numbers 7, 8, and 9. To obtain which justification is the significant one under environment #5, the command `(why-implics (atms-node 'N) (env# 5))` is used, which returns the list `(#<Implic 11>)`; next, `(implication-data (implic# 11))` will then return the information desired (7).

The second type of answer is concerned with the *nodes* immediately justifying the node in question. In this example, there are six nodes potentially justifying node N (i.e., nodes G through L). However, in the solution, under environment #5 ({A,B}), only nodes G and H actually justify node N and cause it to be BELIEVED. To obtain this immediate explanation, use the command `(why-nodes (atms-node 'N) (env# 5))`, which returns the answer `(#<Atms-Node 3> #<Atms-Node 1>)`. Again, it is necessary to call a data access function, in this case `(atms-node-data (atms-node# 1))`, to get the desired information `G`.

Of course, if the network were deeper than two levels, these functions could be used recursively in a backwards-chaining mode on the previously given answers, to offer further immediate explanations of why *those* nodes were believed.

# 12 Conclusion

This manual has presented the theory and practice of the use of the ATR Assumption-based Truth Maintenance System written in Lisp. Using the functions and examples described here, the user can set up a problem, expand and explore the problem interactively, and interpret the results to the problem as represented by the ATMS. The ATMS can propagate implications to maintain truth values; it can explore multiple possible contexts simulta-

neously, even if they are contradictory; and, it can explain and justify its results. The resulting system can be used as a tool to explore constraint, search, logic, and multiple-world problems.

# A Implementation

As a brief review, from the user's viewpoint, there are three kinds of nodes: ATMS-nodes, premises, and assumptions. There is also one kind of connection between nodes, the implication (or, "justification"). Finally, there are the environments, which consist of sets of assumptions. Use of the system consists of creating nodes, and then creating implications to link them together. The user can also indicate inconsistent nodes or sets of nodes. Environments can then be referenced, to see what assumptions are required in a particular possible world, and which possible worlds are inconsistent.

## A.1 Implementation Data Structures

The actual data-structures that are used to accomplish this are somewhat different from the user conceptualization. There are four types of structures in the implementation: the ATMS-node, the assumption-tag, the implication, and the environment. Premises are implemented as a special case of the ATMS-node. An assumption is implemented as an ATMS-node together with an assumption-tag, with a single-antecedent implication pointing to the ATMS-node from the assumption-tag. All of these objects are implemented as structures for speed.

### A.1.1 ATMS-node structure

An ATMS-node has the fields *data*, *implies*, *implied-by*, *label*, *my-assum*, *ID*, and *rule*. In addition, it has an associated print-function. The data field stores the user's data, and is not referenced by the ATMS system. The implies field contains a list of implications that have this node as an antecedent, i.e. the node implies something. The implied-by field contains a list of implications that have this node as a consequent, i.e. this node is implied-by those justifications. The label field contains a sorted list of consistent characterizing environments which this node is in the context of, i.e. directly or indirectly implied by. If this node is a premise, the label consists of a single environment, the null environment `*truth-env*`. The environments in a label are sorted by size; the size of an environment is the number of assumptions it comprises. The my-assum field contains the assumption-tag for this node if the node is an assumption, or nil otherwise. The ID field

contains a unique non-negative integer identifying this node. And, the rule field is usually nil, but can contain a short program that gets executed when the node becomes IN.

### A.1.2 The Assumption-Tag Structure

An assumption-tag has the fields *my-node*, *environments*, and *ID*. In addition, it has an associated print-function. The assumption-tag's my-node field contains the corresponding ATMS-node that gets assumed, that this tag justifies. Assumption-tags can only justify one ATMS-node. The environments field contains a list of all the explicitly-identified environments this assumption is in. And, the ID field contains a non-negative integer to identify this assumption-tag, that is unique among the assumption-tags.

### A.1.3 The Implication Structure

An implication has the fields *data*, *antecedents*, *consequent*, and *ID*. In addition, it has an associated print-function. The data field contains the user data for this implication, which is not used by the ATMS. The antecedents field contains a list of an assumption-tag, or a list of one or more nodes, that are antecedents to the implication. The consequent field contains an ATMS-node that is the consequent of the implication. The ID field contains a non-negative integer to identify the implication, that is unique among the implications.

### A.1.4 The Environment Structure

An environment has the fields *nodes*, *nogood-p*, *size*, *ID*, and *assum-bits*. In addition, it has an associated print-function. The nodes field contains the context of the environment, i.e. a list of all of the nodes that have this environment in their label. The nogood-p field is nil unless the environment is nogood; this provides a quick check, although it is not strictly necessary. The size field provides a count as to the number of assumptions in this environment; it is used to order the environment in lists. The ID field contains a non-negative integer to identify the environment, that is unique among the environments. And, the assum-bits field contains a special bit-array that has a bit set for the number of each assumption that composes the environment.

## A.2 Important Variables

Besides the data structures, there are a few special important variables that are used by the system to represent the ATMS.

### A.2.1 *nogood-node*

Upon reset, the system creates a single ATMS-node and stores it in the variable `*nogood-node*`. This node then is used in a special manner by the system; it is the consequent of implications that represent contradictions, and it is tested for to determine when to handle nogood environments.

### A.2.2 *truth-env*

Upon reset, the system creates a single special environment and stores it in the variable `*truth-env*`. This environment is the empty environment; it uniquely has no assumptions. Thus, it subsumes all other environments, and is therefore a subset of all environments. Since premises have (`*truth-env*`) as their label, all premises are automatically included in all environments.

### A.2.3 *reprocess-implication-queue*

This variable contains a queue of all implications that will be reprocessed. As reprocessing propagates, implications are pushed onto this queue. Reprocessing continues until the queue is empty. Using a queue allows iteration instead of recursion.

### A.2.4 *environments*

Contains a list of all environments in the reverse order created. Includes all environments, even if they are inconsistent.

## A.3 Creation and Propagation

### A.3.1 Creation of ATMS-nodes, premises, and assumptions

Creating an ATMS-node per se has no extra effect, as it is not attached to anything when it is created. It has the default truth value of OUT.

Creating a premise involves creating an ATMS-node and then resetting its label to the empty environment, environment #0 (which, of course, is always true). Premises are thus always automatically IN.

Creating an assumption involves creating an ATMS-node, creating a separate assumption-tag, and justifying the ATMS-node with the assumption node. Assumption-tags are a different kind of structure, containing less information; they are designed to augment the corresponding ATMS-node's information, not duplicate it; thus, normally one thinks of the assumption-tag and its corresponding ATMS-node as a unit. Creating a new assumption has no significant extra effect.

### A.3.2 Assuming or premising an existing node

However, performing an assume-this-node on an existing node that is the antecedent of an implication requires a possible propagation of the assumption, which is done in the same way as installing a new implication. (discussed in the next paragraph). A node is assumed by justifying it with its assumption-tag. This inherently installs an implication (from the assumption-tag to the assumed ATMS-node), which then causes the assumed node to propagate forward, reprocessing all the implications that that node is an antecedent of. Note that there is no need to propagate backwards or to recompute the label of the assumed node from its other, previous implications, because assumption-tags are unique, atomic, and assigned to justify only the one assumed node.

Premising an existing node involves resetting the node's label to the empty, Truth environment, and then propagating that node's change by reprocessing all the implications that use that node as an antecedent.

### A.3.3 Creating and propagating implications

Creating an implication is more complex. When a new implication is installed (or an old ATMS-node is assumed), it is necessary to propagate the new contribution to the truth maintenance network, by reprocessing the implication. There are three cases: either the node implied by the implication (the consequent) is a premise, the no-good node, or a regular node. If it is a premise, propagation stops, as a premise's label is by definition automatically consistent, sound, complete, and minimal. In particular, premises

cannot be nogood or directly imply the nogood node without another node as an antecedent. Given that the node is not a premise, the other two cases are discussed separately below.

### A.3.4 Processing an implication with the nogood node as a consequent

When an implication implies the nogood node, it means that all the characterizing environments in what *would* be the label of the implied node (the nogood node) are inconsistent. Thus, the first thing to do is to calculate the label of the implied node, by forming the cross-product of the environments (labels) of the antecedents, as discussed below. Next, all of these environments are added into a special nogood bin. This bin keeps the nogoods sorted by size, which makes subsumption testing faster. Naturally, the nogood bin is kept minimal, which means that the new nogood environments as they are added are tested for subsuming previous entries in the bin (in which case the previous entry is deleted). There is no need to test for the new nogood itself being subsumed by a previous entry, as it has already been guaranteed to be minimal. In addition to being entered in storage, the new nogoods are tested against all the previous environments, to see whether any of the previous environments are now newly subsumed and are nogood. These environments, however, do not need to be tested for further subsumption, as subsumption is transitive. Finally, all of the newly inconsistent environments are removed from all nodes' labels that contained those environments. These modified nodes will then propagate their changes forward. In addition, the environments are added into the nogood node's label, for reference.

### A.3.5 Processing an implication with a regular node as a consequent

When an implication to be reprocessed implies a node that is not the nogood, the cross-product of its antecedents' labels is calculated, and then ORed into the current label of the consequent. If this action changes the consequent's label, the node's change must be propagated forward.

### A.3.6 Cross-product

Calculating a cross-product for an implication involves the permutations of choosing one environment from each ATMS-node antecedent's label, and combining the resulting assumptions together into a new environment (using OR on the bit vectors). These new environments then form a new label (which will be combined into the consequent node's current label). As these new environments are being created, they are tested for minimality and consistency, using the subsumption test on environments and nogood environments. If an environment is subsumed by a regular environment, then it is redundant (non-minimal) and does not need to be considered further. If an environment is subsumed by a nogood environment, then it is nogood (inconsistent) and again does not need to be considered further.

### A.3.7 Subsumption

Subsumption is conceptually performed by testing to see whether one environment is a subset of another environment. The subset environment then subsumes the superset environment. Subsumption is actually performed by comparing the environments' assum-bit vectors, for speed. If the OR of the assum-bit vectors is equal to the assum-bit vector of the larger environment, then that larger environment is subsumed by the other.

Comparing subsumption between an environment and a label is made faster by having the contents of the labels sorted as to the number of assumptions in each environment. That way, when testing for being subsumed, only the environments in the label smaller or equal to the testing environment's size need to be processed; when testing for subsuming, only the environments of equal or larger size need to be tested.

Subsumption tests occur when new environments are created, and when new nogood sets are noticed or explicitly implied. In this case, if the new nogood is subsumed by previous, smaller nogoods, it is redundant and not noticed. Otherwise, it is a new, valid nogood, and must be tested against environments and nogoods larger than itself, to see whether it subsumes them.

### A.3.8 Propagating a node's changes

Propagating the changes to a node (forward) in the current implication consists simply of (re)processing all of the implications that use that node as an antecedent, in the manner explained above. Reprocessing the implication handles the required details.

Propagation is currently done depth-first, through iteration at the propagating-node level and FIFO-queue recursion at the reprocess-implication level.

## A.4 Forming Answers to Queries

There are a few basic queries that the user can ask the system. The system answers according to the following algorithms.

**Is an environment consistent?** An environment is consistent if it is not a superset of one of the environments that are known to be nogood. This is tested by using the extremely fast subsumption operation.

**Is a node IN?** In other words, is there *some* consistent environment that supports the node? Yes, iff the node's label is non-null.

**Is a node N in the context of an environment E?** Yes, if E is a superset of at least one environment in N's label. (Again, this is tested using the subsumption operation.) No, if otherwise. However, if the environment is inconsistent, the question is probably meaningless; in any case, since only consistent environments are maintained in labels (except for the `*nogood-node*`) the query will not find a match.

## A.5 Efficiency Considerations

Data structures are implemented with Lisp structures, instead of flavor objects. This results in faster access time. Some previous ATMSs have based their propagation on nodes, requiring a node to recompute its label from its justifications and their antecedent nodes when a change is propagated. This involves unnecessary computation. The ATR ATMS bases propagation on implications, which is faster. As explained above, the propagated change contributed by an implication is unioned into the label of the implication's

consequent; there is no need to examine the sister implications contributing to the consequent. Some previous ATMSs have represented their labels using lists, which require list computations. This ATMS uses bit vectors to represent labels; as a result, label computations are extremely fast. In particular, the important subsumption test is represented as two accesses and a single bit-vector operation, resulting in extremely efficient operation.

# B Discussion of Use of the ATMS

## B.1 The ATMS's capabilities

### B.1.1 Atomic Data

The ATMS stores, in each node, data that is useful to the user system. However, the ATMS *never examines this data.* The ATMS treats the data as an atomic assertion, and only works with whether the *node* containing the data is believed or not. This means that the user system can store anything inside the node, including facts, lists, hash tables, functions, or other kinds of complex data structures. The data is treated as an atomic assertion, and assigned a belief value. Later on the user system can access this data, perhaps expanding it or modifying it inside the user system, creating a new ATMS node to store the results of the modification.

Most typically, the data stored inside a node is a sentence, representing a segment of a semantic network.

### B.1.2 Positive Data

The ATMS only stores positive data. The data inside a node is either "believed true" or "not believed true"("no opinion"), but the ATMS cannot represent "believed not true".

In order to represent the negation of data "X", the user system must explicitly create a separate node with the data "(not X)" inside it (remember that we can put anything in the data of a node that we want). Then, when this node is believed by the system, the ATMS is representing "believed (not X)". (Of course, when this node is not believed by the system, it represents "not believed (not X)", which is still "no opinion"–it does not mean "believed X".)

### B.1.3 Constant Data

The data stored in the ATMS is treated as constant, since it is not used by the ATMS. The ATMS cannot represent variables, and there is no way to have the ATMS instantiate different values, match patterns, or perform other manipulations requiring variables. That is the job of the user system.

### B.1.4 Finite Problem

Since an ATMS explicitly represents all important instantiations of the user system's variables, the problem must be finite.

## B.2 What are the Strengths and Weaknesses of an ATMS?

An ATMS is useful in deterministic problems. ATMSs are also useful when there is just one solution that must be found, or several solutions that must all be found. One of the strengths of an ATMS is its ability to represent and reason about many different possible worlds simultaneously, where these worlds can be conflicting or contradictory. Naturally, an ATMS is useful when truth maintenance must be done, i.e. when the belief or disbelief of a particular concept has consequences that propagate to other concepts; thus, an ATMS can be employed in problems when there is binary evidence. It is particularly useful when a concept's belief value will be changed back and forth many times. ATMSs can also be employed with advantage in problems where concepts have justifications, and where explanations for the belief in a particular concept are desired.

However, since an ATMS represents discrete concepts that propagate truth values in an all-or-nothing manner, the ATMS itself is not good at stochastic problems, or those dealing with continuous variables that take on ranges of important values. Also, for the same reason the ATMS itself is not good at weighted evidential reasoning, where some possibilities must be chosen over other possibilities because of slightly larger evidence scores. In addition, the ATMS automatically finds all significant possible solutions; thus, it is wasteful when there is a large class of solutions, only one of which needs to be found.

Besides these, an ATMS by itself cannot represent possible worlds that have the same state but different histories. It can represent definite nonmonotonic actions, simultaneous possible states, or possible monotonic temporal actions, but it cannot represent *possible nonmonotonic actions*. For this reason, an ATMS must be augmented if it is to reason about actions over time.

## B.3 Summary of Conceptual User Operation of the ATMS

- The ATMS is given a set of assumptions.
- The ATMS given a set of implications.
- These are supplied one at a time.
- The ATMS incrementally updates itself as assumptions and implications come in:
- The ATMS determines the contexts.
- It answers: When has a context become inconsistent?
- It answers: Does a node hold in this context?
- It answers: What is in this context?
- It remembers all partial results of the user system; these do not have to be rederived, e.g. when switching search-spaces.
- The user system can switch states back and forth easily by temporarily changing assumptions.

As was mentioned before, it is important to note that although assumptions are structures in the problem solver, they are *atomic* in the ATMS. Problem solving is the process of accumulating implications and changing beliefs until some goal is met. The ATMS only knows what the user tells it.

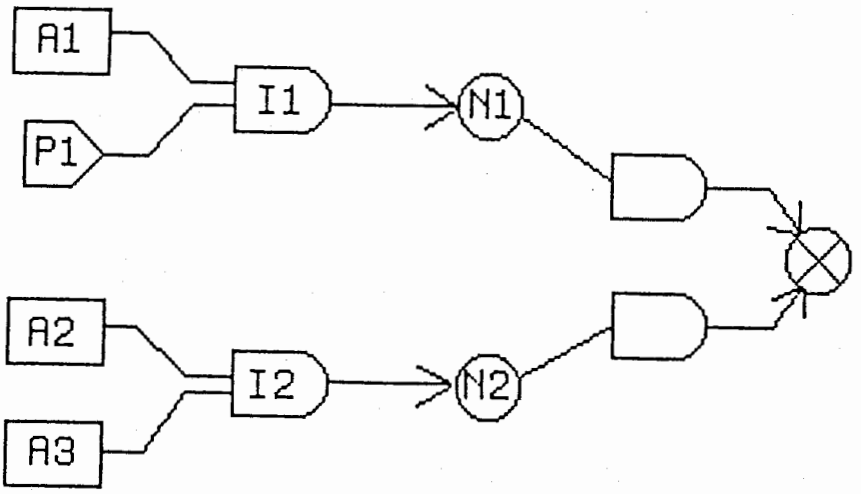



Figure 9: Network for Example 1.

# C Further Examples

## C.1 Example 1: Two Implications

The network for this example is shown in Figure 9.

```
This demo demonstrates the capabilities of the ATMS system.
The variable *watch-atms* is T, so whenever a new item
is added to the ATMS, the ATMS prints it out.
Naturally, the first thing to do is to reset the ATMS.

(reset-atms)
Atms-node NOGOOD-NODE.
(setq A1   (assumption 'A1)   )
Assumption Atms-node A1.
Assuming node #<Atms-Node 1> : A1.
Implication ASSUMED: A1 <= #<Assum 1>

(setq P1   (premise 'P1)    )
Premise Atms-node P1.

(setq N1   (atms-node 'N1)   )
```

```
Atms-node N1.

(setq I1    (implication N1 (list P1 A1) 'I1)  )
Implication I1: N1 <= #<Atms-Node 2> #<Atms-Node 1>

(setq A2   (assumption 'A2)   )
Assumption Atms-node A2.
Assuming node #<Atms-Node 4> : A2.
Implication ASSUMED: A2 <= #<Assum 2>

(setq A3   (assumption 'A3)    )
Assumption Atms-node A3.
Assuming node #<Atms-Node 5> : A3.
Implication ASSUMED: A3 <= #<Assum 3>

(setq N2   (atms-node 'N2)   )
Atms-node N2.

(setq I2    (implication N2 (list A2 A3) 'I2)  )
Implication I2: N2 <= #<Atms-Node 4> #<Atms-Node 5>

(print-atms)
Nodes:
  #<Atms-Node 0>: NOGOOD-NODE
  #<Atms-Node 1>: A1
  #<Atms-Node 2>: P1
  #<Atms-Node 3>: N1
  #<Atms-Node 4>: A2
  #<Atms-Node 5>: A3
  #<Atms-Node 6>: N2
Assumptions:
  #<Assum 1>: #<Atms-Node 1>
  #<Assum 2>: #<Atms-Node 4>
  #<Assum 3>: #<Atms-Node 5>
Implications:
  #<Implic 1>: #<Atms-Node 1> <= #<Assum 1>
  #<Implic 2>: #<Atms-Node 3> <= #<Atms-Node 2> #<Atms-Node 1>
  #<Implic 3>: #<Atms-Node 4> <= #<Assum 2>
  #<Implic 4>: #<Atms-Node 5> <= #<Assum 3>
  #<Implic 5>: #<Atms-Node 6> <= #<Atms-Node 4> #<Atms-Node 5>
Environments:
  #<Env 0>: {}
  #<Env 1>: {#<Atms-Node 1>}
  #<Env 2>: {#<Atms-Node 4>}
```

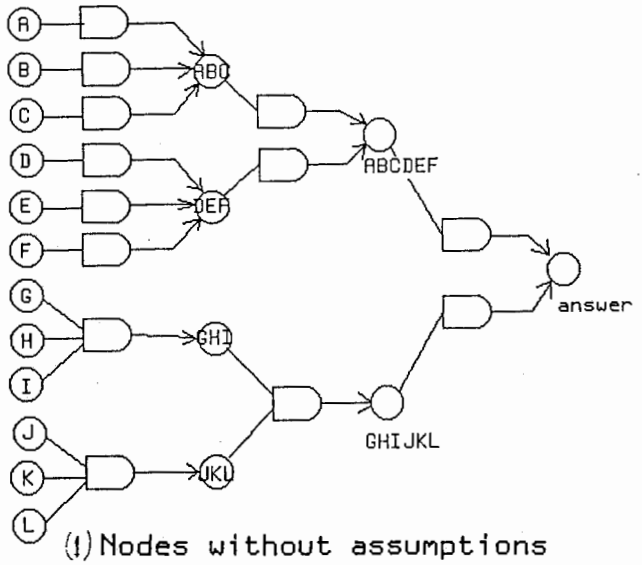
```
  #<Env 3>: {#<Atms-Node 5>}
  #<Env 4>: {#<Atms-Node 4>, #<Atms-Node 5>}

(explain-nodes)
NOGOOD-NODE is out.
A1 is in, under (#<Env 1>)
P1 is in, under (#<Env 0>)
N1 is in, under (#<Env 1>)
A2 is in, under (#<Env 2>)
A3 is in, under (#<Env 3>)
N2 is in, under (#<Env 4>)

(nogood   N1)
Implication NOGOOD: NOGOOD-NODE <= #<Atms-Node 3>

(nogood   N2)
Implication NOGOOD: NOGOOD-NODE <= #<Atms-Node 6>

(print-atms)
Nodes:
  #<Atms-Node 0>: NOGOOD-NODE
  #<Atms-Node 1>: A1
  #<Atms-Node 2>: P1
  #<Atms-Node 3>: N1
  #<Atms-Node 4>: A2
  #<Atms-Node 5>: A3
  #<Atms-Node 6>: N2
Assumptions:
  #<Assum 1>: #<Atms-Node 1>
  #<Assum 2>: #<Atms-Node 4>
  #<Assum 3>: #<Atms-Node 5>
Implications:
  #<Implic 1>: #<Atms-Node 1> <= #<Assum 1>
  #<Implic 2>: #<Atms-Node 3> <= #<Atms-Node 2> #<Atms-Node 1>
  #<Implic 3>: #<Atms-Node 4> <= #<Assum 2>
  #<Implic 4>: #<Atms-Node 5> <= #<Assum 3>
  #<Implic 5>: #<Atms-Node 6> <= #<Atms-Node 4> #<Atms-Node 5>
  #<Implic 6>: #<Atms-Node 0> <= #<Atms-Node 3>
  #<Implic 7>: #<Atms-Node 0> <= #<Atms-Node 6>
Environments:
  #<Env 0>: {}
XX#<Env 1>: {#<Atms-Node 1>}
  #<Env 2>: {#<Atms-Node 4>}
  #<Env 3>: {#<Atms-Node 5>}
```

```
XX#<Env 4>: {#<Atms-Node 4>, #<Atms-Node 5>}

(explain-nodes)
NOGOOD-NODE is out.
A1 is out.
P1 is in, under (#<Env 0>)
N1 is out.
A2 is in, under (#<Env 2>)
A3 is in, under (#<Env 3>)
N2 is out.
```

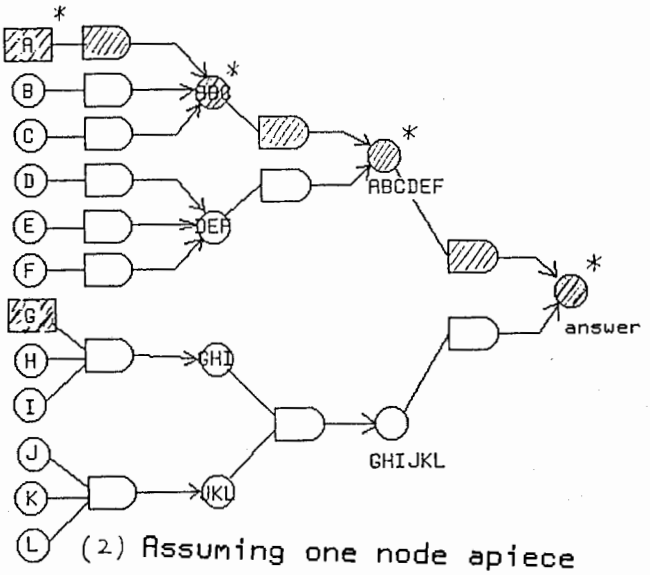
## C.2 Example 2: AND and OR Networks

The network for this example is shown in Figure 10. This example is more complex, but it amply demonstrates the capabilities of the system. The example experiments with both a network of AND connections and a network of OR connections. As the output from this program is too long to reasonably put in a manual, this example is left as an exercise for the user.

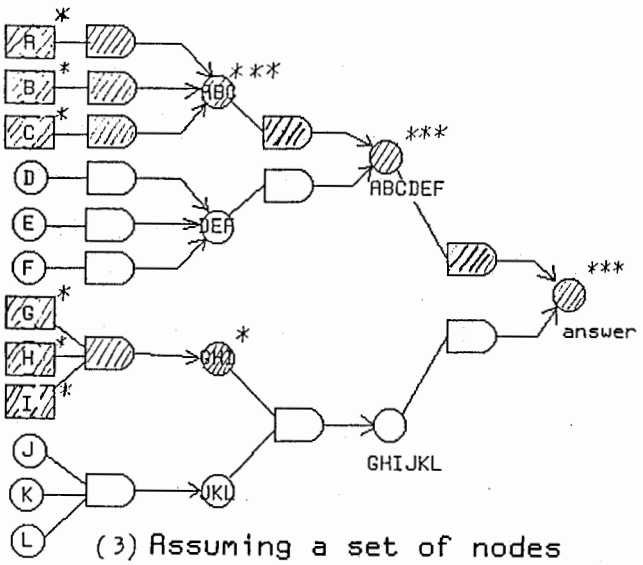
```
(reset-atms)

(setq a (atms-node "A"))
(setq b (atms-node "B"))
(setq c (atms-node "C"))
(setq d (atms-node "D"))
(setq e (atms-node "E"))
(setq f (atms-node "F"))
(setq g (atms-node "G"))
(setq h (atms-node "H"))
(setq i (atms-node "I"))
(setq j (atms-node "J"))
(setq k (atms-node "K"))
(setq l (atms-node "L"))

(setq ABC (atms-node "ABC"))
(setq DEF (atms-node "DEF"))
(setq GHI (atms-node "GHI"))
(setq JKL (atms-node "JKL"))

(setq ABCDEF (atms-node "ABCDEF"))
(setq GHIJKL (atms-node "GHIJKL"))
```



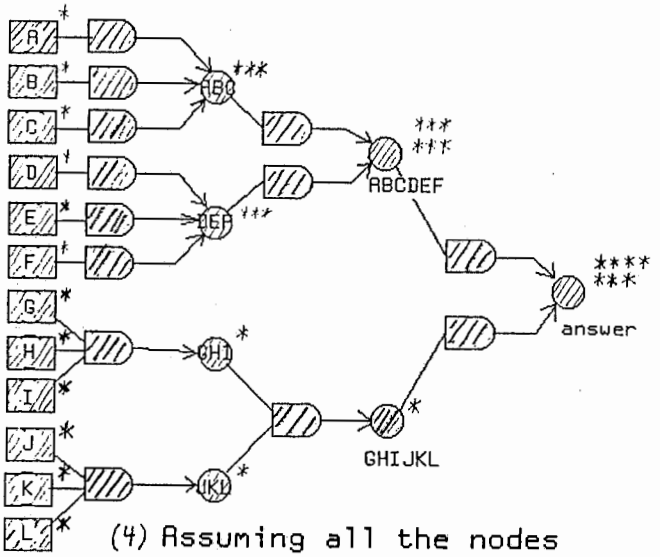

(1) Nodes without assumptions



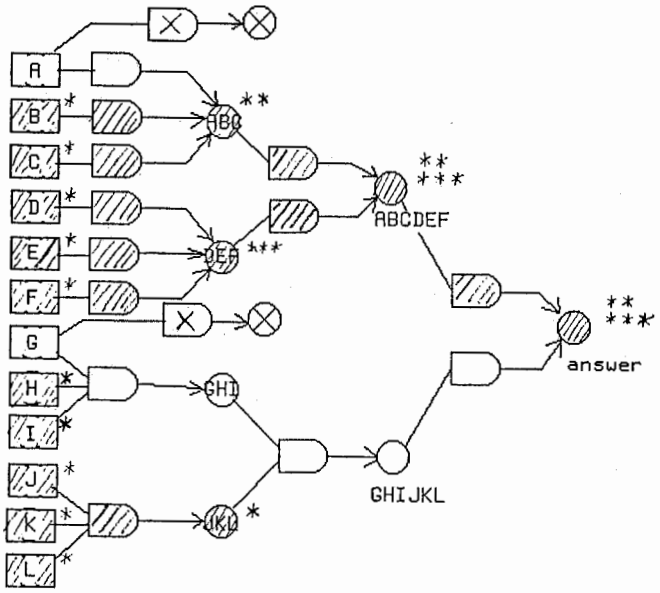

(2) Assuming one node apiece



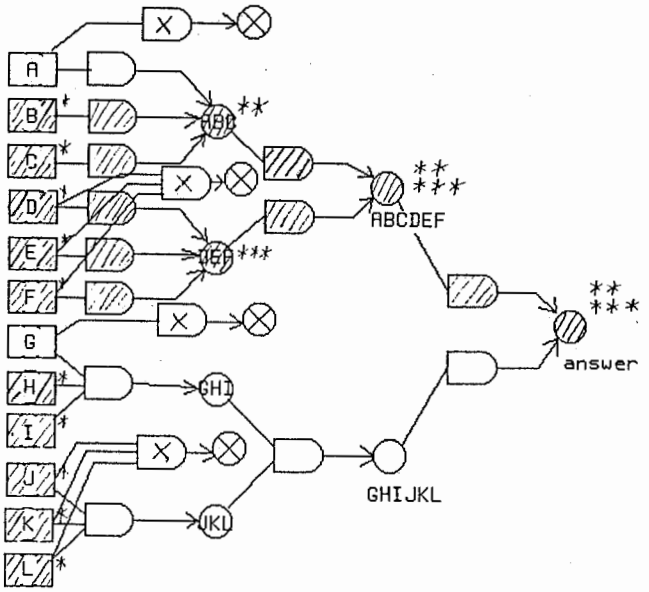

(3) Assuming a set of nodes




(4) Assuming all the nodes




(5) Setting one node apiece to nogood




(6) Setting sets of nodes to nogood

Figure 7. Network for Example 2.

```
(setq answer (atms-node "answer"))

(Implication  ABC 'Imp1  A)
(Implication  ABC 'Imp2  B)
(Implication  ABC 'Imp3  C)

(Implication  DEF 'Imp4  D)
(Implication  DEF 'Imp5  E)
(Implication  DEF 'Imp6  F)

(Implication  GHI 'Imp7  G H I)

(Implication  JKL 'Imp8  J K L)

(Implication  ABCDEF 'Imp9   ABC)
(Implication  ABCDEF 'Imp10  DEF)

(Implication  GHIJKL 'Imp11  GHI JKL)

(Implication  answer 'Imp12  ABCDEF)
(Implication  answer 'Imp13  GHIJKL)

(print-atms)
(explain-nodes)

(assume-this-node A)
(explain-nodes)

(assume-this-node G)
(explain-node G)
(explain-node GHI)

(assume-this-node B)
(assume-this-node C)
(explain-nodes)

(assume-this-node H)
(assume-this-node I)
(explain-node GHIJKL)
(explain-node GHI)


(assume-this-node D)
```

```
(assume-this-node E)
(assume-this-node F)

(assume-this-node J)
(assume-this-node K)
(assume-this-node L)

(print-atms)
(explain-nodes)


(nogood A)
(nogood G)

(print-atms)
(explain-nodes)


(nogood-set  D E F)
(explain-nodes)

(nogood-set  J K L)
(explain-nodes)

(print-atms)
```

# D Command Dictionary

**(add-assums-to-env old-env assumptions ...)** Creates (if necessary) and returns a new environment consisting of the assumptions of the old environment plus the new series of assumptions. Currently returns nil if new environment is nogood. Does not affect the old environment.

**(all-node-envs node)** Returns a list of *all* of the known consistent environments under which a given node is believed. This function is slightly expensive.

**(assume-this-node node)** Turns an ATMS-node into an assumption. (Technically, justifies the node with a new assumption-tag whose data contains the node.) Returns the node. Typically used only for effect. Of course, the user should not call this on nodes that are already assumptions or premises.

**(assumption data)** Constructs and returns an Assumption node storing the given information.

**(Assumption# n)** Accessor functions for assumptions.

***assumption-count*** The number of assumptions known to the system.

**(assumption-data assum)** Returns the data stored in an assumption.

**(assumption-ID assump)** ID number function for assumptions. Returns NIL if not an assumption.

**(assumption-p node)** Tests whether object is an assumption (i.e., an assumed node) or not.

***assumptions*** This variable stores a list of all the assumptions known to the system.

**(Assum# n)** Accessor functions for assumptions.

**(atms-node data)** Constructs and returns an ATMS node representing the given information. The nodes are numbered serially. Note: Node 0 is always the NOGOOD-NODE.

**(ATMS-Node# n)** Accessor functions for ATMS-nodes. These functions return the node, given the ID number for it. Same as (node# n).

***atms-node-count*** The number of ATMS-nodes, including those that have been turned into assumptions or premises, known to the system.

**(atms-node-data node)** Returns the data stored in a node.

**(atms-node-ID node)** ID number function for nodes.

**(atms-node-p node)** Tests whether object is an ATMS-node or not. NOTE: "assumptions" (assumed nodes) and premises are also ATMS-nodes.

***atms-nodes*** This variable stores a list of all the ATMS-nodes known to the system. This includes the assumptions and the premises.

**(characterizing-env env)** Returns the characterizing environment of the given environment (possibly itself). Returns nil if inconsistent.

**(context env)** Returns a list of the nodes in an environment's context, including the ATMS-nodes, the assumptions, and the premises. Works even if the context is invalid. This is an expensive function to call.

**(create-env assum-list)** Creates a new environment for the system to keep track of and follow, consisting of the set of all the assumptions in the given assumption-list. Returns the environment. Returns the old environment instead of creating it if previously there. Currently returns nil if new environment is nogood. If an ATMS-node in the assumption list was not in fact previously an assumption, it is *assumed* by this function. Note that this side-effect should be used with care.

***debug-atms*** This flag makes the system print out debugging information. Default is nil.

**(dont-use assum-list env-list)** Returns a list of environments where environments containing any of the given assumptions have been deleted.

**(dont-use-nodes nodes envs)** Returns a list of environments where environments whose context contains any of the given nodes have been deleted. A rather expensive function.

**(env-assums env)** Returns a list consisting of the assumptions that are BELIEVED in a given environment. Does not check whether environment is inconsistent or not. Note that more, derived ATMS-nodes will be believed under this environment, in the environment's context.

**(Environment# n)** Accessor function for environments.

***environment-count*** The number of environments known to the system.

**(environment-ID env)** ID number function for environments.

***environments*** This variable stores a list of all (both valid and inconsistent) of the environments known to the system.

**(Env# n)** Accessor function for environments.

**(env-nogood-p env)** Tests whether env is nogood.

**(explain-node node)** Gives environments in which node is IN.

**(explain-nodes)** Runs explain-node on all the nodes.

**(find-env assum-list)** Finds and returns an existing environment. Returns `nil` if it did not exist previously. Does not create any new environments. This is a fast function.

**geometric-limit-increase** This flag tells whether *incremental-assumption-limit* doubles after every expansion (geometric increase) or stays constant (arithmetic increase). This number indirectly affects memory allocation, paging, and performance. Default is T.

**(Implic# n)** Accessor functions for implications.

**(implication consequent data antecedents)** Constructs and returns an implication. Same as `(justification ...)`.

**(Implication# n)** Accessor function for implications.

***implication-count*** The number of implications known to the system.

**(implication-data impl)** Returns the data stored in an implication.

**(implication-ID implic)** ID number function for implications.

**(implication-p imp)** Tests whether object is an implication or not.

***implications*** This variable stores a list of all the implications known to the system. Each assumption internally generates an implication; these are included as well.

**(inconsistent-p env)** Returns T if given environment is NOGOOD (INCONSISTENT), nil otherwise. An environment is NOGOOD if the `*nogood-node*` is BELIEVED because of it (i.e., in its context). Same as `nogood-p`.

**(in-context-p node env)** If the given node is in the given environment's context, returns a (usually smaller) characterizing environment describing why that node is believed. Otherwise, returns nil.

***incremental-assumption-size*** This number tells how much the system's bit-vector size is increased during the next growth cycle. See `*initial-assumption-limit*`. This number indirectly affects memory allocation, paging, and performance. Default is 50.

**(inference consequent data antecedents)** Constructs and returns an implication (inference). Same as implication.

***initial-assumption-limit*** This number gives a soft limit on the number of *assumptions* that the system can store. It is used to determine the initial size of the assumption-bit-vector assigned to each environment. It must be set before calling `(reset-atms)`. Set this to the reasonable maximum number of assumptions expected to be handled by the system. This number affects memory allocation, paging, and performance. Default is 200.

**(IN-p node)** Tests whether node is IN. Returns a list of consistent environments entailing the node (the label) if the node is IN; returns nil if the node is OUT. This is the recommended function to use when tracing a node with a user-program.

**(install-action node action)** Installs the command `(action)` into the given node. If the given node becomes IN, (i.e., believed in *any* valid context), the given action command is executed.

**(in-world-p node env)** Same as `in-context-p`.

**(justification consequent data antecedents)** Constructs and returns an implication (justification). Same as implication.

**(Justification# n)** Accessor function for implications.

**(justification-data just)** Returns the data stored in an implication.

**(justification-ID just)** ID number function for implications.

**(Just# n)** Accessor function for implications.

**(Node# n)** Accessor functions for ATMS-nodes. These functions return the node, given the ID number for it. Same as (atms-node# n). Note that `(Node# 0)` returns the NOGOOD node.

**(node-envs node)** Returns a list of the minimal environments under which the given node is believed.

**(node-label node)** Returns a list of the minimal environments under which the given node is believed.

**(nogood node1)** Builds a justification from the node to *nogood-node*. Standard method of entering contradictions, which is the same as permanently making the node's data false. This function can also be called with a sequence of nodes, in which case each node in the sequence is set to NOGOOD.

**(nogood-env env)** Forces the given environment (and all of its supersets) to become NOGOOD. Calls `nogood-set` on the (conjunction of the) set of assumptions composing the environment. In general, this should be used only because of higher-level knowledge not part of the knowledge represented in the ATMS.

***nogood-node*** This variable stores the NOGOOD node. This node is allocated on reset. Note that `(Node# 0)` also returns this node.

**(nogood-p env)** Returns T if given environment is NOGOOD (INCONSISTENT), nil otherwise. An environment is NOGOOD if the `*nogood-node*` is BELIEVED because of it (i.e., in its context). Same as `inconsistent-p`.

**(nogood-set node1 node2 etc)** Builds a justification to *nogood-node* based on the *conjunction* of the given nodes. Standard method of entering contradictions. Note carefully that `(nogood-set)` of a set of nodes, which contradicts the AND of the set, is not the same as `(nogood)` of each of the members of the set, which contradicts the OR of the set.

**(OR-env env1 env2)** Returns an environment consisting of the union of the assumption sets from the two given environments. This may be inconsistent, even if both of the previous two are not. Such an environment might not be a characterizing environment.

**OS** This variable holds the Output Stream for the print functions. Default is T, meaning standard screen output stream.

**(OUT-p node)** Tests whether node is OUT. Returns T if OUT, NIL otherwise.

**(premise data)** Constructs and returns a Premise node storing the given information.

**(Premise# n)** Accessor function for premises. This function returns a premise. Since premises are really ATMS-nodes, this is the same as Node#.

***premise-count*** The number of premises known to the system.

**(premise-data node)** Returns the data stored in a premise.

**(premise-ID node)** ID number function for premises. Same as (atms-node-ID).

**(premise-p node)** Tests whether object is a premise or not.

***premises*** This variable stores a list of all the premises known to the system.

**(premise-this-node node)** Turns an ATMS-node into a premise. Technically, overwrites the label with the single, empty environment *TRUTH-ENV*. Same as (presume-this-node).

**(presume-this-node node)** Turns an ATMS-node into a premise. Technically, overwrites the label with the single, empty environment *TRUTH-ENV*. Same as (premise-this-node).

**(print-assum assum)** Prints an assumption.

**(print-assums)** Prints a list of all the assumptions, and the corresponding nodes.

**(print-atms)** Dumps everything. Use this to get used to the system.

***print-data*** When this flag is T, the print functions print out the data inside nodes and assumptions. When it is nil, the print functions only print out a numbered node. Set this to nil when very long data is stored in nodes. Default is T.

**(print-implic implic)** Prints a given implication.

**(print-implics)** Prints a list of all the implications, including assumption justifications.

**(print-env env)** Prints an environment.

**(print-envs)** Prints a list of all the environments.

**(print-node node)** Individual item printing functions.

**(print-nodes)** Prints a list of all the nodes, and their data.

**(reset-atms)** Clears the system out.

**(sig-envs env-list)** Returns a list of environments where subset and inconsistent environments have been eliminated. Defaults to using `*environments*`, all of the known environments, as input if no argument is given.

**(significant-envs env-list)** Returns a list of environments where subset and inconsistent environments have been eliminated. Defaults to using `*environments*`, all of the known environments, as input if no argument is given.

**(subsumed-by-p larger-env smaller-env)** Tests to see whether larger-env is subsumed by (is a superset of) smaller-env. Returns `T` if subsumed, `nil` otherwise. Extremely fast.

***truth-env*** This variable stores the empty environment. This environment's context contains all the premise nodes; it is always true.

**use-uniquification** This flag tells whether ATMS data is treated as being unique (under `equal`) or whether it can be duplicated. If unique, `(atms-node data)` and similar functions will return a previously created node instead of creating a new one. Default is T.

***watch-atms*** This flag makes the system print out a notification each time an item is created. Default is T.

***watch-enlarge*** This flag makes the system print out a message when the system enlarges the bit-vector arrays for assumptions. Default is T.

**(why-assumptions node env)** Explains the assumptions that directly or indirectly contribute to the given node under the given environment. Returns a list of all the BELIEVED assumptions that justify the node in the environment's context.

**(why-env-assums node)** Explains the different assumption sets that this node is BELIEVED in. Instead of returning a list of environments justifying this node, like `why-envs`, this function returns the environments' assumption sets, in the form of a list of lists of assumptions.

**(why-envs node)** Returns a list of the consistent environments under which (in whose context) this node is BELIEVED.

**(why-implications node env)** Explains the contributing immediate implications that make the given node believed under the given environment. Returns a list of all the active implications that *directly* actually justify the given node in the given environment's context. Does not return implications that indirectly justify the node, or potentially justify the node but are inactive. Returns the system-generated justification for an assumption.

**(why-nodes node env)** Explains the contributing immediately preceding nodes that make the given node believed under the given environment. Returns a list of all the believed nodes that *directly* justify the given node in the given environment's context.

**(why-nogood-assumptions env)** Explains the assumptions that directly or indirectly contribute to NOGOOD under the given environment. The environment should be inconsistent. This is a very useful function, as it returns only the mutually conflicting assumptions that are causing the problem with an inconsistent environment.

**(why-nogood-implications env)** Explains the implications that immediately contribute to the `*nogood-node*` under the given environment. The environment should be inconsistent. Returns a list of the active implications that actually justify the *NOGOOD-NODE* in the environment's context.

**(why-nogood-nodes env)** Explains the immediately preceding nodes that contribute to making the `*nogood-node*` believed under the given environment. The environment should be inconsistent.

# References


[dK86a] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28(2):127–162, March 1986.

[dK86b] Johan de Kleer. Extending the atms. *Artificial Intelligence*, 28(2):163–196, March 1986.

[dK86c] Johan de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28(2):197–224, March 1986.

[JdKW87] Kenneth Forbus Johan de Kleer and Brian Williams. Aaai'87 tutorial on truth maintenance systems. In *AAAI'87: The Sixth National Conference on Artificial Intelligence*, Seattle, WA., 1987. Tutorial No. TA 4.

[MM88] David McAllister and Drew McDermott. Aaai'88 tutorial on truth maintenance systems. In *AAAI'88: The Seventh National Conference on Artificial Intelligence*, St. Paul, MN., 1988. Tutorial No. MP1.

[WN88] John R. Walters and Norman R. Nielsen. *Crafting Knowledge-Based Systems: Expert Systems Made (Easy) Realistic*. John Wiley & Sons, New York, NY, 1988. A good explanation of non-monotonic irreversible actions.