TR-I-0369

The B-SURE MANUAL
Version 2.3
B-SURE マニュアル

John K. Myers

March 12, 1993

Abstract

This manual presents user documentation for the ATR Interpreting Telephony Research Laboratories B-SURE representation system. B-SURE, standing for "the Believed Situation and Uncertain-Action Representation Environment", is a system that is able to represent and store situations, states, and actions, in multiple possible action worlds. B-SURE is a general-purpose system that can work with any application that requires representing multiple possible actions. The resulting system is useful for such natural language tasks as planning, plan recognition, and parallel task scheduling.

THE B-SURE MANUAL

Version 2.3

John K. Myers

ATR Interpreting Telephony Research Laboratories
Sanpeidani, Inuidani, Seika-cho, Soraku-gun
Kyoto 619-02 Japan
Netmail: myers@atr-la.atr.co.jp

Abstract

This manual presents user documentation for the ATR Interpreting Telephony Research Laboratories B-SURE system. B-SURE, standing for "the Believed Situation and Uncertain-Action Representation Environment", is a system that is able to represent and store situations, states, and actions, in multiple possible action worlds. The actions can have *nondeterministic* outcomes--that is, an action can have many possible outcomes, only one of which will become true. In addition, the actions can have outcomes that are *nonmonotonic*, that is, they can retract states. The system supports explicit representations of actions and situations used in intentional action theory and situation theory. Agents have free will as to whether to choose to perform an action or not. Both types and instances are supported, for situations, actions, and states. The system can perform global reasoning simultaneously across multiple possible worlds, without being forced to extend each world explicitly, by using implications. B-SURE is a general-purpose system that can work with any application that requires representing multiple possible actions. The resulting system is useful for such natural language tasks as planning, plan recognition, and parallel task scheduling.

Acknowledgment

This research was supported by ATR Interpreting Telephony Research Laboratories. I would like to express my gratitude to Dr. Akira Kurematsu and to Mr. Hitoshi Iida for managing and supporting this research. I would also like to acknowledge the friendliness and the helpfulness of the people in the Natural Language Understanding Department.

Contents

List of Figures

1 Working with B-SURE

The BSURE system is contained in the directory `LM01:>myers>BSURE>*`. Besides the files in this directory, it also uses standard files `LM01:>myers>system` and `LM01:>myers>atms5`. The BSURE system is loaded completely by loading file `LM01:>myers>BSURE>Load-BSURE-system`.

An example of exactly how to use the BSURE system is presented in Appendix C starting on page 43. Examples of how to use the BSURE system are also contained in the file `LM01:>myers>BSURE>TIAMAT-rep.lisp` which is also in the BSURE directory. The functions `(B-test-5)`, `(B-test-6)`, and `(B-test-7)` are the best ones to call here. Note that these functions call `(draw-graph)` as one of the last things that they do.

The BSURE system is written in Common Lisp and runs on at least a Symbolics platform. Care was taken to make sure that the Common Lisp is mostly machine-independent, and the system should be able to run on other platforms such as the Sun with little or no modifications. Of course, this would require a Common Lisp ATMS, such as `SQ:/usr2/myers/nl/atms5-sq`. The BSURE graphics system, which is also loaded, is highly Symbolics-dependent (e.g., it uses presentations, and a special Symbolics graph-drawing algorithm) and cannot be ported to other machines but must be duplicated.

2 Introduction

This manual describes the ATR Interpreting Telephony Research Laboratories' B-SURE (*B*elieved *S*ituation and *U*ncertain-action *R*epresentation *E*nvironment) system, version 2.3. B-SURE is a data-base that is able to represent and store situations, actions, states, and implications, in multiple action possible worlds. Unlike the previous version of the ATMS, which was unable to represent nonmonotonic actions correctly, the B-SURE system fully supports actions that can delete states. The main task of the B-SURE system is to *represent actions and situations correctly*, including actions that can have more than one possible outcome, different future actions that an agent *might* perform, past actions that an agent definitely *has* performed, and different possible-world timeline histories based on what could happen. The key to good reasoning is a powerful representation system that is able to accurately model details of a problem. Once a good representation has been established, problem computations often become straightforward.

Recent advances in situation theory [BP83,Bar89] and the theory of intentions [Bra87] have offered many new insights on significant problems found in natural-language understanding. However, these theories offer philosophical approaches only, and do not give instructions for building concrete representation and reasoning engines. At the same time, the software systems that have been built for reasoning and representation fall short in any number of areas. Production systems and semantic networks can follow chains of inferences, but can only represent one possible world at a time–they cannot reason with states that are both possibly true and

possibly not true, while keeping the chains of resulting inferences separate. Most planners work with limited possible worlds, but cannot reason and perform inferences across multiple worlds at the same time. The classical ATMS[1] can represent and reason with multiple timeless possible worlds, but cannot represent actions [dK86a]– in particular, nonmonotonic actions where a retracted state is both believed to be true in the world before the action takes place, and believed to be *not* true in the world representing the situation after the retracting action has taken place, cannot be represented. In addition, the ATMS only represents propositions that are instantiated constants or Skolem constants; it does not represent uninstantiated variables. A modified ATMS that can represent nonmonotonic transitions between worlds has been developed [MN86], but this system does not explicitly represent situation types and instances, action events, nor nondeterminism. Most plan inference systems have ignored free will and the explicit representation of the right to choose actions, e.g. to choose to be uncooperative. Almost all previous systems have ignored the nondeterministic quality of real-world actions that necessitates commitment in intentions. Real actions can result in one of several possible outcome situations, whereas almost all previous systems are are completely unable to model nondeterministic outcomes. Only decision-analysis systems have modeled expected values of actions, and they do not support inferencing. See [BL85] for an excellent summary of issues.

The B-SURE (Believed Situation and Uncertain-action Representation Environment) package is an implemented system that supports representation, planning, decision-making, and plan recognition using probabilistic and uncertain actions with nondeterministic outcomes in multiple possible action worlds. Situations, states, and action events are all represented explicitly, using types (variables) and instances. The B-SURE system is implemented as a series of extensions to a classical ATMS. The resulting system is very useful, and is being used in plan recognition, intentional agent, and scheduling research.

The user specifies state types, situation types that use those state types, uncertainties or probabilities, transitions that use those uncertainties and situation types, and action types that use those transitions. The user then instantiates one or more situation types to represent the starting situations. The user then instantiates an action type in a given situation instance, producing an action instance. Each action type has a precondition situation type. Two modes are possible; in the automatic mode, the system has the responsibility of checking that the precondition situation type is valid before instantiating an action, whereas in the manual mode, the user has this responsibility. The system automatically instantiates the action's possible outcome situations and returns the action, which has pointers to the previous situation instance, the Chooses node, the outcome situations, their transitions, and their Happens nodes. The user's system can then reference information in the resulting structure, such as whether a state is true or possible in a given world or not, and reason with the represention to produce useful results.

[1] Assumption-Based Truth Maintenance System [dK86a]

2.1 How to read this manual

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 the B-SURE system does, and a discussion of the types of data structure objects that the system works with to represent problems.

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 B-SURE system or an ATMS before. The user should be able to read the manual, run the examples, and afterwards understand how to use the system. 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. A deeper understanding of the ATMS used to support the B-SURE system can be found by reading the ATMS manual [Mye89b]. In fact, it is necessary to read this in order to learn about the five-valued logic, consisting of `hypothetical`, `possible`, `actual`, `inconsistent`, `null`. However, this manual is designed to be mostly self-contained.

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.

Action An **action** in B-SURE represents a transition from one *situation* to another situation. However, in a *nondeterministic action*, the actual *resulting situation* is unknown until the action is executed. Actions have *types* and *instances.*

Action Instance A data structure representing an *instance* of an *action.* Action instances are represented by PAWs (Performing Action Worlds).

Action Type A data structure representing the definition of a *type* of *action.*

Action World A data structure, which is stored in an ATMS-node, that represents a timeline history of actions and situations in the form of an explicitly-represented *environment* bit-vector. This bit-vector contains a set of all of the Happens, Chooses, and Not-yet-deleted assumptions that are believed valid in the current situation instance or action instance. Action Worlds are necessary because a normal ATMS cannot support representation of nonmonotonic actions [MN86]. Types of **Action Worlds** include *State Action Worlds* and *Performing Action Worlds.*

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

Assertion A concept. An **assertion** is a description of the world. "**Assertions**" is the name for *states* that is used by many expert systems and by the ATMS system. Assertions are called *states* in the B-SURE system.

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.*

Belief Value Whether a node is *believed* or *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 A non-technical word. An idea about something. A concept can be represented by a *type*, an *instance*, a *proposition*, or some other data structure, etc. The *belief value* of a concept is usually represented by a *node*.

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.

Constituent Sequence Barwise and Perry's technical term for a *state* [BP83, p. 53]. In their language, a constituent sequence consists of a relation between individuals, objects, properties, and space-time locations.

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*.

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.

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*.

Deterministic Something that is **deterministic** is known ahead of time. It must happen. See *deterministic action* and *nondeterministic*.

Deterministic Action A **deterministic action** is one that has only one possible *outcome situation*. If the action is executed, the single outcome must happen. Previous planning systems and plan recognition systems have almost all used deterministic actions in their plans. Deterministic actions are often not good models of real-world actions.

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, Represented but not used by the system in supporting inferences. 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.*

Feature Structure A **feature structure** is a format used to represent information, consisting of pairs of features and feature values (which may be constants, variables, or nested feature structures) arranged in an unordered list between square brackets. Various formats of feature structures are possible. The following is an example of a feature structure:

```
[[reln want]
 [agen Caller]
 [obje [[reln say]
        [agen Caller]
        [obje "Hello"]]]]
```

Feature structures are one of the methods used to represent information in a description of a *state.*

First-Order Probabilities A **first-order probability** is a normal, typical *probability* (measure) of the type that is normally used by most people.

Formula A concept. A **formula** is a description of the world. "Formulas" is the name for *states* that is used by most theorem-proving systems. Formulas are called *states* in the B-SURE system.

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.

Implies A technical term that means that there exists an ATMS *implication* between a *node* representing a *concept* or a *conjunction* of nodes, forming the *antecedent*, and a single node forming the *consequent.* The implication supports reasoning in a matter corresponding to intuition–if the antecedents are *believed*, then the consequent is *believed* also.

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.

Influence An **influence** is a quantity that determines how likely it is that the *outcome* of a *nondeterministic action* is a particular *situation*. An **influence** is the main piece of information attached to a *transition*. An **influence** may be a customary **first-order probability** or it may be an uncertain **second-order probability**.

Instance An **instance** is a data-structure that represents a particular instantiation of a *type*. **Instances** are theoretically bound in space and time, although these variables are not supported by the system and must be provided by the user. Instances may be actual, possible, hypothetical, inconsistent, past, present, or in a possible future world. States, actions, and situations all have types and instances. Each instance *implies* its type. An instance can only be an instance of one type.

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 an atms-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.

Logical Form A **logical form** is a format used to represent information, consisting of atomic symbols and nested logical forms arranged in a list between parenthises. The following is a logical form: `(Wants Caller (Say "Hello"))` . Logical forms are one of the methods used to represent information in a description of a *state*.

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*.

Nondeterministic Something that is **nondeterministic** is not completely known or "determined" ahead of time. It may or may not happen. See *nondeterministic action* and *deterministic*.

Nondeterministic Action A **nondeterministic action** is one that has many possible *outcome situations*. If the action is executed, only one of these outcome situations will actually occur; however, it is unpredictable which one will happen. Previous planning systems and plan recognition systems have almost all used *deterministic actions* in their plans. Nondeterministic actions are often better models of real-world actions than deterministic actions are. See *Uncertain Nondeterministic Action*.

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 <u>not</u> 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.

Outcome The **outcome** of an action is the results of that action; what happens or occurs when the action is performed, as a direct result of performing the action. When a *nondeterministic action* is performed, the actual particular **outcome** is unknown ahead of time. The actual outcome may be one of several possible outcomes. In the B-SURE system, it is assumed that the user knows all the possible outcomes of an action. Although an **outcome** is technically a concept, it can also be used as a short name for *outcome situation.*

Outcome Situation An **outcome situation** is a particular situation that represents one possible *outcome* of (the results of performing) a *nondeterministic action.* A *nondeterministic action* will have more than one **outcome situations.** An *action type* will have *situation types* for its **outcome situations;** an *action instance* will have *outcome instances.*

p The symbol "p" is used to represent a (*first-order*) probability distribution that is defined over a set of outcomes in the world.

PAW See *Performing Action World.*

Performing Action World A special kind of data structure that represents an action instance that is being performed, along with its history; abbreviated PAW. A PAW is a type of *Action World.* It stores the history of Chooses and Happens assumptions, along with the Starting Situation assumption, that were necessary to have happen in order for the action instance it represents to become actual.

A PAW directly represents an action instance. There is no separate data structure to represent an action instance, since all action instances must be performed in a particular situation instance.

PNDA See *Probabilistic NonDeterministic Action.*

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.*

Precondition Situation A **precondition situation** is attached to an action. It logically determines whether the action can be executed in a given world or not. If that situation is true in that given world, then the action can logically be executed. B-SURE provides two levels of checking for precondition situations, set by a flag. For the first level, the system automatically checks precondition situations for each action instance when its instantiation is requested, and does not perform the instantiation if the precondition is not valid. For the second level, the precondition situation must be used by the user to check for validity; the system performs no checking by itself. Apart from this, precondition situations are not used in B-SURE version 2.3.

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.

Probabilistic NonDeterministic Action This is represented by the acronym **PNDA**. A **PNDA** is a *nondeterministic action* in which the *influences* are represented by *first-order probabilities.* The B-SURE system supports representation of PNDAs.

Probability The likelihood of a particular outcome occurring. Also, the mathematical science that deals with describing and working with this likelihood. Also, a short name for a first-order probability constant representing the measure of a likelihood.

Proposition A concept. A **proposition** is a description of the world. "**Propositions**" is the name for *states* that is used by some expert systems and some predicate-calculus systems. Propositions are called *states* in the B-SURE system.

q The symbol "q" is used to represent a probability distribution that is defined, not over a set of outcomes in the world, but over a set of possible probabilities of outcomes. It is the "second-order" part of a second-order probability. See *second-order probabilities.*

Relation Barwise and Perry's word for an operator that groups things together. Relations operate over individuals, objects, and space-time locations. A **relation** plus its arguments together form a *constituent sequence*, which is normally called an *assertion*, a *proposition*, or, in the B-SURE system, a *state.*

Resulting Situation Another name for an *outcome situation.* A resulting situation is the situation that results after an *action* is executed.

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.

SAW See *State Action World.*

Second-Order Probabilities A **second-order probability** is a probability measure that consists of a random variable **p** representing the value of a first-order probability ranging over the closed interval $[0, 1]$, together with a (second-order) probability distribution $q(\mathbf{p})$ defined over that interval. Second-order probabilities can explicitly represent uncertainty and confidence in estimates.

Situation (Barwise and Perry) Barwise and Perry [BP83, p. 49] use the term **situation** to cover a very large range of types of things, as is explained in Section 5. Situations are roughly divided along two axes, into *real situations*

and *abstract situations*, and also into *states of affairs* and *courses of events*. *Real situations* occur in the real world with real things, and therefore have no place in a computer model. *Abstract situations* occur as models, and therefore all computer situations are by definition "abstract". *States of affairs* denote static situations; these are simply called *situations* in B-SURE. *Courses of events* denote events that are nominalizations of the performance of actions, i.e. the transition from one state-of-affairs situation to another due to an action being performed; these are simply called *actions* or *action instances* in B-SURE.

Situation (B-SURE) A **situation** is a set of positive and negative states, each with a belief value. A **situation** corresponds to Barwise and Perry's *state of affairs*. Situations have types and instances.

Situation Instance A data-structure representing an *instance* of a *situation*. A Situation instance is stored by a SAW (State Action World) that keeps data on the history of the situation.

Situation Type A data-structure representing the definition of a *type* of *situation*.

State A concept. A **state** is a description of the world. States are sometimes called *propositions*, *formulas*, or *assertions*, and are used by most AI systems to represent knowledge. In the B-SURE system, a state can be represented by a *logical form*, a *feature structure*, or something else–a *user-defined state*. States are treated as atomic in the B-SURE system and in version 2.3 are not examined, other than to be printed out. In Barwise and Perry [BP83, p.53,50], states are called constituent sequences composed of relations that deal with individuals and space-time locations. A unary relation is known as a property.

States have *types* and *instances*.

State Action World A special kind of data structure that represents the history of a situation instance; abbreviated SAW. An SAW is a type of *Action World*. It stores the history of Chooses and Happens assumptions, along with the Starting Situation assumption, that were necessary to have happen in order for the situation instance it represents to become actual.

In the current version 2.3, a State Action World and the situation instance that it represents are two separate data structures, even though they both basically represent the situation instance. This may change in the future. The reason for the current theory is that it is possible to have a situation instance that is created indirectly by inferences from the states of different outcomes, which is different from the type of situation created directly from the immediate outcome of an action.

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.

Transition A **transition** is a change that occurs in the world between the execution of an action and the occurrence of one of the action's outcomes. Also, a data-structure that represents this change. A **transition** has a single *influence* and a single *outcome situation.* A *nondeterministic action* will have one **transition** for each of its *outcome situations.*

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.

Type A **type** is a data structure that defines a class of objects. A **type** may have any number of *instances.* Each instance *implies* the type. Situations, actions, and states all have types and instances.

Uncertainty The word **uncertainty** is used in this work in a technical sense to denote the general concept or an instance of a second-order probability.

Uncertain NonDeterministic Action This is represented by the acronym **UNDA**. An **UNDA** is a *nondeterministic action* in which the *influences* are represented by *uncertainties.* The B-SURE system supports representation of UNDAs.

UNDA See *Uncertain Nondeterministic Action.*

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.

User-Defined State A **user-defined state** is a piece of information that is used to represent a *state*, that is not a *feature structure* or a *logical form.* Since the B-SURE system does not work with the internal contents of states (other than to print them out when requested), the user is free to use whatever information is desired when creating a *state type.* It is up to the user to ensure that the information is in a usable form.

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 ATMS Data Types

In order to understand the B-SURE system well, it is useful to first review the ATMS system on which it is based. There are a number of explicit major kinds of data in the ATMS system. These are:

ATMS-node A node. The ATMS-node is the basic unit of the ATMS system. Nodes get assumed and presumed. Nodes have a basic belief value of BELIEVED or NOT BELIEVED in any *one* possible world. When looking at the universe of possible worlds as a whole, a node will take on the belief value of **actual, possible, hypothetical, inconsistent,** or **null**, depending upon its existence and its belief values in the various possible worlds. A node is used to store data such as states, situations, and actions.

assumption An assumption is a special kind of node that is used to justify other concepts. Assumptions are both BELIEVED and NOT BELIEVED. An assumption thus splits the universe into two new sets of possible worlds. Assumptions create environments.

premise A special kind of node that is always true. Premises are BELIEVED in all possible worlds, and are thus considered to be **actual**. Premises are implemented by having the empty environment (#0) as their label.

implication An AND GATE structure between nodes. An implication takes many *antecedents* and one *consequent*. If all of the antecedents are BELIEVED in a given possible world, then the consequent must be BELIEVED in that possible world as well. An Implication is sometimes called a Justification, a Constraint, or an Inference in other works in the literature.

the nogood node The nogood node is a single special node that represents the concept of inconsistency. Any pair of nodes that together imply the nogood node cannot both be BELIEVED in the same possible world. Such nodes are said to be *pairwise nogood*. Any set of nodes that together imply the nogood node cannot all be BELIEVED in the same possible world–they must have at least one node that is NOT BELIEVED. Such nodes are said to be *mutually inconsistent*. Any single node that directly implies the nogood node can never be believed in any possible world, and is said to be **inconsistent**. **It is a conceptual error to have a premise that directly implies the nogood node, or to have two premises that are pairwise nogood. This breaks the belief maintenance capability of the ATMS.**

environment A set of assumptions that define a possible world. Each assumption in the environment is BELIEVED under that environment. Environments are currently implemented as bit-vectors.

3.2 B-SURE Data Types and Major Concepts

There are a number of explicit major kinds of data in the high-level B-SURE system. These are:

state The *state* is the basic fundamental unit of the system. A state encodes a statement, predicate, or proposition about the world. A state can be *positive* (present) or *negative* (absent). States have *belief values* that determine whether the system believes that they could be possible in the future or that they have happened already.

situation A *situation* is a set of states. A situation is meant to correspond with the classification "state of affairs" introduced in Barwise and Perry [BP83].

action An *action* represents a change between one situation and another. An agent must *choose* to perform an action in order for the action to get started.

transition A *transition* represents the change from the performance of an action to a new (outcome) situation. An action has a list of outcome transitions. A transition has an influence and an outcome situation.

influence An *influence* represents the degree of likelihood that a particular transition will become actual, that is, how likely it is that execution of the action will result in the transition to a given outcome situation. Influences can be probabilistic or 2nd-order uncertainties.

UNDA This stands for Uncertain Non-Deterministic Action. An UNDA is an action that has more than one possible outcome situation (it is "nondeterministic").

type A *type* defines something in an abstract manner. States, situations, and actions all have types and instances.

instance An *instance* is a particular instantiation of a concept defined by a type.

3.3 Secondary Concepts

Chooses node A Chooses node is an assumption that is associated with an action that represents the fact that the agent *chooses* to execute the action. If the Chooses node is presumed true, it represents the fact that that the agent has chosen to begin executing the action, and that the action is now under progress. Often an agent will only be able to perform one action at one level at a time, and so the various Chooses nodes emanating from a particular situation will be made pairwise exclusive.

Happens node A Happens node is an assumption that is associated with a transition from an action performance to an outcome situation, that represents the fact that the given outcome situation in fact happens. A Happens node is the instantiation of a transition type. If the Happens node is presumed true, it represents the fact that the given outcome has actually occurred. Usually the transitions from a particular type of action will be mutually exclusive, so the Happens assumptions will be pairwise inconsistent. It is possible to specify action types that have transitions that are not mutually inconsistent.

3.4 Other Details

logical-form states and feature-structure states The B-SURE system does not work with the contents of states; the states are encapsulated. However, the user system may want to tell the difference between states represented by logical forms, feature structures, or other methods. In addition, it is useful for the B-SURE system to be able to print out state descriptions based on the type of contents of the state. The B-SURE system thus supports three types of states: logical-form states, feature-structure states, and other types. The logical-form states and the feature-structure states are each subtypes of the ordinary state type. The system automatically classifies the input data; the classification can also be done explicitly by the user using commands provided to support this facility.

State Action Worlds (SAWs) Situation instances are represented in the timelines by State Action Worlds. A SAW has a situation instance and a timeline history.

Performing Action Worlds (PAWs) A PAW represents an action instance.

3.5 Reset Commands

(reset-BSURE) Clears the B-SURE system out. Wipes out all known State Types, Situation Types, and Action Types. Resets the ATMS and clears out all nodes. Automatically initializes Node# 0 as the NOGOOD-NODE, and Environment# 0 as the Truth Environment.

(reset-SURE) Same as (reset-BSURE).

(reset-UNDA) Same as (reset-BSURE).

3.6 Commonly Used Type-Creation Commands

Type-Creation commands are called by the user to define types.

(nice-make-state-type data &optional (value NIL)) Defines and returns a state type. Checks to see whether the data is a logical form, a feature structure, or something else, and quietly defines the appropriate subtype. All state type-creation commands use a special state-type *uniquification* algorithm that checks to see whether the state type has been defined yet or not, by using a hash on `equal`. Returns the old state type if the name is redefined; does not change the value.

(make-situation-type name list-of-state-types) Defines and returns a situation type.

(make-MU-prob 0.5) Defines and returns a probabilistic influence.

(make-MU-p/q 0.0 0.8 0.2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0) Defines and returns an uncertain influence, defined by a distribution of the probability of the 21 probabilities from 0.00 to 1.00 at intervals of 0.05 apiece (i.e., 0.00, 0.05, 0.10, etc.). The entries default to zero and can thus be omitted if unnecessary (i.e., the example could have been defined by `(make-MU-p/q 0.0 0.8 0.2)`). The entries must sum to 1.0. For more information on second-order probabilities as used to represent uncertainties, see the author.

(make-transition-type influence resulting-situation-type &optional (name "") (URC nil)) Defines and returns a single transition type from an unspecified action to a specified outcome situation, with a corresponding associated specified probabilistic or uncertain influence.

(make-UNDA-type name documentation precondition-situation-type list-of-transition-types) Defines and returns a single transition type from an unspecified action to a specified outcome situation, with a corresponding associated specified probabilistic or uncertain influence.

3.7 Other Type-Creation Commands

Type-Creation commands are called by the user to define types.

(make-real-state-type data-or-state-type &optional (value NIL)) This routine is called when you are not sure whether what you are holding is data or is a state type, and you want to make sure it's a state type. Checks to see whether it's a state type or not. If so, the routine returns it. If not, the routine calles `nice-make-state-type` and returns the new type. Uses uniquification.

(make-nice-state-type data &optional (value NIL)) Same as `nice-make-state-type`. Uses uniquification.

(make-LF-state-type data &optional (value NIL)) Creates and returns a Logical-Form state. The data should be a logical form. This is used mostly for printing out. Uses uniquification.

(make-FS-state-type data &optional (value NIL)) Creates and returns a Feature-Structure state. The data should be a feature structure. This is used mostly for printing out. Uses uniquification.

(make-state-type data &optional (value NIL)) Creates and returns a general state. The data should be something that is defined and handled by the user. This is used mostly for printing out. Uses uniquification.

3.8 Commonly Used Instance-Creation Commands

These routines are used by the user to create instances of objects that already have been defined by type definitions.

(nice-make-state-instance state-type &optional (data 'UNBOUND) (value NIL)) Correctly makes a state instance of the right kind, given a state type. Quietly checks to see whether the state-type is a logical-form state, feature-structure state, or user-defined state, and then creates an instance of the corresponding type.

(make-nice-state-instance state-type &optional (data 'UNBOUND) (value NIL)) Same as `nice-make-state-instance`.

(make-situation-instance situation-type &optional (name situation-type-name) (value situation-type-value) (URC situation-type-URC)) Makes and returns a new situation instance of the given type. The name, value, and Uncertain Resource Consumption vector all default to those of the situation type. Automatically makes instances of all of the situation type's states.

3.9 Other Instance-Creation Commands

These Instance-Creation commands are sometimes called by the user to create instances of types explicitly.

Calling a make-state-instance routine with the wrong state type simply goes ahead and incorrectly allocates a state instance of the type mentioned in the name of the routine. The B-SURE data structures are perfectly fine; the user may have problems with the state data types, however.

(make-LF-state-instance state-type &optional (data 'UNBOUND) (value NIL)) Creates and returns a Logical-Form state. The data should be a logical form. This is used mostly for printing out.

(make-FS-state-instance state-type &optional (data 'UNBOUND) (value NIL)) Creates and returns a Feature-Structure state. The data should be a feature structure. This is used mostly for printing out.

(make-state-instance state-type &optional (data 'UNBOUND) (value NIL)) Creates and returns a general state. The data should be something that is defined and handled by the user. This is used mostly for printing out.

3.10 Commonly Used Action Commands

(start-situation situation-type &optional (value situation-type-value)) Creates and returns a State Action World (SAW) that represents an instance of the given situation type. Used for creating situation instances that start out action sequences, i.e. that are not derived from previous actions.

(do-UNDA-in-world UNDA-type SAW &optional (Agent NIL)) Hypothetically performs an instance of the given uncertain action type in the given State Action World (SAW) situation instance. Creates an instance of the action, and instances of the resulting outcome situations. Returns the action instance, in the form of a Performing Action World (PAW).

3.11 Data Pointer-Following Commands

These commands are used to get one piece of data from another.

(action-world-outcomes PAW-action-instance) Returns a list of the State Action Worlds (SAWs) representing the situation instances of the possible outcome situations for the given action instance.

(performing-world-outcomes PAW-action-instance) Same as `action-world-outcomes`. Returns a list of the State Action Worlds (SAWs) representing the situation instances of the possible outcome situations for the given action instance.

(state-world-actions SAW-for-situation-instance) Returns a list of the Performing Action Worlds (PAWs) representing the action instances of the possible subsequence (downstream) actions that have been entered for the given situation instance's State Action World (SAW).

3.12 History Mechanism Commands

These commands are used to manage the history mechanism that represents actual execution of the actions.

(Happening SAW) Describes the fact to the system that the given situation has started happening and is now currently going on. Makes the previous action world Past. (In the current system, the Past flag now stores a pointer to the next Happening action world, instead of simply a T/NIL flag.) Converts the assumption for the current world into a presumption, making the current

world Actual and all other mutually-exclusive worlds Inconsistent. Does not make the current given world Past. Actually works for both SAWs and PAWs. This function is actually the same as Choosing.

(Happened SAW) Describes the fact to the system that the given situation has happened already. Ensures that the previous action world is Past. (In the current system, the Past flag now stores a pointer to the next Happening action world, instead of simply a T/NIL flag.) Converts the assumption for the current world into a presumption, making the current world Actual and all other mutually-exclusive worlds Inconsistent. Makes the current given world Past by setting its flag to T, since the next world hasn't been Chosen yet. Actually works for both SAWs and PAWs. This function is actually the same as Chose.

(Choosing PAW) Describes the fact to the system that the given action has been Chosen by an agent, i.e. the action has started happening and is now currently going on. Makes the previous situation world Past. (In the current system, the Past flag now stores a pointer to the next chosen action world, instead of simply a T/NIL flag.) Converts the assumption for the current world into a presumption, making the current action world Actual and all other mutually-exclusive worlds Inconsistent. Does not make the current given world Past. Actually works for both SAWs and PAWs. This function is actually the same as Happening.

(Chose PAW) Describes the fact to the system that the given action was Chosen by the agent, has been performed, and has happened already. Ensures that the previous situation world is Past. (In the current system, the Past flag now stores a pointer to the next Chosen action world, instead of simply a T/NIL flag.) Converts the assumption for the current world into a presumption, making the current world Actual and all other mutually-exclusive worlds Inconsistent. Makes the current given action world Past by setting its flag to T, since the next world, i.e. the nondeterministic outcome, hasn't Happened yet. Actually works for both SAWs and PAWs. This function is actually the same as Happened.

3.13 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.

3.14 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.

3.15 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.

3.16 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.

3.17 User Access Commands

3.17.1 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.

3.17.2 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.

3.17.3 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.

3.18 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`.

3.19 Environment Commands

3.19.1 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.

3.19.2 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.

3.19.3 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.

3.20 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.

3.21 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.

3.22 Significant Variables

use-parallel-action-exclusions This variable is the flag for whether parallel actions coming out of the same situation are automatically made mutually exclusive or not. T = no parallel actions are allowed–when one action becomes true, all the rest become inconsistent. NIL = parallel actions allowed; if one action happens, the rest are not disabled. The default is T. Also see make-NONEX-UNDA-type.

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

ES This variable holds the Error 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.

3.23 System Flag Variables

watch-BSURE This flag makes the system print out a notification each time something changes in the BSURE system. Default is NIL.

use-parallel-action-exclusions This variable is the flag for whether parallel actions coming out of the same situation are automatically made mutually exclusive or not. T = no parallel actions are allowed–when one action becomes true, all the rest become inconsistent. NIL = parallel actions allowed; if one action happens, the rest are not disabled. The default is T. Also see make-NONEX-UNDA-type.

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 the B-SURE system do?

The B-SURE system is a representation system for states, actions, and situations in different possible worlds.

First, the user defines state types, and situation types composed of states. Then the user defines action types that transition between situations.

After that, the user defines a starting world (or two, or more), which is an instance of a situation.

Next, the user starts exploring what could happen, by defining hypothetical actions that could take place in any one world (if an agent chooses to execute that action). Each action will have one or more resulting worlds that will occur as outcomes. These worlds are still hypothetical, as of yet.

The user builds trees of possible choices of actions, possible outcomes from these actions, and reactions to these possible outcomes. There is a clear difference between abstract types and concrete instances.

Instances (and types) can be hypothetical, possible, or actual, also inconsistent or null. Hypothetical means that the user is simply considering the concepts and drawing hypothetical conclusions; there is no commitment yet. Possible means that the action or the world could actually happen if certain things come about. Actual means that the action (situation, state, etc.) is in fact now happening or has in fact happened. Inconsistent means that the action will never become possible nor actual. Null means that the system does not have a representation for that concept.

After the user has specified most of the hypothetical actions and worlds of interest, then the user can start asserting that some things actually happen. An agent actually chooses to perform an action in an actual world; in this case, the action turns from being merely hypothetical or possible into an actual occurrence.

However, the system has no way of knowing which outcome has actually occurred, until the user informs the system that a particular outcome happens. Before this, all outcomes are possible; after this, one outcome becomes actual, and the rest become inconsistent.[2] In this way, the user keeps track of which situations and states have actually occurred, which ones are still possible, and which ones are merely hypothetical.

In addition, the system can set up implications between states, so that if the conjunction of a set of certain antecedent states are all believed possible or actual, then an implied consequent state is automatically also believed possible or actual as well.

At any one point, a user can ask whether a state is true in any one possible world, or get a listing of all the states that are true in any one world.

The system does other things based on representing probabilities and uncertain probabilities of action outcome transitions.

[2]In some particular applications that can actually have more than one outcome occur, it is useful to be able to specify actions in which the other outcomes do not become inconsistent but stay possible. The system also supports these.

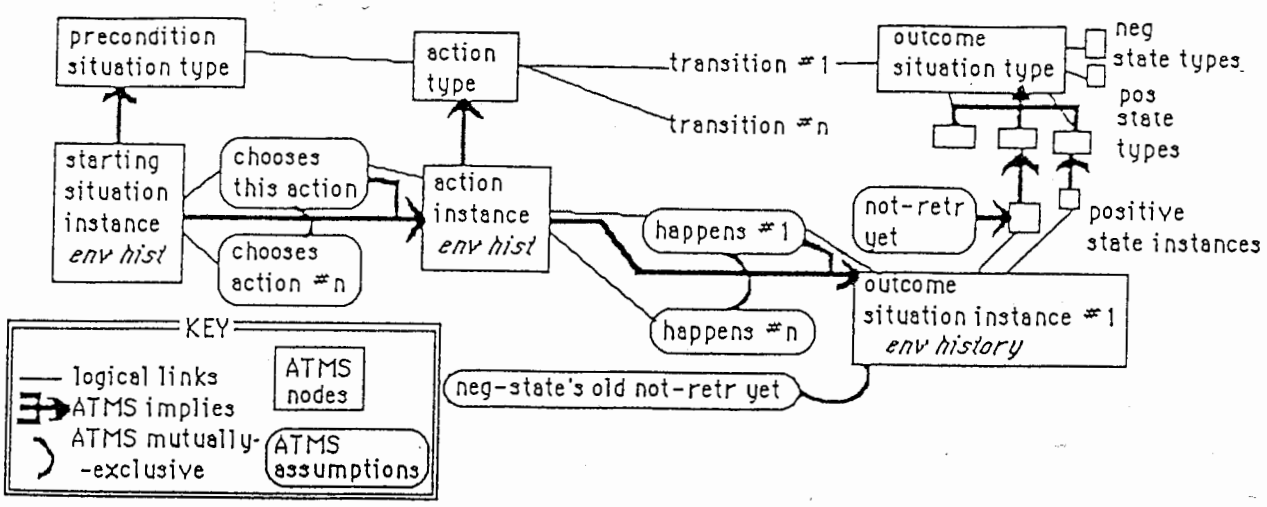

Figure 1: Structure for Representing Nondeterministic Actions

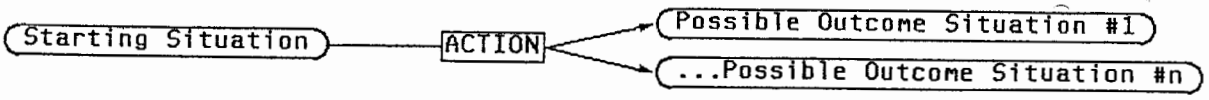

Figure 2: Compact Graphical Representation which Omits States and Types

Because possible worlds are complex, it is unfortunately necessary for the user to assert each hypothetical action to be explored in each significant world of interest *by hand*. This is an unavoidable design feature grounded on being able to represent nonmonotonic ordered actions in time, using current technology. In contrast, the NP system automatically asserts and explores all possible chains of actions that could be executed from a given possible world, but can only represent monotonic, partially-ordered actions in a basically timeless fashion. The extra expressive ability requires stronger control by the user.

5 Situation Theory

In [BP83], situations are divided into the categories *abstract* and *real*, and also into the categories *"states of affairs"* and *"courses of events"*. Abstract situations denote situations that are mental representations. All the situations discussed in this paper are "abstract situations". Real situations denote situations as they actually are in the real world. Since it basically never makes sense to talk about real situations in the computer, there is no need to supply these in a representation environment. "States of affairs" correspond to situations that are static, called simply *situations* in this paper. "Courses of events" correspond to situations that describe actions that are being executed, called *action events* or *actions* in this paper. Barwise and Perry also make use of "relations" defined over "individuals" and "space-time locations". This paper takes as primitive the expression of a relation, which will be termed a *state*. The user is free to mention individuals or space-time locations in state descriptions as desired. State descriptions may be represented using *logical forms*, *feature structures*, or *other* methods–since the contents of states are not used by SURE except for output, it does not matter. States, situations, and actions are assigned one of the belief values {`definitely believed true, possibly believed true, not believed true, believed not true, not believed`}, otherwise known as

{actual, possible, hypothetical, inconsistent, null}, corresponding to the amount of support offered by the system's underlying ATMS representation (see [Mye89a] for more information).

6 Intentional Action Theory

One model of intentions states that an intention is a *choice* to perform an action, plus a *commitment* to obtaining its desired outcome[CL87]. With deterministic action outcomes, there is no real need for *endeavoring* [Bra87], since once the action has been started, it is guaranteed to finish properly. Many planners in fact operate in this "fire and forget" mode. However, once it is acknowledged that action execution is in fact nondeterministic and can have undesirable outcomes, the need for endeavoring becomes clear. The planner must predict the likelihood of possible outcomes happening, and judge which action sequence offers the best chances. It must interactively maintain a history of past endeavors and results, and modify its future behavior based on current outcomes. Acting intentionally becomes significantly more interesting and realistic with the explicit representation of possible chains of nondeterministic actions.

7 Previous Efforts

DeKleer [dK86a] presents the first ATMS. Morris and Nado [MN86] present an ATMS that can represent nonmonotonic transitions, but do not handle probabilities, uncertainties, explicit situation types, state types, nor action events. The research of Allen (e.g. [AK83,All87]), who uses a predicate-calculus representation, offers some of the best multiple-worlds (deterministic) action representation in this field. Charniak and Goldman [CG89] use probabilities and Bayesian nets to represent the truth value of probabilistic statements and attack story understanding. Although nondeterministic-outcome actions are not represented, and Bayesian nets cannot support global inferencing with nonmonotonic actions, their work is important. Norvig and Wilensky [NW90] comment on problems of probabilistic statements. The most similar work is recent research by Rao and Georgeff (e.g. [RG91]), who use a modal logic instead of an ATMS to represent nondeterministic actions.

8 SURE Entities & Implementation

The underlying ATMS works with *nodes*, *assumptions*, and *implications* (justifications). See [dK86a].

A *state* consists of a proposition about the world. States are primitives. A *situation* is a set of positive and negative (withdrawn) states. An *action event* represents the state that execution of the action has started. States, situations, and actions have types and instances. See figure 1. (The abridged representation

of figure 1 is shown in figure 2.) Existance of an instance in a world always implies existance of its type. A *chooses* node is an assumption associated with an action instance that represents whether an agent chooses to execute that action or not. The chooses assumption together with the starting situation instance imply the action instance. Since an agent typically can only execute one action in a given situation, the situation's ensuing chooses assumptions are rendered mutually exclusive (pairwise "nogood"). Action types have precondition situation types. Action instances are instantiated from types by first verifying that the precondition situation type is believed true in that world. Action instances transition from a starting situation instance to one of a number of known nondeterministic outcome situation instances. Actions have *transitions*. A transition has an outcome situation and a probability or an uncertainty. An *uncertainty* is defined as a probability random variable of range $[0, 1]$ together with an associated second-order probability distribution. Uncertainties are initialized using maximum-entropy theory, and get updated as outcome observations are taken, to enable the system to learn possible probabilities. Uncertainties are used to represent *confidence* in values and to make decisions regarding information-gathering activity. The calculus of uncertainties is too complex to explore further here, and is not required for understanding the main capabilities of the representation; probabilities are sufficient. Transitions can be types or instances. A transition instance is defined as a *happens* assumption. An action instance, together with a happens assumption, imply the corresponding outcome situation instance. Typically only one outcome situation can occur from a given action instance, so the action's happens assumptions are made mutually exclusive. A situation type is implied by its state types. When an outcome situation instance is instantiated, all of its new positive states are instantiated and all of its old negative states are retracted. A positive nonpermanent state instance is implied by a *not-retracted-yet* assumption. The outcome situation instance remembers these. Situation and action instances store an explicit *environment history* of all added state, chooses, and happens assumptions that are currently believed true in that possible world's timeline. A negative state is retracted by making the situation instance and the state's "not-retracted-yet" assumption mutually inconsistent, and deleting the state's assumption from the outcome situation's environment history. A state type or instance or situation type's belief value in a particular world is found by testing that node against a situation instance's environment history. Situation types and instances can have values. Actions can have costs. The expected value of an action is determined by summing the transition probabilities times the expected values of the outcome situations, when known, and subtracting its cost. The expected value of a nonvalued situation instance is determined by maximizing the expected values of the possible subsequent actions, when known. In this manner, decision theory determines the course of action with the maximum expected value at any one situation, for a planning agent. This can be used to predict the probable next course of action of a planning agent by an observing agent performing plan recognition (actually, "decision recognition").

9 Representing Nondeterministic Actions

The main construct of SURE is an ATMS network structure for representing nondeterministic actions. This is shown in figure 1. The structure is instantiated from an action type. It starts with an ATMS node containing the State Action World that represents a situation instance. This has been previously justified by other starting assumptions. The State Action World contains an explicit environmental bit-vector that has a 1 bit for each of the assumptions leading up to this world, plus each of the nondeletion assumptions from this world's previous timeline that are still valid in this world. The user first searches for an appropriate action-type to perform by observing whether the action-type's precondition situation type is believed `possible` in the given situation instance. Having found a desirable action, the user then instructs the system to perform the given action-type in the given situation instance, with an optional Agent argument.

The system creates an assumption for the chooses structure, to represent the fact that the agent may or may not volitionally choose to perform the action.

10 Maintaining an Interactive History

One important advantage of the SURE system is that not only can it be used for hypothetical reasoning about future events, but the same structures can then be used as a history mechanism for interactively monitoring and representing the history of the actual events as they occur. A user system should start out in a known situation, which is presumed `actual`. Typically, the user system will use SURE to explore many different nondeterministic-action sequences and make decisions as to which actions are the best ones to perform. The system will then start executing the first action in the chosen sequence. At this point, the user system should instruct the SURE system to `presume` the chooses assumption associated with the chosen action being executed, which will change its truth value from "possibly believed true" to "definitely believed true". If the chooses node has already been made inconsistent with other chooses nodes (because the user-system or agent could only perform one action at a time), those other nodes are automatically rendered "believed not-true" at this point. The presumption of the chooses node renders the associated implied Action Event instantiation "definitely believed true" at this point, also. This represents the fact that the action has started and is currently being executed.

When the action finishes, it is necessary for the SURE system to realize which outcome occurred. This is typically performed by the system setting up a *recognition demon* that is attached to a separate state or situation type that, when true, reliably indicates that a given outcome has occurred. When the demon fires, it presumes the outcome's happens assumption. It is important to ensure that one and only one recognition demon fires. Alternatively, the user can control presuming the happens nodes directly. When a single happens assumption is presumed, it automatically renders its sibling happens assumptions "`definitely believed not-true`".

The combination of the happens node being presumed and the action event

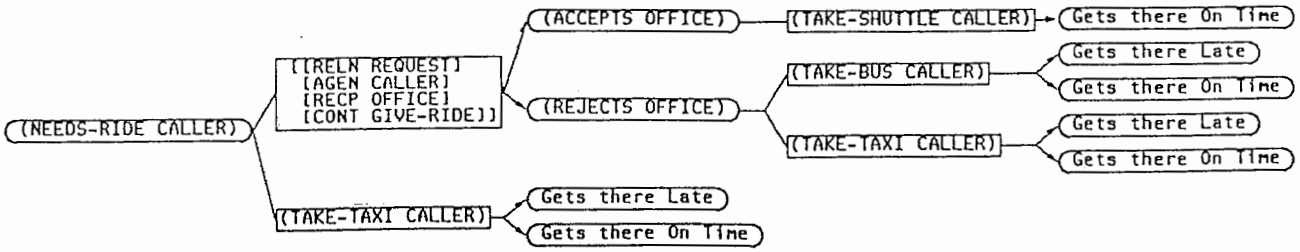

Figure 3: Modeling a Plan/Decision Inference Problem in Getting to a Conference On Time

node already being believed true renders the appropriate resulting situation instance believed true. Note that if any instance becomes true, so does its associated type node. as well.

At any one point in time, the states, situations, and action event instances that have happened in the world already are believed true; and the situations and events that have not happened yet but could happen are believed possible. In this way, the system maintains a timeline history of the situations and action events that have in fact occurred, while allowing hypothetical planning and exploration of possible future events in the same data structure.

It is not necessary for the system to maintain only a single timeline history. It is possible to maintain disjoint histories, to represent e.g. progress made by different processing agents, progress made in different domains, or progress made at different hierarchical levels of abstraction. It is possible to maintain forking (nondisjoint) histories if this makes sense, and the mutual exclusion options have been turned off (see Section 9).

10.1 Counterfactuals

The system maintains the structures of past possibilities that did not happen. Although these are `not believed true`, it is possible for the user to explore these structures and perform reasoning on what *could* have occurred had certain actions been chosen or certain nondeterministic outcomes happened, by supplying an extra *counterfactual* assumption to justify the desired action or situation instance. It is even possible to add to these structures, if necessary. This can be used to explicitly represent newly-received past counterfactual information (e.g., "If you had applied for the conference last June, the cost would have been 35,000 yen") and the associated reasoning derived from such assertions. Such reasoning has traditionally been very difficult to represent, because of the negative truth values.

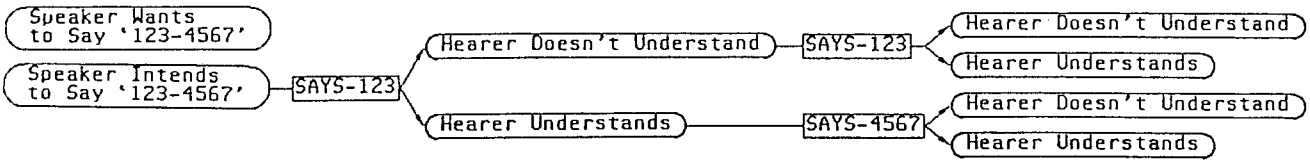

Figure 4: Modeling an Intention to Communicate a Telephone Number Correctly

11 Decision Inference Example

A researcher is calling a conference office from the train station and wants to get to the conference on time. He has a choice between asking for taxi directions, or requesting the office to send a shuttle-bus out directly to give him a ride. The shuttle will take him directly to the conference on time. If he requests and the office turns him down, he has a choice between taking a taxi, and taking the regular bus. These cost different amounts of money and have different chances of getting to the conference on time. See figure 3. The plan inference system must predict which paths of information he will explore, i.e. what he will say next; and then which decisions he will make for his actions. This is done using "decision inference", by understanding which action trees offer the best expected value based on the value and chances of outcomes. Note that the shuttle-bus, the taxi, and the regular bus will all three allow the researcher to possibly obtain his desired goal, but there are definite preferences. The system should not remain uncommitted. See [Mye91] for more details.

12 Intentional Communication Example

A recent analysis of 12 actual interpreted telephone conversations revealed that 31% of the utterances were spent in requests for confirmation and repetitions of information such as telephone numbers, name spellings, and addresses, that were not completely understood the first time [OCP90]. This means that the traditional plan-recognition model of assuming that the hearer automatically understands the semantic content of the speaker's utterance is fallacious. The speaker, and the system too, must consider the case in which the hearer does not understand an utterance. Since the speaker wants and intends to communicate specific information[3], the speaker will endeavor to ensure that the information is communicated, by repeating an utterance when it is not understood. Thus, speaking an utterance is a nondeterministic action; it is unclear whether the hearer will understand or not. Intentional utterance acts are therefore modeled as nondeterministic-outcome actions by SURE. Different courses of the conversation can be represented depending upon the outcomes of the utterance acts. See Figure 4.

[3]Note that people do not always decide to intend to endeavor to do everything that they want. Intending is quite different from wanting.

13 A Problem with B-SURE

There is a mistake that the naive user can make that should be watched out for. The symptoms are that the user builds a possible network of actions, starts presuming actions that have Happened, and all of a sudden the *entire* network turns hypothetical–even those actions that had Actually happened in the past.

The reason why this happens is that the system automatically assumes that only one action can actually be performed in one situation at a time. All of the rest are mutually inconsistent, unless the system flag `use-parallel-action-exclusions` is set to NIL at the time the network is *built*, or unless the function `make-NONEX-UNDA-type` is used to specify nonexclusive UNDA action types. When the user asserts that an action definitely happens, and then asserts that a different (mutually-exclusive) action definitely happens in the *same situation*, the ATMS underlying the history mechanism breaks and *all* nodes turn hypothetical. It is a conceptual error to attempt to specify that two different things actually (not possibly) happened in one history line at the same time, without making them nonexclusive.

14 Summary of Conceptual User Operation of the B-SURE system

- The system is initialized with a set of state types.
- The system is initialized with a set of situation types using the state types.
- The system is initialized with a set of influences.
- The system is initialized with a set of transition types using the influences and the situation types.
- The system is initialized with a set of action types using the situation types for preconditions and the transition types for results.
- The user starts the system by declaring that some of the situation types have been instantiated into situation instances.
- The user performs planning and searching by instantiating possible instances of actions in specific situation instances.

15 Conclusion

B-SURE is a system that represents states, situations, and nondeterministic actions in timeline histories of sequential nonmonotonic possible worlds. This capability is necessary for supporting plan recognition or decision recognition with nondeterministic actions, scheduling nondeterministic processes, creating intentional agents, and understanding the actions of people operating in the real world.

A Notes on Version History

These notes are presented for implementers who may have to change the system. It is *not* necessary for users to read or understand these notes.

Version 1.0

```
This version used ATMS action worlds, but did not use SAWs nor PAWs.
```

Version 1.1

```
  Moved to atms-action-worlds1.
  \#+'s deleted.
  The current possible-action-worlds package is still fumbling deletion.
It goes through most of the motions, but does not acknowledge that
deleted nodes are no longer in the indicated world.
The problem lies with (in-action-world-p).
Solution:  ''Form an environment using (create-env) or (add-to-env)
  of ALL PREVIOUS worlds plus ALL associated undelete sets.
     Stick this in the world node.
Test a node against a world by pulling the env out of the world node,
and doing an (in-context-p) to test."
```

Version 2.0

```
 Choice, Happens, Performing-world, State-world, etc.
Deletes no longer supported--situations do not yet accept negative states.
Transitions use situations. Super major reorganizations.
```

Version 2.1

```
  delete overhaul. Modified:  del-node-from-world;
do-action-in-world, do-raw-action-in-world, add-node-to-world
        uses new add-assums-to-env.
  Took the implications off the situation instances, put them on SAWs.
  Cleaned up a bunch of problems.  Make-CHOOSES got turned into
    do-UNDA-in-world; there's really no need for
    an independent make-Chooses.  All routines are much more logical now.
    Make-STATE-WORLD sideways-implies its new states; the Situation Instance
    now does not get its own node, and is a dinosaur kept for documentation.
    Make-HAPPENS calls Make-STATE-WORLD and fills in
    the Happens Transition.  The old do-action-into-world has been
    modified extensively, is now attach-new-world-to-world; it takes care of
```

hooking up ALL pointers. TransAssum data type allows this.
Probable problem
with making Performing Action World in a depth-first manner: The Env magic probably needs to be propagated before the new SAW deletes its states. Made Fill-In-Performing-World to take care of this problem; Happens instantiations are delayed until the Performing World gets its proper Env magic.

attach-new-world-to-world is called for both Chooses and Happens instantiations. It assumes that only one Happens can occur and that they are pairwise exclusive (true, as long as the outcome events are mutually exclusive, which they will be in all of the foreseeable future); and also that only one Chooses can occur and that the Chooses action options are mutually exclusive. This last is equivalent to assuming that the agent can only perform one action at a time. In order to correct this, make a copy of attach-new-world-to-world called attach-new-world-to-world-without-nogood, and take out the pairwise nogood. Other methods will be nastier because node can't be pushed before nogooding.

World-p not picking anything up because assum history not yet stuffed on start-situation. Using quick-world-p.
Problem: the Env magic must be stuffed in an Action World before nodes can be added or deleted. Had to also make a Fill-In-State-World to take care of this problem.

A.1 Version 2.2

Hacked up use-parallel-action-exclusions, use-parallel-outcome-exclusions.
Currently this is a system-wide flag;
the right way to do this would be to make a new kind of flavor for parallel action-outcomes, and maybe
a new kind of flavor for parallel-izable actions.
But let's get it to work right first.
Modified routines: do-action-in-world
(not used?!); attach-new-world-to-world; make-HAPPENS; do-UNDA-in-world.
Error msgs in del-node-list-from-world
using del-node-to-world; I corrected them.
State types and state instances; must be
able to instantiate state instances after working with them.
Almost everything with a state in it has been modified.
start-world, do-action-in-world, do-raw-action-in-world
deleted. Can be found in atms-action-worlds1.
Names on transition types. \$its use Sit's name.
NONEX-UNDA-TYPEs. Use-parallel-output-exclusions cut out.
Hacked make-HAPPENS.

Agents on Chooses.

make-chooses taken out. Do-UNDA-in-world now returns the action-world,
not the chooses assumption.
Some loops changed to dolists. reset-SURE; reset-BSURE.
Turned watch-UNDA into watch-SURE.
Action-world-actions returning a list of NODES,
not ACTIONS. Fix this.

FEATURE. Symptoms: After presuming a lot of nodes in the network,
all of a sudden everything turns Hypothetical.
Reason: Oh no, you Presumed two outcome Happens from the
same Action, and the mutually-exc-disable
flag was not turned off. You have just proved NOGOOD from
TRUE, which blows away your logic.
SOLUTION: Don't use mutually exclusive outcomes on that action.

A.2 Version 2.3

Past markers on the Action Worlds. Happening, Happened, Choosing,
Chose functions.
watch-SURE defaults to NIL for the production version.
tr-assums for next worlds on Action World structure.
Actions turned to tr-assums; new Actions corrected.
Modified: choosez-nodes, happenz-nodes,
attach-new-world-to-world.
Well, I finally found a use for situation instances
vs. SAWs. Use the situation instances to
store the values of the situation itself.
Use the SAWs to store the value of the situation
plus all of its future potentials etc.
Required when incremental value happens. Could need changing.
Null precondition situations allowed in Make.
Fill-in-Performing-world now operates on
reverse of list, to get order to come out right.
Currently actions coming out of a situation
node are reverse-temporally-ordered because of a Push.
There is no good way to get around this yet.
Local and Total URC. Total only represents
DOWNSTREAM Total + local. What about upstream totals?
Total URC on Happens was macro'd to Type, don't do that!
Converted stuff to Actual, Feasible, Potential.
Happened, Happening now set Past to point to the next
one, not just T.
Hacked in value initialization on make-situation-type.
make-state-world uses situation-instance value.
watch-SURE to *watch-BSURE*. BSURE-FS-p,
BSURE-LF-p. get-State-type.

B Notes on Implementation and Theory of the System

B.1 Notes on States

Start with the States. There is a need to have state types and state instances, even though a state is more like an adjective than a noun. States represent abstract State Types and concrete State Instances. In real-world planning applications, often the data for the actual State Instance will be unbound until the action instance is actually executed and the results observed. For this reason, State Instances are structures with data slots. Since this system only worried about the -identity- of the state and its values, uncertainties, etc., the actual -contents- of the state are left open for the user to change. States can be represented using Logical Forms, Feature Structures, or anything else you wish–since states are encapsulated, it doesn't matter what the data is. Currently states can only be positive–in other words, existence=true, nonexistence=false. However, it is of course possible to define negative states by including a NOT in the encapsulated definition, and certain commands retract states (rendering them not believed in that world). A type has a list of instances; an instance points to its type. Currently both State Types and Instances have ATMS-nodes.

B.2 Notes on Situations

A Situation is defined as a set union of states. Situations comprise Situation-Types, which are patterns corresponding to the abstract conceptual occurrence of the situation, and Situation-Instances, which are concrete, instantiated occurrences of the situation (and use State Instances). A type has a list of instances; an instance points to its type. Situation-Types have a set of State Types; Situation Instances, of State Instances. A Situation-Instance may be in the future, or it may be only "possible". Unclear whether it makes sense to talk about "hypothetical" instances or not.

B.3 Notes on UNDAs

The system concerns itself with Uncertain Non-Deterministic Actions (UNDAs). UNDAs have types and (virtual) instances; the instances are actually a collection of other structures, mostly tied together by a Chooses node. An UNDA-Type points to a precondition Situation-Type and a number of Transition-Types. A Transition-Type points to a resulting Situation-Type.

B.4 Notes on Transitions

Transition-Types have an uncertain Influence, and also an URC. What would be a transition-instance is called a Happens structure. All Happenses are thus concrete.

A Happens points to its Transition-Type to determine its influence chances and URC. A Happens points from its previous world to its resulting world.

B.5 Notes on Chooses nodes

An UNDA instance is basically represented by a Chooses node. Chooses means "chooses to do the action", as in "plumps"; no decision-theoretic optimal choice is implied, and many action instantiations from a single situation will each have their own Chooses node. A Chooses points from its previous world to its resulting world. Chooses nodes have optional Agent slots.

B.6 Notes on Action Worlds

Actions in the UNDA package are represented by multiple possible action worlds. Worlds are explicit node structures, that support all this mess. The two important kinds of worlds are the State Action World (SAW), and the Performing Action World (PAW). A State A.W. represents the fact that a Situation-Instance has transitioned into being. It points to its Situation-Instance. It also has a best-action slot, to support action decision-theoretic choices. A Performing A.W. represents the fact that an action, (natural or volitional), is being performed. It points to an UNDA-Type. Both SAWs and PAWs are supported by custom low-level Action World structures, which should be ignored by the user. Both SAWs and PAWs have corresponding atms-nodes. Situation-instances do not need atms-nodes. SAWs "sideways imply" (with a Non-Deleted assumption) all of their resulting added nodes. SAWs are currently created from a template of a Situation Instance, which provides the State Instances.

B.7 Notes on Implications

A State Instance implies its State Type. The occurrence of all of the State Types implies a Situation-Type. A Situation-Instance used to imply each of its comprising State Instances (really Types), but now this is not true–they are "sideways". A State Action World plus a Non-Deleted assumption sideways implies each of its added State-Instance nodes.

B.8 Notes on URC

The Uncertain Resource Consumption (URC) package is a note-taking package that currently only sits on top of UNDA and goes along for the ride. A URC is actually a vector of uncertain consumptions, notably Time, Money, Fuel, and Prestige.

B.9 Notes on Deletion Theory

Deletes currently work by creating a node called (UNDELETED #<Atms-node #FOO>). The deletion routine then searches for the node with this name. This could be a problem, because it does not seem to support adding the node in multiple places in the action net, nor deleting the node and then adding it again. Or does it? Answer: There is no problem with adding the node at different places in the tree. There is no problem with deleting the node. There is a small problem with adding, deleting, and adding the node again, because currently the same (old) assumption is used to justify the node. However, this assumption has been made mutually incompatible with the history, by the delete.

C Example Listing

This listing illustrates part of a program to set up and monitor nondeterministic systems as they execute.

```
(reset-SURE)

(setq speak-state   (make-nice-state-type '(Sound Input)))
(setq hear-state   (make-nice-state-type '(SpRec Output)))
(setq morph-state  (make-nice-state-type '(Morph Analysis Output)))
(setq patt-state  (make-nice-state-type '(Pattern Matcher Output)))
(setq temp-state  (make-nice-state-type '(Template Example Distance)))

(setq start-sit    (make-situation-type "Speech In" (list speak-state)))
(setq hear-sit     (make-situation-type "SpRec Out" (list hear-state)))
(setq morph-sit     (make-situation-type "Morphs" (list morph-state)))
(setq patt-sit     (make-situation-type "Match" (list patt-state)))
(setq temp-sit    (make-situation-type "Dist" (list temp-state)))


(setq influence-1 (make-MU-prob 1.0))
(setq influence-01 (make-MU-prob 0.1))
(setq influence-025 (make-MU-prob 0.25))
(setq influence-05 (make-MU-prob 0.5))
(setq influence-03 (make-MU-prob (/ 1 3)))

(setq trans-05-hear   (make-transition-type influence-05 hear-sit))
(setq trans-025-morph   (make-transition-type influence-01 morph-sit))
(setq trans-05-morph   (make-transition-type influence-05 morph-sit))
(setq trans-025-patt   (make-transition-type influence-01 patt-sit))
(setq trans-03-patt   (make-transition-type influence-03 patt-sit))
(setq trans-05-patt   (make-transition-type influence-05 patt-sit))
(setq trans-025-temp   (make-transition-type influence-01 temp-sit))
(setq trans-03-temp   (make-transition-type influence-03 temp-sit))
(setq trans-05-temp   (make-transition-type influence-05 temp-sit))


(setq unda-hear (make-UNDA-type 'Speech-Rec "Hearing Speech Recognition"
start-sit
(list trans-05-hear
      trans-05-hear
      )))


(setq unda-morph (make-NONEX-UNDA-type 'Morph-Analys "Morphological Analysis"
```

```
hear-sit
(list trans-05-morph
      trans-05-morph
      )))

(setq unda-patt (make-NONEX-UNDA-type 'Patt-Match "Pattern Matching"
morph-sit
(list trans-05-patt
      trans-05-patt
      )))

(setq unda-temp (make-NONEX-UNDA-type 'Temp-Match "Template Pattern Matching"
patt-sit
(list trans-03-temp
      trans-03-temp
      trans-03-temp
      )))



(setq start-SAW (start-situation start-sit))

(setq list-of-hears
      (action-world-outcomes
(do-UNDA-in-world unda-hear start-SAW)))

  (loop for hear-SAW in list-of-hears do
    (setq list-of-morphs
  (action-world-outcomes
    (do-UNDA-in-world unda-morph hear-SAW)))


    (loop for morph-SAW in list-of-morphs do
      (setq list-of-patts
    (action-world-outcomes
      (do-UNDA-in-world unda-patt morph-SAW)))

      (loop for patt-SAW in list-of-patts do
(setq list-of-temps
      (action-world-outcomes
(do-UNDA-in-world unda-temp patt-SAW)))
      )

    )
  )
```

```
(format T "Presuming ~A.~%"
                    (happens-atms-node (state-world-happens start-SAW) ))
(Happened start-SAW)
;;(presume-this-node (happens-atms-node (state-world-happens start-SAW) ))

(setq SpRec-Action (car (state-world-actions start-SAW)))
(Happened SpRec-Action)

(setq SpRecOut-SAW (car (performing-world-outcomes SpRec-Action)))
(Happened SpRecOut-SAW)

(setq Morph-Action (car (state-world-actions SpRecOut-SAW)))
(Happened Morph-Action)

(setq MorphAnal-SAW (car (performing-world-outcomes Morph-Action)))
(Happened MorphAnal-SAW)

(setq Patt-Action (car (state-world-actions MorphAnal-SAW)))
(Happened Patt-Action)

(setq Match-SAW (car (performing-world-outcomes Patt-Action)))
(Happened Match-SAW)

(setq Temp-Action (car (state-world-actions Match-SAW)))
(Happened Temp-Action)

(setq dist-hap1-SAW (first (performing-world-outcomes Temp-Action)))
(Happened dist-hap1-SAW)

(setq dist-hap2-SAW (second (performing-world-outcomes Temp-Action)))
(Happened dist-hap2-SAW)
```

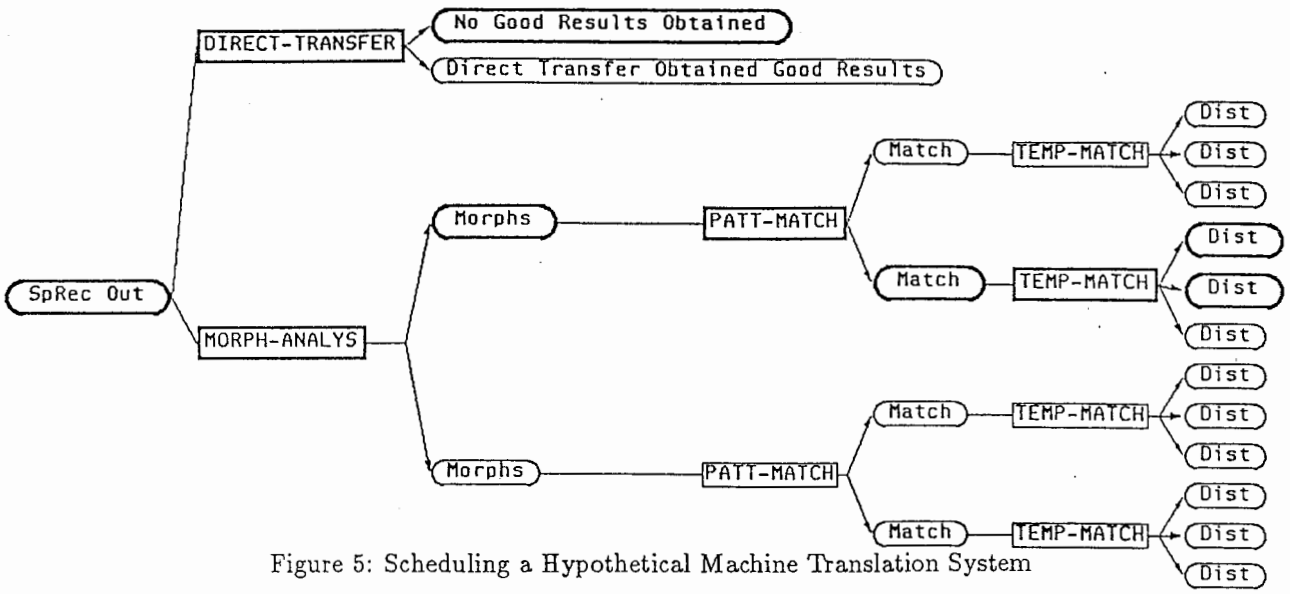

Figure 5: Scheduling a Hypothetical Machine Translation System

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 ATMS system out.

(reset-BSURE) Clears the B-SURE system out. Wipes out all known State Types, Situation Types, and Action Types. Resets the ATMS and clears out all nodes. Automatically initializes Node# 0 as the NOGOOD-NODE, and Environment# 0 as the Truth Environment.

(reset-SURE) Same as (reset-BSURE).

(reset-UNDA) Same as (reset-BSURE).

(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

[AK83] James F. Allen and Johannes A. Koomen. Planning using a temporal world model. In *IJCAI'83*, pages 741–747, Karlsruhe, West Germany, 1983.

[All87] James Allen. *Natural Language Understanding.* Benjamin/Cummings Pub. Co., Menlo Park, CA, 1987.

[Bar89] Jon Barwise. *The Situation in Logic.* Center for the Study of Language and Information (CSLI), Stanford, CA., 1989.

[BL85] Ronald J. Brachman and Hector J. Levesque. *Readings in Knowledge Representation.* Morgan Kaufmann, Los Altos, CA, 1985.

[BP83] Jon Barwise and John Perry. *Situations and Attitudes.* The MIT Press, Cambridge, Mass., 1983.

[Bra87] Michael E. Bratman. *Intention, Plans, and Practical Reason.* Harvard Univ. Press, Cambridge, MA, 1987.

[CG89] Eugene Charniak and Robert Goldman. A semantics for probabilistic quantifier-free first-order languages, with particular application to story understanding. In *IJCAI'89*, pages 1074–1079, Detroit, MI, 1989.

[CL87] Philip R. Cohen and Hector J. Levesque. Intention = choice + commitment. In *AAAI'87*, pages 410–415, Seattle, WA, 1987.

[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*, 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.

[MN86] Paul H. Morris and Robert A. Nado. Representing actions with an assumption-based truth maintenance system. In *AAAI'86*, Philadelphia, PA, 1986.

[Mye89a] John K. Myers. An assumption-based plan inference system for conversation understanding. In *WGNL Meeting of the IPSJ*, pages 73–80, Okinawa, Japan, June 1989.

[Mye89b] John K. Myers. The atms manual (version 1.1). Technical Report TR-1-0074, ATR Interpreting Telephony Research Laboratories, Kyoto, Japan, February 1989.

[Mye91] John K. Myers. Plan inference with probabilistic-outcome actions. In *Conf. Proc. Information Processing Society of Japan*, volume 3, pages 168–169, Tokyo, Japan, March 1991.

[NW90] Peter Norvig and Robert Wilensky. Abduction models for semantic interpretation. In *COLING-90*, volume 3, pages 225–230, Helsinki, Finland, August 1990.

[OCP90] Sharon L. Oviatt, Philip R. Cohen, and Ann Podlozny. Spoken language in interpreted telephone dialogues. Technical Report AIC-496, SRI International, Menlo Park, CA, 1990.

[RG91] Anad S. Rao and Michael P. Georgeff. Asymmetry thesis and side-effect problems in linear-time and branching-time intention logics. In *Proceedings of IJCAI-91*, Sydney, Australia, 1991.

[WN88] John R. Walters and Norman R. Nielsen. *Crafting Knowledge-Based Systems: Expert Systems Made (Easy) Realistic.* John Wiley & Sons, New York, NY, 1988. pp. 253-284.