TR-I-0367

The NP Manual
NPマニュアル
Version 3.1

John K. Myers

March 16, 1993

Abstract

This manual presents user documentation for the ATR Interpreting Telephony Research Laboratories Natural-language Plan-inference system, NP version 3.1. NP is an assumption-based system that uses Nadine-style feature structures as its basic data representation, for plan schemata input, utterance input, and output. The system thus will be able to take input directly from the semantic parser, and send output directly to the transfer and/or generation modules

THE NP MANUAL

Version 3.1

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 Natural-language Plan-inference system, NP version 3.1. NP is an assumption-based system that uses Nadine-style feature structures as its basic data representation, for plan schemata input, utterance input, and output. The system thus will be able to take input directly from the semantic parser, and send output directly to the transfer and/or generation modules.

The NP system actually contains two other systems, the NFL Natural-language Fact/Rule Language inference engine, which also uses feature structures, and the ATMS Assumption-based Truth Maintenance System. These systems can be used by themselves, for tasks other than plan inference. They are described in here as well.

NP is a Natural-language Plan-inference system that is based on assumptions and uses feature structures as its input and output. Plan schemata with preconditions, decompositions, and effects are represented by feature structures, which can be taken directly from the output of a semantic parser. The system's result is a network of believed assertions in an ATMS knowledge base, representing the inferred plans. This network can drive user-supplied processing demons or can be used to answer language-system queries. The plan-inference component is implemented using models of recognition, prediction, and inference, and a feature-structure-based inference engine called NFL (Natural-language Fact/Rule Language). NFL is implemented using a nonmonotonic rewriting system for pattern-matching, and a Rete algorithm for control and conjunction testing. Its output is assertions in the ATMS data-base. The ATMS allows pre-instantiation of hypothetically known assertions and implications. When these match observed or derived assertions, significant time is saved. The ATMS also permits simultaneous consideration of multiple possible inputs or inferred plan outputs, which will be important for disambiguation. A dialog understanding example illustrating how plans are inferred from multiple alternative inputs is presented.

Acknowledgment

This research was supported by ATR Interpreting Telephony Research Laboratories. I would like to express my gratitude to Dr. Akira Kurematsu, President of the Interpreting Telephony Research Laboratories, for the invitation to come to ATR, for providing the support that enabled this research to be done, and for the interest shown in this work. Thanks are also due to Mr. Hitoshi Iida, who provided helpful suggestions as to the technical direction of this work. Finally, I would like to acknowledge the friendliness and the helpfulness of the rest of the people in the Natural Language Understanding Department.

Contents

List of Figures

1 Working with the NP system

NP is installed as a System on the Symbolics network, in directory `LM01:>NP>*`. This directory is designed to contain the complete source to the code of the system itself, including the `system` lisp extension file, a copy of `atms5`, and a copy of `node-graphics`. The conversation data to be processed is kept in a subdirectory, `LM01:>NP>example-conversations>*`.

The NP system is invoked on the Lisp Machine by using the command `Load System NP`.

The plan schemata for the system are kept in various files starting with `LM01:>myers>np1-plan-XXX`.

Running the NP system is done by typing (`NP` *input-conversation-file-code training-conversation-file-code list-of-plan-actions-file-codes*). The input-conversation file and the training-conversation file are determined based on conversation codes, which are presented here (from function `demo-file` in file `np2-prog`.

```
  (case code
    (0 "lm01:>myers>convn>conv0-ex")
    (A "lm01:>myers>con>flail-conv-Ae")
    (B "lm01:>myers>con>flail-conv-Be")

;    (1 "lm01:>myers>convn>conv1-test")
    (1 "lm01:>myers>convn>conv1-ex")
    (1s "lm01:>myers>convn>conv1-ex-short")
    ((1r r1) "lm01:>myers>convn>rough-1")
    ((1-4L mixed) "lm01:>myers>convn>conv1-4L-mixed")
    (2 "lm01:>myers>convn>conv2-ex")
    (3 "lm01:>myers>convn>conv3-ex")
    (4 "lm01:>myers>convn>conv4-ex")
    (5 "lm01:>myers>convn>conv5-ex")

    ((d0 0d c0d cd0) "lm01:>myers>convn>c-demo-0")
    ((d1 1d c1d cd1) "lm01:>myers>convn>c-demo-1")
    ((d2 2d c2d cd2) "lm01:>myers>convn>c-demo-2")
    ((d3 3d c3d cd3) "lm01:>myers>convn>c-demo-3")
    ((d4 4d c4d cd4) "lm01:>myers>convn>c-demo-4")
    ((d5 5d c5d cd5) "lm01:>myers>convn>c-demo-5")
    ((d6 6d c6d cd6) "lm01:>myers>convn>c-demo-6")
    ((d7 7d c7d cd7) "lm01:>myers>convn>c-demo-7")

    ((cAs cAst) "lm01:>myers>convn>cA-short-test")
    ((cBs cBst) "lm01:>myers>convn>cB-short-test")
    ((c1s c1st) "lm01:>myers>convn>c1-short-test")
    ((c2s c2st) "lm01:>myers>convn>c2-short-test")
    ((c3s c3st) "lm01:>myers>convn>c3-short-test")
    ((c4s c4st) "lm01:>myers>convn>c4-short-test")
    ((c5s c5st) "lm01:>myers>convn>c5-short-test")
```

```
    ((c1sp c1p c1stp) "lm01:>myers>convn>c1-short-test-preinit")

    (6 "lm01:>myers>con>flail-conv-6e")
    (7 "lm01:>myers>con>flail-conv-7e")
    (8 "lm01:>myers>con>flail-conv-8e")
    (9 "lm01:>myers>con>flail-conv-9e")
    (10 "lm01:>myers>con>flail-conv-10e")
    (-1 "lm01:>myers>convn>conv0-ex")
    ((4L1 4L) "lm01:>myers>convn>4L-conv1")
    ((4L2 FS-LF)  "lm01:>myers>convn>conv1-ex")
    (train  "lm01:>myers>cs-conv2")
    (test  "lm01:>myers>convn>small-test")
    (same  same-file-name)
    ((nil) nil)
    (  T (progn
  (format T "~&Using unfamiliar conv file ~A.~%" code )
  code))
)
```

The plan action schemata files use different codes, determined in function `plan-file` in file `np2-prog`.

```
  (loop for mycode in code
collecting
  (case mycode
    (1A "lm01:>myers>np1-plans-1a-TRNSFR-rules")
    ((4L1 4L) "lm01:>myers>np1-plans-4L")
    ((4L2 FS-LF) "lm01:>myers>np1-plans-4L-FS-LF")
    (0 "lm01:>myers>np1-plans0-test")
    (1 "lm01:>myers>np1-plans1-shortAns")
    (10 "lm01:>myers>np1-plans10")
    (2 "lm01:>myers>np1-plans2-informif")
    (3 "lm01:>myers>np1-plans3-hello-bye")
    (4 "lm01:>myers>np1-plans4-CAN")
    (5 "lm01:>myers>np1-plans5-asking-knowing")
    (6 "lm01:>myers>np1-plans6-domain")
    (7 "lm01:>myers>np1-plans7-idioms")
    (8 "lm01:>myers>np1-plans8-cmnSns-time")


    ((d0 0d c0d cd0) "lm01:>myers>np1-plan-c0-short")
    ((d1 1d c1d cd1) "lm01:>myers>np1-plan-c1-short")


    ((cAs cAst) "lm01:>myers>np1-plan-cA-short")
    ((cAi cAio cAo) "lm01:>myers>convn>cA-inter-old-test")
```

```
    ((cBs cBst) "lm01:>myers>np1-plan-cB-short")
    ((cBi cBio cBo) "lm01:>myers>convn>cB-inter-old-test")
    ((c1 1c 1r r1) "lm01:>myers>np1-plan-c1-rough")
    ((c1s c1st) "lm01:>myers>np1-plan-c1-short")
    ((c1i c1io c1o) "lm01:>myers>convn>c1-inter-old-test")
    ((c1rit c1it 1rit r1it c1irt) "lm01:>myers>convn>c1-inter-rough-test")
    ((c1ri 1ri r1i c1ir) "lm01:>myers>convn>c1-inter-rough")
    ((c2s c2st) "lm01:>myers>np1-plan-c2-short")
    ((c2i c2io c2o) "lm01:>myers>convn>c2-inter-old-test")
    ((c3s c3st) "lm01:>myers>np1-plan-c3-short")
    ((c3i c3io c3o) "lm01:>myers>convn>c3-inter-old-test")
    ((c4s c4st) "lm01:>myers>np1-plan-c4-short")
    ((c4i c4io c4o) "lm01:>myers>convn>c4-inter-old-test")
    ((c5s c5st) "lm01:>myers>np1-plan-c5-short")
    ((c5i c5io c5o) "lm01:>myers>convn>c5-inter-old-test")

    (train "lm01:>myers>cs-plans2")
    (test "lm01:>myers>convn>inter-small-test")
    (full "lm01:>myers>cs-plans")
    ((normal default can)
        "lm01:>myers>cs-plans-can")

    (  T (progn
  (format T "~&Using unfamiliar plan file ~A.~%" mycode )
  mycode))
    )

into answerlist
finally (return answerlist)
 )
```

The results are represented in the system's ATMS data-base. Finally, if you want the results printed out in a graph, you should call `draw-graph (reverse *atms-nodes*))`.

2 Introduction

This manual describes the ATR Interpreting Telephony Research Laboratories' NP plan inference system, version 3.1. NP stands for the *Natural*-language *Plan*-inference system. The NP system accepts descriptions of general actions, in the form of plan schemata with variables. Next, the system is preinitialized with common-sense knowledge assertions and hypothetical knowledge. Finally, the system is given parsed feature-structure utterances from a conversation. The system instantiates a hierarchy of plan schemata representing the recognized, predicted, and inferred plans abstracted from the conversation.

The NP system actually consists of three systems, or layers, taken together. Because the user can interface with NP at any one of these layers, it is important to understand

all of them. The *NP plan inference* layer is a plan inference engine. It works with feature structures as its basic input format, both for data and plans to be recognized. This is the level that the user will probably use most often. The NP plan inference layer uses the *NFL* layer. NFL stands for the *N*atural-language *F*act/Rule *L*anguage. It is an inference engine that works with feature structures as its input. NFL uses the *ATMS* layer to record its inferences. ATMS stands for *A*ssumption-based *T*ruth *M*aintenance *S*ystem[dK86a]. The ATMS is a data-base that is able to represent and store concepts (*atms-nodes*) and constraints between these concepts (*implications*) that occur in different possible situations at the same time, known as *multiple possible worlds*. Worlds are set up by "assuming" a concept–if the assumed concept is believed, this contributes to forming one possible world, whereas if the assumption is disregarded (not believed), this forms a different possible world.

Dialog understanding is important for machine translation. In order to disambiguate possible translations, it is necessary to represent the perceived beliefs and goals of the dialog participants. However, beliefs cannot be directly observed; thus, when understanding a dialog, it is necessary for a system to make assumptions. The dialog participants must also make assumptions about domain operations and communication, that must be modeled by the system. However, when moving beyond the understanding of simple dialogs to using input from a real corpus, these assumptions can be mistaken. The understanding system therefore must be able to 1) *explicitly model assumptions*; 2) *retract mistaken assumptions*; 3) *automatically retract all beliefs that depend on mistaken assumptions*; 4) *represent multiple possibilities*; 5) *explicitly represent the difference between possible and actual belief.*

This paper presents a plan-inference system built using an assumption-based truth maintenance system (ATMS) that accomplishes these requirements. The system can be used as a tool to represent and understand plans based on assumptions and facts. An *assumption* is a possible belief that is treated as believed true, but may be (nonmonotonically) retracted later. A multi-valued *uncertainty logic*, containing the values **actual, possible, hypothetical,** and **inconsistent**, is used to represent assumptions and the degree of belief in assertions' *current* and *predicted* occurrence. The system uses precondition/effect/decomposition plan schemata with variables in order to represent actions and plans.

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, technical discussions of various aspects of the NP system are presented. Finally, a command dictionary is provided in the appendix for the convenience of the user.

It is strongly recommended that the reader first read the ATMS manual [Mye89b], in order to get a background for the logic and the underlying operations, before reading this manual. Although this manual was originally designed to stand by itself, it is easier to use the two manuals rather than having to repeat most of the ATMS manual in here. Note that the NP system uses a later version of the ATMS than that discussed in the ATMS manual [Mye89b], and some of the commands have been changed or upgraded. All of the important commands have been documented here.

3 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 A conceptualization of a change that happens in the real world. The dual of *state*. Actions supposedly map one *world* into another world. For the purposes of the current version of NP, time is disregarded (e.g., actions are either assumed to occur instantaneously, or the duration does not matter). Actions are represented by *plan schemata*.

Action Schema See *plan schema*.

Actual A belief value. When an *assertion* is **actually** believed, then it is considered to be "real"–it definitely happened in the "real world". Also see *possible, hypothetical, inconsistent*, and *null*.

Alternatives When a person utters a sentence, the speech recognition module and the parsing module create a number of different possible interpretations for that single utterance. These are input to the NP system as a set of **alternatives**. An **alternative** has two distinguishing characteristics: (1) It is a *possible*, not an *actual* observed utterance; (2) It is *pairwise inconsistent* with all other utterances in that alternative set. That is, only one of the **alternatives** will be *believed true*, and the rest *not believed*; however, the system does not know which one to believe.

Antecedent The IF part of an IF-THEN concept. Both *NFL rules* and *implications* have antecedents. Each rule or implication can have one or more antecedents.

Assertion A concept. A "fact" or "statement", that will either be *believed* or *not believed*. Assertions are explicitly represented in the system, by using *feature structures*. Giving a concept to the system is called making an **assertion**, or **asserting** a statement. An assertion can have an interpretation or belief value of *actual*, *possible*, *hypothetical*, or *inconsistent*. Since an assertion by definition must be represented inside the system, it is techically impossible for an assertion to have the belief value of *null*.

Assume To believe that a concept is *possible* (as opposed to *actual* or *hypothetical*). Also, 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 also 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*).

Belief Value A value assigned to a *state* or *assertion* by an observer describing whether the observer believes that that state corresponds with the real world or not. **Belief values** are used by the NP system instead of *truth values* to interpret results. The system currently uses a five-valued **belief value** system. See *actual, possible, hypothetical, inconsistent,* and *null.*

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

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

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

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

Consequent The THEN part of an IF-THEN concept. Each *implication* has one **consequent.** *NFL rules* can have more than one **consequent.**

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

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

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

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

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

Deletion Physically removing an *item* from the *knowledge base.* When an item is **deleted,** its truth value becomes *null.* See *retraction.*

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

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

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

Feature Structure A particular data representation method. A **feature structure** comprises a list of features. Each feature has a value that can be an atomic value or a feature structure. The *NP* and *NFL* systems use the Hasegawa-style Nadine feature structures.

Hypothetical A truth value. Assertions that are *hypothetical* are known to the system, but are not believed true, nor believed false. The system has no opinion about them. The system simply knows that the assertions could exist.

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. See *justification.*

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

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

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

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

Inconsistent is also a belief value for a concept, corresponding to permanently *not believed.* Concepts that are *inconsistent* will never be *believed true* by the system.

Invalid *Inconsistent.*

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

Justification A justification is actually the same as an *implication*, but the conceptualization is different. A believed ATMS-node that is not an assumption must have at least one implication that **justifies** why this node is believed. The node is the *consequent* of the justification, and the node is justified by the *antecedent* nodes. All of the antecedent nodes must be believed in order for the nodes to "actually justify" the consequent; otherwise, they simply "potentially justify" the consequent. The justification is the link between the antecedents and the consequents. A justification

is both this concept, and an alternative name for the *implication* data structure that represents this concept.

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

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

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

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

Mutually Inconsistent A set of two or more items is **mutually inconsistent** if all (the *conjunction*) of the items cannot be *believed true* at the same time (i.e., in the same *possible world*). For a set of n mutually inconsistent items, it is alright to have any $(n-1)$ items be *believed true*. Mutual inconsistency is implemented in the *ATMS* by building a single *implication* that has all of the items as its *antecedents* and the *nogood-node* as its *consequent*.

NFL The **N**atural language **F**act/Rule **L**anguage (or, the **N**adine-based **F**act/Rule **L**anguage). An inference-engine system that works with feature structures as its basic input and output, and asserts its results into the ATMS. Since NFL uses the RWS rewriting system to pattern-match new assertion feature-structures against all of the rule patterns, it is rather slow.

Node An ATMS-node, Assumption, or Premise.

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

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

Not Believed A truth value for a concept (ATMS-*node*) in a particular *possible world* (*context*). NOT BELIEVED corresponds to UNKNOWN in a trinary TRUE/FALSE/UNKNOWN logic. See *believed*. Other ways of thinking about NOT BELIEVED include DISREGARDED, or NO OPINION. Note that NOT BELIEVED is not the same as FALSE; there is no way to explicitly represent FALSE using an ATMS.

No Opinion NOT BELIEVED.

Null A belief value. Concepts that are null are completely unknown to the system. The system has no opinion as to whether the concept could be true or not. The system has no representation for the concept.

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.

Pairwise Inconsistent A set of two or more items is **pairwise inconsistent** if (at most) only one item in that set can be *believed true* at any one time (i.e., in any one *possible world*). This follows because if any two or more items in the set were to be believed, the belief would be inconsistent. For example, a set of input utterance *alternatives* is **pairwise inconsistent**–only one of the alternative utterances can be *believed true*, and the rest must be wrong. However, it is not known which one to accept. **Pairwise inconsistency** is implemented in the *ATMS* for a set of n nodes by iterating through the $n(n-1)/2$ set of all possible pairs of nodes, and setting the conjunction of each pair to imply the *nogood-node*.

Plan Schema A **plan schema** is a form of representation for a single *action*. A **plan schema** names the action, and defines it. **Plan schemata** are composed of a list of precondition states that are necessary in order for the action to take place; a list of effect states that become true after the action takes place; and a list of decomposition actions that compose the defined action.

Plan Schemata More than one *plan schema*. The representation method used to represent *actions*.

Possible A belief value. If an *assertion* is **possibly** believed, it could be true, or it could not be true. Possible beliefs are used to represent *alternatives*.

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

Premise A concept that is considered to be always true, no matter what. Technically, a premise is BELIEVED in all possible worlds. A premise cannot be retracted, but it can be deleted. Premises represent the truth value *actual*.

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 (setting its truth value to *null*), whereas retraction simply removes the use of the node by the system (setting its truth value to *inconsistent*). Items can now be deleted in the current system.

Schema See *plan schema*.

Schemata This is the (irregular) plural form of the word *schema*.

State A basic concept. A logical sentence or *assertion* about the conditions of certain things in the world, along with an associated *belief value*. States are timeless. *Actions* are thought of as being a change between states or state values.

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

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

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

Truth Value A value associated with a particular *state* or *assertion*, defining whether that state corresponds to the real world or not. Traditional binary **truth value** systems used the values TRUE and FALSE; new trinary **truth value** systems use the values TRUE, UNKNOWN, and FALSE. The objective truth of whether a state actually exists in the real world usually cannot validly be determined. It seems always necessary for an observer to subjectively determine the truth of an assumption. For this reason, the NP system uses a five-valued *belief value* system instead of a **truth value** system for actual result interpretation.

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.

Utter To speak with the mouth. To make a single phrase or sentence known as an *utterance*, that is treated as a single unit.

Utterance A sound, consisting of a single phrase or sentence, that is made by a person wanting to communicate. Also, the *feature-structure* representation of that sound, as obtained from the results of the speech recognition module and/or the syntactic/semantic parsing module. Utterances are *asserted* into the system, as input.

Valid Not *inconsistent.*

World See *possible world.*

World State The set of all significant *states* defining a *possible world.*

4 Data Type Explanation

This section presents a description of the NP system's data types. The data types are divided based on whether they belong to the NP plan inference system, the NFL inference engine, or the ATMS truth-maintenance system.

4.1 NP System Data Types

Plan Schema A description/definition of a single action, used in the recognition and inference of plans. A plan schema has an action name or description, a series of preconditions, a series of decompositions, and a series of effects. All of the components of the plan schema, including the action name or description, can include variables. All of the components of a plan schema, and the plan schema itself, are represented using feature structures. (The plural of schema is "schemata", this word is irregular in English.)

Sufficiency Set An optional specification attached to a particular plan schema. Normally, an action requires that the set of all of its preconditions and decompositions be present in order to be recognized. The specification of sufficiency sets allows subsets of these to recognize the action. This is useful in cases where the action has multiple alternative decompositions.

Assertion (Utterance) A logical structure representing a particular complex concept inside the system. Whereas an assertion is any concept, an utterance is a concept or phrase that has been spoken by one of the conversation participants. Assertions are represented using feature structures. Assertions are assigned a logical belief value by being stored in ATMS-nodes. Assertions are used to represent utterances, facts or statements about the conversation, common-sense knowledge statements, and derived results. Utterance assertions are used as input to the system. The following five types of utterance assertions are distinguished:

Preinstantiation Utterance, or Hypothetical Utterance This is an utterance that the system thinks beforehand *could* be said in the actual conversation. The system uses the hypothetical utterances to preinstantiate chains of reasoning. The system gives no commitment whatsoever to the preinstantiation utterances; it does not believe that they exist in the real world.

Actual Utterance This is an utterance that the system believes actually exists in the real world. The system is completely committed to this utterance. This category also includes assertions about domain knowledge and common-sense facts known to be true. Actual utterances are used to input data from conversations known with certainty.

Possible Utterance This is an utterance that the system believes *may* actually exist in the world. The system believes that the utterance may or may not have been observed. This category also includes assertions about uncertain domain facts. Note that in most cases the following classification, *alternative utterances*, will be used instead of this one.

Alternative Utterances These are a mutually exclusive set of utterances, one of which the system believes *may* actually exist in the world. Alternative utterances must always be entered in sets. Only one utterance from the set is allowed to be believed in any one possible world. Thus, the system only believes one utterance at a time, but it explores the possible belief of each utterance in the set simultaneously. The system does not force belief in one of the alternatives; it is possible for all of the alternatives to be not believed. Alternative utterances are used to input data from conversations that have been recognized with uncertainty, where each uttered phrase has many alternative candidate utterances, and it is not known for certain that the actual utterance is in fact contained in the alternatives.

Goal Utterance This is an utterances that is possibly or definitely known to represent the goal of one of the conversation participants. In the current system, these must be extracted and explicitly asserted to the system. Goal utterances are used for plan prediction and inference. The current plan inference method does not bother to discriminate between actual goals and possible goals; both trigger plan inference.

4.2 NFL System Data Types

NFL-Fact An assertion made to the NFL system. NFL-facts are represented by using feature structures. NFL facts are always hypothetical; the NFL system does not distinguish between the actual, possible, and hypothetical belief values, nor between different possible worlds. NFL uses the ATMS to perform these functions. NFL-facts get placed in a single fact-pool.

NFL-Rule An inference rule that is specified to the NFL system. NFL rules are composed of a list of antecedent patterns, a list of consequent patterns, and a special optional list of effect-consequent patterns. Each of these components is in the form of a feature structure, and each feature structure can have variables. When all of the antecedents consistently match a particular set of facts in the NFL fact-pool, the consequents and the effect-consequents are instantiated, and the entire inference instance (antecedents imply consequents, and the first consequent implies all of the effect consequents) are instantiated (hypothetically) into the ATMS. If the flag `*nfl-propagate*` is T, both the consequents and the effect-consequents are inserted back into the NFL fact-pool.

4.3 ATMS System Data Types

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

ATMS-node A node. Otherwise known as a Concept, a Statement, or (sometimes, depending upon the usage) an Assumption. Nodes are used to store Utterances or Assertions.

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

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

In addition, each ATMS-node can merely be a simple node, or it can be modified to become one of the following two mutually-exclusive subtypes:

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

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

5 Command Type Explanation

This section presents a description of the NP system's commands. The commands are divided based on whether they belong to the NP plan inference system, the NFL inference engine, or the ATMS truth-maintenance system. In addition, the commands are arranged by the type of function they perform.

5.1 NP Plan Inference System Commands

This section presents the commands used for the NP Plan Inference system.

5.1.1 NP Creation Commands

Utterance Assertion Commands. These commands assert utterances into the NP system, with varying levels of realization. The utterance takes the form of a single feature structure, or a set of mutually exclusive feature structures (e.g., from the results of an ambiguous process). Actually, the feature structure does not have to represent an utterance–it could represent world knowledge or other assertions just as easily.

(preinstant-utt FS) This command enters an utterance into the NP system for preinstantiation. The utterance is then a hypothetical fact or concept that the system knows about, but does not believe exists in the world yet. Preinstantiation utterances are used to allow the system to do (slow) reasoning off-line, ahead of time. The system reasons with the utterance, even though the system knows the utterance is only hypothetical.

(hypothetical-utt FS) This command enters a hypothetical utterance or concept into the NP system. It is actually the same as `preinstant-utt`.

(actual-utt FS) This command enters a feature structure representing actual utterance or concept into the NP system. After this, the system will believe, with certainty, that the utterance happened or the concept exists in the "real world". If the system has seen the concept before with a hypothetical or possible value, no (slow) reasoning is performed. However, the system does update the values of resulting implications, in a rapid manner. If the system has not seen the concept before, first the concept is instantiated hypothetically and reasoning is performed using the concept; next, the concept's value is upgraded to ACTUAL and the system correspondingly updates implied concepts.

If one of a set of pairwise inconsistent concepts is asserted as actual, the remaining concepts automatically immediately become inconsistent.

It is a mistake to assert as ACTUAL a concept that is already INCONSISTENT.

(possible-utt FS) This command enters a feature structure representing a possible (uncertain) utterance or concept into the NP system. After this, the system will believe, with uncertainty, that the utterance may have happened or that the concept may exist in the "real world". However, other possibilities are also allowed. If the system

has seen the concept before with a hypothetical value, no (slow) reasoning is performed; the system simply updates the value of the representative ATMS node to POSSIBLE, and propagates the resulting implications in a rapid manner. If the system has not seen the concept before, the concept is sent to the NFL inference engine and reasoning is performed with it in a hypothetical manner. Next, the concept is then upgraded from HYPOTHETICAL to POSSIBLE.

It is a mistake to assert as POSSIBLE a concept that is already ACTUAL or INCONSISTENT.

(alternative-utts FS1 FS2 ...) This command enters a set of feature structures representing pairwise inconsistent possible (uncertain) utterances or concepts into the NP system. After this, the system will believe, with uncertainty, that any one utterance from the set may have happened or may exist in the "real world". Only one concept or utterance out of the set will be true, and the rest will be false. However, the system will not know which one is true, and will explore all possibilities. It is also possible that none of the utterances are true. If the system has seen a concept before, no reasoning is performed with that concept; otherwise, the system uses the concept for inferencing. In any case, the resulting updates to the given node values are propagated.

If, later on, one of the alternatives is re-asserted as ACTUAL, all of the other alternatives immediately become INCONSISTENT.

It is a mistake to assert as an alternative any utterance that is already ACTUAL or INCONSISTENT.

(goal-utt FS) This command enters a feature structure representing a goal utterance or assertion.

(inconsistent-utt FS) This command enters a feature structure that is self-inconsistent and will never be believed by the system in any possible world.

(incon-utt FS) Same as the `inconsistent-utt` command.

(mutually-inconsistent-utts FS1 FS2 ...) This command enters a set of feature-structure assertions that are mutually inconsistent. The entire set, i.e. the conjunction of all of the assertions taken together, is made inconsistent and will never be believed in any possible world. However, of the n assertions in the list, any $n-1$ or less assertions may be believed at the same time in any one possible world.

(mutual-incon-utts FS1 FS2 ...) Same as the `mutually-inconsistent-utts` command.

(pairwise-inconsistent-utts FS1 FS2) This command enters a set of feature-structure assertions that are pairwise inconsistent, i.e. they can never both appear in the same possible world. This is the same as the mutually-inconsistent construction, with $n = 2$.

(pair-incon-utts FS1 FS2) Same as the `pairwise-inconsistent-utts` command.

5.1.2 NP Modification Commands

Currently there is no way to modify a fact or rule in the NP system.

5.2 NP Commands

(NP-Action [plan-FS]) Declares a plan schema to the system. The schema should be an explicit feature-structure. The semicolon character, ";", supports to-end-of-line comments, even inside the feature structure. It is important that the action have at least one precondition or decomposition; otherwise, it will never be instantiated and will be useless. The current version is UNABLE to accept extra features in the data to be matched, that are not described in the plan feature structure.

(NP-Input "documentation-string" [data-FS]) Declares an input data assertion to the system. The schema should be an explicit feature-structure. The semicolon character, ";", supports to-end-of-line comments, even inside the feature structure.

5.3 NFL Inference Engine Commands

This section presents the commands used for the NFL inference engine.

5.3.1 NFL Creation Commands

(nfl-rule ant-FS1 ant-FS2 ... ant-FSn consq-FS) This command is the main user interface for entering a rule into the NFL system. A rule is specified, such that a series of antecedents implies a consequent. Because of the syntax, currently only one consequent pattern can be specified. All of the feature structures can have variables. However, the variable names should be the same between feature structures. The feature structures currently must be in internal format (they must have been read in already). This function returns the created nfl-rule object. There is no particular reason why the user system should save this object.

(nfl-rule-list '(ant-FS1 ant-FS2 ...) '(consq-FS1 consq-FS2 ...) &optional '(eff-FS1 eff-FS2 ...)) This command is the main system interface for entering a rule into the NFL system. A rule is specified, such that a series of antecedents implies a series of consequents, and then the LAST consequent (only) implies an (optional) series of effect consequents. The command takes (two or) three lists as input; each of the series of patterns must be a list of feature structures in internal format. This function returns the created nfl-rule object. There is no particular reason why the user system should save this object.

(nfl-fact FS) This command is the main interface for entering a fact into the NFL system. A fact must be in internal feature-structure format. In the current system version, all of the NFL-rules must be entered into the NFL system before the NFL-facts start being entered. The system only tests facts against those rules that are already there. When a new rule is entered, it is currently not tested against any facts that might already be there. This was made expedient by the implementation of the pattern-matcher, which uses the rewriting-system engine.

(clock-nfl-stack) This command pulls the first rule instantiation off the top of the stack, examines it, and fires the rule if the antecedent arguments are consistent. It returns `T` if an entry was found, and `nil` if the stack was empty.

Since this function is now called with a `(loop while (clock-nfl-stack))` at the end of `(nfl-fact)`, there is currently no need for the user to ever use this function. This may change to more explicit control in the future.

5.3.2 NFL Modification Commands

Currently there is no way to modify a fact or rule in the NFL system.

5.3.3 NFL Deletion and Initialization Commands

Currently there is no way to individually delete a fact or rule in the NFL system.

(reset-nfl) Resets the NFL system. Clears out the rules, their patterns, and the active rules stack. (In the current system there is no fact table to clear). Caution: ALSO RESETS THE HASEGAWA RWS SYSTEM.

5.3.4 NFL User Query Commands

(nfl-rule-p item) Tests whether `item` is an nfl-rule or not.

5.3.5 NFL User Output Commands

(print-nfl-rules) Prints out all nfl-rules.

(print-nfl-rule nfl-rule) Prints out a single specified nfl-rule.

(print-nodes) Prints a list of *all* of the nodes in the ATMS. This includes at least all of the NFL facts known to the system.

5.3.6 NFL User Access Commands

(nfl-rule# n) Accessor function for nfl-rules. Returns the object representing nfl-rule #n.

(nfl-pattern# n) Accessor function for nfl-patterns. Not normally used.

5.3.7 NFL Explanation Commands

The firing of NFL rules can be backtraced by using the explanation capabilities of the ATMS. See Section ??. In particular, the functions `(why-env-assums fact)` and `(why-envs fact)` are useful, for a given fact feature structure.

5.3.8 NFL Significant Variables

nfl-rule-count This variable contains an integer that tells the number of nfl-rules that have been entered into the system. The default is 0.

nfl-pattern-count This system variable contains an integer that tells the number of nfl-patterns that have created by the system. The default is 0. The user should not need to use this variable.

nfl-answer-stack This system variable is used by NFL to interface with the RWS rewriting system. The pattern-match answers from the RWS system are pushed on this stack by RWS and pulled off by the NFL system. The user should not need to use this variable.

nfl-rules This variable stores a hash-table of all the rules known to the NFL system.

nfl-patterns This variable stores a hash-table of all the rule patterns known to the NFL system.

nfl-active-rules This variable stores the NFL "stack", a heap of all of the activated rules that are waiting to be processed and possibly fired if consistent. Since the current system always reprocesses this "stack" until it is empty, this variable should always contain an empty heap when examined by the user.

nfl-answer-stack This system variable is used internally by the NFL system when accepting results from the RWS system. The RWS is made to push its bindings on this stack when a new nfl-fact is submitted for pattern recognition. The NFL system then takes the results of the recognized patterns and uses them to put rule instantiations on the active rules stack. If no antecedent patterns match, this variable is `nil`.

5.3.9 NFL System Flag Variables

nfl-propagate This flag determines whether the NFL system asserts the consequents of rule firings back as facts into the NFL system, therebye propagating them and firing more rules, along with entering the results of the fired rule into the ATMS (`T`); or, whether when a rule fires the results of the fired rule are simply entered into the ATMS, but not propagated as new facts to the system (`NIL`). The default is `T`.

nfl-assert-unused-facts This flag determines whether orphan facts that are not used by any rule are entered into the ATMS (`T`) or are simply forgotten (`NIL`). The default is `T`.

nfl-debug When this flag is non-NIL, each new or propagated fact is printed out as the first action performed by `nfl-fact`. This flag is useful in detecting infinite loops in rule sets. The default is `nil`.

nfl-dont-repropagate When this flag is true, the NFL system does not re-use any newly re-asserted facts that it has already seen before, but instead throws them away before they are compared with any rules. Telling the system about a fact once is enough. Since in the current version it is expected that all of the rules will have been predefined, there is no problem using this feature. This flag prevents duplication of results, and basically enforces monotonic behavior for the inference engine. If the inference engine is expected to obtain nonmonotonic results, this should be set to `nil`.

This feature actually works by testing the ATMS table, using the uniquification algorithm. If the fact is already in the ATMS (even if it's only hypothetical), it is not used.

The default is `T`.

5.4 ATMS Truth Maintenance System Commands

5.4.1 ATMS Creation Commands

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

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

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

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

(assumption data) Constructs and returns an Assumption node storing the given information. For future expansion, it is possible to assign a probability number to the assumption when it is created, by calling `(assumption data prob)`. Currently, the probabilities are not used otherwise by the system.

(implication consequent data antecedent1 A2 ...) Constructs and returns an implication. This function is mostly for human users. Same as `(justification ...)`. The consequents and the antecedents can either be atms-nodes or data. The system will check each consequent and antecedent node to make sure that it is in fact a node; if not, it will use the old node containing that data, or it will create a new atms-node for that data if necessary.

(implication-list consequent-node data (list antecedent1 A2 ...)) Constructs and returns an implication. This function is useful when you have a variable containing a list of the antecedents. The consequents and the antecedents can either be atms-nodes or data. The system will check each consequent and antecedent node to make sure that it is in fact a node; if not, it will use the old node containing that data, or it will create a new atms-node for that data if necessary.

(sys-implication consequent-node data antecedent-node1 A2 ...) Constructs and returns an implication. This function is mostly for computer users. Assumes that the consequents and antecedents are nodes already, and does not check for legality. This results in significant speed gains, at the cost of extra safety.

(justification consequent-node data antecedent1 A2 ...) Same as implication.

(inference consequent-node data antecedent1 A2 ...) Same as implication. The "inference" terminology is supported but not encouraged; use "implication" or "justification" instead.

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

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

(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.

(inconsistent env) Same as `(nogood env)`.

5.4.2 ATMS 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. Optional arguments: Assumption-implication data, and the assumption probability (not used): `(assume-this-node node data prob)`.

5.4.3 ATMS Deletion Commands

(del-atms-node name-or-node) Hard-deletes an atms-node.

(del-implic implication) Hard-deletes an implication.

(unassume name-or-node) Turns a node from an assumption back into a hypothetical node.

(del-env environment) Hard-deletes an environment. Not supported yet.

(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.

5.4.4 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. It is now possible to call this routine several times on the same node, and install several different actions; when the node becomes IN, all of the actions are performed. The action should be of the form '(funcname arg1 arg2). Most of the time, one of the args will be the node itself. If the args are not constants, they must be evaluated: '(funcname ,node ,arg2). The function can have any number of nodes; the literal is simply stored and evaluated later.

5.4.5 Significant Variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4.6 System Flag Variables

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

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

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

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

6 What is a Plan?

NP is a *plan inference* system that attempts to recognize and understand *plans*. For the purposes of this work, a *plan* is a series of actions that, when taken together, lead to a *goal*. A *goal* is a state of affairs, or *situation*, that is desired by an agent. Normally, it is assumed that the agent *intends* to obtain the goal, and thus will be following the series of actions composing the plan. For this reason, if the goals of the agent are known, and the current actions of the agent are known, then the future actions of the agent can be inferred.

The characteristics of the current system are as follows:

- The system has no explicit representation for time.
- The system can represent multiple alternative possible worlds, consisting of alternative plans. However, the multiple worlds are timeless; in a sense, the system always lives in *now*.
- The system recognizes plans with monotonic actions. Although it is possible for the user to nonmonotonically retract assertions that are believed concerning the initial situation, it is currently impossible for an action to retract an assertion as part of the action's effects. It is possible to assert the negation of a state, as an action effect; however, in this case, both the state and its negation are believed in the resulting possible world, which is normally considered to be erroneous. Because there is no concept of a transition of states over time, if an initial required precondition in a series of actions is later retracted, the actions also go away. The decision to represent only monotonic actions is similar to circumscription. This restriction makes the plan inference system easier to build, and it makes the system run faster. However, it makes the system less powerful than one that can represent nonmonotonic actions. Plans containing monotonic actions are sometimes called *linear* in the planning literature; those containing nonmonotonic actions are then called *nonlinear*.

7 Format for Plan Schemata

The system is able to recognize plans because the user specifies action templates to the system, in the form of plan schemata. A plan schema is composed of a number of parts. These include: the action name, or a description of the action; preconditions of the action; decompositions; and effects. There must be only one action name or description. All of the other components are optional, and can take none or more entries. Preconditions and effects are typically states, while decompositions are typically other, more low-level, action descriptions. Since the system is based on feature-structures, it is possible to have the preconditions or effects be action descriptions, and the decompositions be states. However, this abuses the model conceptualization, and should be avoided if possible.

The plan schemata are specified using feature structures, in Hasegawa-style Nadine format. One feature is specified for each component. Although the order of the features does not matter to the system, the following order is recommended for notational consistency: (1) the action description, (2) the preconditions (if any), (3) the decompositions (if any),

(4) the effects (if any). The feature slot names are coded to correspond to the components. These code names are in fact special ("magic") features and are contained inside the single feature-structure itself, and so there is no need to provide lists of feature structures for the different components. The action description has the feature slot name of **action**. All the other feature slot names consist of a code plus an integer. The preconditions have the code **prec** (so, for instance, precondition features will have the slot names **prec1**, **prec2**, etc.). The decompositions have the code **dec** (e.g., **dec1**). And, the effects have the code **eff** (e.g., **eff1**). See Section 10 for further explanation and an example.

8 Purpose.

This work is aimed at developing a natural language understanding module to be used in the ATR Interpreting Telephony Research Laboratories' Japanese-English automatic telephone interpretation system. Such a system will minimally include modules for performing speech recognition, syntactic/semantic parsing, natural language understanding, language-dependent concept transference, language generation, and speech generation [KU89, IKYA89]. It is the task of the NP system to help with the natural language understanding module. The natural language understanding component must take input from the parser module, store contextual information about the progress of the conversation, and provide output for the transfer and generation modules. In addition, the understanding system should be able to answer specific queries from the transfer or generation modules should further information be required.

As a first step in creating a full understanding module, this work contributes a plan-inference system and a rule-based inference system. Since the basic data-structure of the parser, transfer, and generation modules is the *feature structure* [Shi86], a powerful frame-like structure popular in natural language processing, both the **NP** plan-inference system and the **NFL** rule-based inference system have been implemented using feature structures as the basic data structure. This is necessary to allow the understanding system to be easily integrated with these other modules.[1] The plan-inference system uses plan schemata in order to be able to represent general types of plan actions. Since the system is to use context-dependent information to reason about the knowledge and intentions of dialog participants, it must be able to match multiple patterns against an unordered set of assertions. It should explicitly represent and work with assumptions, inconsistency constraints, and multiple possible inferred plans.

In the future, the actual input from the parser will consist of multiple possible parses rather than single parses. This is because the speech recognition module produces multiple possible input utterances, and because the parser produces multiple possible semantic parses for each utterance. The understanding system will have to disambiguate between these possibilities. Disambiguation requires the ability to represent and reason with multiple mutually-exclusive alternative inputs for a single utterance. In NP, this is provided by strongly basing both the plan inference and the inference engine on assumptions by using an ATMS, as will be explained.

9 The Domain of the Problem.

The NP plan-inference system was tested by understanding conversations between two people in a single language (Japanese). Utterance parses were obtained from the expected output of the semantic parser, which was machine- and partially hand-generated independently by a parsing expert, Mr. Masaaki Nagata [Nag89]. The input consists of a series of feature structures in textual format. The NP system converts these to internal feature-structure format using the `read-fs` command developed by Toshirou Hasegawa [Has89]. The system then works with this internal representation.

[1] This specification was originally proposed by Mr. Hitoshi Iida, the manager of this project.

The NP system serially processes the utterance parses, attempting to maintain a representation of the currently-believed concepts as the conversation progresses. The system does not take part in the dialog. The conversation is a task-oriented dialog on the subject of registering for a conference. Output consists of a representation of the plan structures found in the conversation, and explicit reports of inferred plans.

10 Plan Schemata.

The system is initialized with action template declarations in the form of plan schemata represented using feature structures. The plan schemata are best understood by considering a simple plan action.[2]

```
[[action  [[RELN Identity-of-other-confirmed-1]
           [AGEN ?questioner]
           [RECP ?answerer]]]

 [prec1   [[RELN Confirming-identity-of-other-1]
           [AGEN ?questioner]
           [RECP ?answerer]]]

 [dec1    [[RELN  Hai-AFFIRMATIVE]
           [AGEN ?answerer]
           [RECP ?questioner]]]

 [dec2    [[RELN  Sou-Desu-CONFIRMATION]
           [AGEN ?answerer]
           [RECP ?questioner]]]

 [eff1    [[RELN Know-Identity]
           [AGEN ?questioner]
           [OBJE ?answerer]]]   ]
```

A schema has an action name or description, as well as a series of preconditions, decompositions, and effects. Plan schemata are formed from (possibly cyclic) feature structures and can include variables, "co-instance tag" variables and "rest" variables.

11 The NP Plan-Inference System.

The NP system is composed of three layers: the *plan inference* layer, which consists of a conceptual model for plan inference plus routines to implement this model using an inference engine; the *NFL inference engine* layer, which consists of pattern matching routines, control routines, and routines to assert concepts and implications into an ATMS;

[2]This plan says that an "Identity of other confirmed" action necessarily occurs when a precondition question of "Confirming identity of other" has been asked, and decomposition answers of "Hai" ("Yes") and "Sou desu" ("It is") both occur. Then the effect of "Know identity" must also occur. This action is part of a plan for learning the identity of a caller at the opening of a telephone conversation.

and the *ATMS* layer, which records the results of the system, propagates implications and maintains consistency, responds to degree-of-belief requests, and fires user-specified, event-driven processing demons. These layers will now be discussed in turn.

12 The Plan Inference Layer.

Plan inference requires two things: conceptual models of the plan recognition, prediction, and inference processes, and a method of instantiating these models using the rules of an inference engine.

A strong view of plan recognition is taken. Recognition of an action occurence implies the occurence of each of the action's effects. Recognition is based on *necessary entailment*—observation of all of the preconditions and decompositions forces recognition of the action. This faithfully models certain kinds of actions, such as those defined by *conventional generation* or other types of *generation* processes [Gol70]. Other kinds of actions, where inputs can have multiple interpretations (such as Kautz's hunting example [Kau87][3]) require more explanation. First, if the definition of an action allows unwanted ambiguity when recognizing inputs, then that definition is incomplete and must be augmented with the appropriate preconditions or decompositions. However, some of these states required to complete the action's definition may be *unobservable* [Mye88]. In this case, such states must be represented by *assumptions*, which have the belief value POSSIBLE. In effect, the assumption simultaneously explores both the cases in which the state is true and those in which it is not true. Second, in some cases the requirement that all the decompositions or preconditions be present is simply too strong. Also, the action's definition might have multiple alternative decompositions. In these cases, the model can be weakened by specifying explicit precondition and decomposition state *sufficiency sets* which certify that the action has occurred.[4] (E.g., {prec1, dec2} is sufficient for the previous page's example.)

The preceding discussion concerns (certain) input states which are *actually* observed, resulting in ACTUAL recognition of the actions. It is also possible to have uncertain inputs that are *possibly* observed, resulting in POSSIBLE recognition. In addition, it is possible to have multiple conflicting alternative inputs, where only one input corresponds to reality. In this case, the recognized actions are POSSIBLE as well. This capability is significant for representing ambiguous spoken language input.

The prediction model, in contrast to the recognition model, uses a weak method similar to spreading activation. Each assertion is duplicated in a parallel top-down network where it is marked PREDICTED. The possible or actual declaration of a goal sets the corresponding predicted-state's value to possible or actual. Goals can be genuine states or action-occurrence states. Prediction of any one of the effects of an action causes prediction of the action. When an action is predicted, the inference is made that each of the action's

[3]A man walks into a bank holding a gun, possibly in an attempt to rob the bank, or possibly only cashing a check after going hunting. The situation must be disambiguated based on the man's intent, an unobservable state. Note that if there is a law against walking into a bank with a gun, the example is unambiguous and the man's intent does not matter; the occurence necessarily entails that that law has been broken (by conventional generation).

[4]This is a refinement of Knoblock's necessary and sufficient conditions [Kno88].

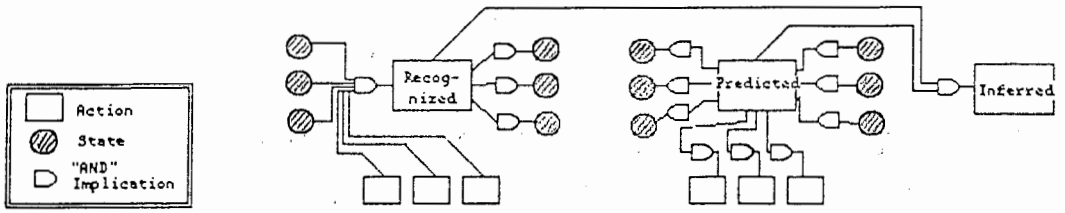


Figure 1: Models for Plan Recognition, Prediction, and Inference.

preconditions and decompositions is predicted as well. Prediction thus propagates top-down through the predicted decompositions, and in a backward-chaining manner through the predicted preconditions.

The system supports specification and simultaneous exploration of multiple possible goals. Due to the facilities of the ATMS, each activation spread is labeled with the name of the goal assumption that originally caused it. Thus, if there are multiple goals, it is possible to query a possibly predicted node about which assumption causes it to be believed. If more than one goal is contributing to its belief, more than one answer will be returned.

The inference model is very simple: any concept in which both the current occurrence and the predicted occurrence are believed POSSIBLE or ACTUAL signals an inferred plan. The plan has been completed through the current occurrence, and is predicted to (possibly) continue through the predicted occurrences to the goal that caused the predictions.

Naturally, it is possible to implement other models of recognition, prediction, or inference, using the system. An examination of these specifications reveals that the current system can infer plans with monotonic actions (although particular states may be retracted in a nonmonotonic fashion) without resorting to searching. Nonmonotonic extensions are being investigated.

The plan inference models are instantiated using inference rules. Each input plan schema is interpreted into a series of NFL rules, according to the models. Originally the rule for the preconditions and decompositions inferring the action, and the rule for the action inferring each effect, were separate. However, this required that the action name possess all variables found in the effects (otherwise unbound-variable problems resulted). Currently, recognition is instantiated with a single rule using the special syntax explained below. Prediction and inference are instantiated in the same manner.

13 NFL, A Feature-Structure-Based Inference System.

The NP system is based on **NFL**, a forward-chaining inference engine that uses feature structures as its basic representation for assertions and rule patterns. It is tied to the ATMS and instantiates all successful rule firings into the ATMS.

The NFL inference-engine data consists of an unordered set of assertions and a set of rules.[5] A rule consists of a conjunction of antecedent patterns and a set of consequent

[5]Although NFL can employ rule priorities to determine firing order, in the problems encountered thus far no need has been found to do this.

patterns. The antecedent patterns of all rules are kept together in a system set. Each new assertion is matched against this antecedent pattern set. If the assertions match all of the antecedent patterns of a particular rule in a consistent manner, then that rule's consequent patterns are instantiated and asserted. A stack is used to help maintain order during processing. Pattern matching and binding consistency checking are provided by parts of a nonmonotonic feature-structure rewriting system [Has89]. Control flow is performed with a simple Rete algorithm [For82][BFKM85].

NFL rules have an optional additional class of consequents called *effect consequents*, designed to support action representation. When a rule fires successfully, all of its consequents and effect consequents are instantiated and asserted back into NFL's assertion set. In addition, the bound antecedents, consequents, and effect consequents are instantiated into the ATMS. Implications are created between the conjunction of the antecedents and each consequent, and also between the first consequent and each of the effect consequents. Typically, the antecedents will represent action decompositions and preconditions; a single consequent will represent the action performance; and the effect consequents will represent the action effects. This particular ATMS structure is designed to reflect the model of an action, and makes ATMS tracing easier.

14 The ATMS Layer.

An ATMS, originally proposed by deKleer [dK86a], is a special kind of data base. A feature-structure assertion is stored in an *atms-node*, which has an associated truth value. The atms-nodes are linked by *implications* (or *justifications*), which take a number of antecedent nodes and a consequent node as arguments. If the antecedents are all true, the system ensures that the consequent is true as well. These results propagate. Thus, the name "truth maintenance". Further details can be found in [Mye89b].

The ATMS's representation of an assertion's value can have at least two possible interpretations. The customary interpretation is that each assertion takes on one value of a two-valued logic {BELIEVED, NOT BELIEVED} in multiple possible worlds. The interpretation followed here is that each assertion takes on one value of a five-valued uncertainty logic {ACTUAL, POSSIBLE, HYPOTHETICAL, INCONSISTENT, NULL} in a single world [Mye89a].[6]

The ATMS uses instantiated assertions (with the variables bound). The system operates the ATMS by instantiating a network of hypothetical assertions and implications, representing prederived conclusion chains from NFL rules. Later, while processing the conversation, some assertions are recognized as being possible or actual, so the system modifies

[6]This is abstracted from the ATMS by the following method: nodes that are premises or derived solely from premises are permanently IN in all present and future possible worlds and are ACTUAL. Other nodes that are IN are POSSIBLE. Nodes that are permanently OUT ("nogood") are INCONSISTENT. Other nodes that are OUT are HYPOTHETICAL. Assertions or concepts that have no representative atms-node are NULL. Note specifically that simply because a node is believed true in all known consistent possible worlds, it may not be ACTUAL–it might only be POSSIBLE. This is because later nonmonotonic information could render it NOT BELIEVED in some new possible worlds. See [Mye89a] for further discussion.

In addition to simple retraction, which changes a node's value from POSSIBLE to INCONSISTENT, it was found useful to create a true delete function that changes a node's value from POSSIBLE to NULL. This is convenient for setting and clearing processing flags.

the value of the existing hypothetical node. If all of the antecedents of an implication become possible or actual, the ATMS modifies the value of the implication's consequent. Naturally, this effect propagates. Since truth maintenance is essentially done by spreading activation on a network which has already been instantiated, following a series of inferences for NFL is quite fast and involves no pattern matching. This results in saving considerable time compared to working with an inference engine, when the system deals with concepts that are already known hypothetically.

The ATMS always represents the current state of the system's beliefs. An external system (e.g. the generation module) can query the ATMS as to whether a particular assumption is actually or possibly believed. The ATMS can print out a list of newly believed assertions. Also, the user can attach event-driven processing demons to specified assertion nodes, which fire when the assertion becomes possibly or actually believed. These are typically used to process and report derived results, including reporting inferred plans.

15 Operation of the NP System.

The system is initialized with a set of plan schemata files chosen from a library (this capability is important for plan design). The plan inference layer takes these plan schemata and interprets them into NFL rules. At this point the variables in the rules are unbound.

At prerun-time, the system is fed a list of initialization concepts, again in feature-structure format. These are assertions that *a priori* can be assumed to be significant, including hypothetical utterances, world knowledge, and common-sense knowledge. The system submits these to the NFL assertion set, thus triggering rules which instantiate conclusions and assert the bound results hypothetically into the ATMS. Possible goals are also asserted here.

Next, at run-time, the input utterances are asserted one by one, in the following manner: the system first tests to see if the utterance is hypothetically known to the system. If it is not, the system submits the utterance to the NFL assertion set, and expends the effort required to follow NFL inferences and instantiate hypothetical nodes as before. After this, the system upgrades the utterance's node according to the certainty of the observation. Certain utterances are set to ACTUAL (premised). Uncertain utterances and multiple alternative utterances are set to POSSIBLE (assumed). In addition, a multiple alternative utterance is specified as pairwise inconsistent with each previous utterance in its alternatives set (i.e., the pair's conjunction is set to imply "nogood"). If a possible goal is recognized by the system, instead of assuming the concept itself, the prediction of the concept is assumed. Since at this point no matching is done and the system is essentially executing productions on nodes which have already been instantiated, the operations of recognition, prediction and plan inference are quite fast.

16 Multiple Possible Input Example.

A common problem in Japanese speech recognition is distinguishing sentence-final *ka*, a question marker, from *ga*, a moderator. The following example demonstrates the sys-

tem's capabilities to accept multiple input alternatives, infer plans, and represent multiple possible output plans. The example has been simplified for clarity.

Plans that are used in this example include: If one communicates *moshi-moshi-open-dialog*, then one is greeting. Greeting is one form of opening communications. Having the identity of the other speaker confirmed is another way to open communications. As shown previously, having the identity confirmed comprises having an outstanding confirming action occur (e.g., a request to confirm identity has been made), and both *hai-affirmative* and *sou-desu-confirmation* occur. One effect is that the identity is then known. Moreover, a statement with a *ga-moderation* can be an indirect request.

A caller opens with "Moshi-moshi" ("Hello"). Two alternative inputs are submitted for the next utterance: "Sochira wa kaigi-jimu-kyoku desu ka."("Is that the conference office?"), and "Sochira wa kaigi-jimu-kyoku desu ga." ("That is the conference office..."). The reply is "Hai. Sou desu." ("Yes. It is."). The inputs for the second utterance alternatives are:

```
[[RELN  S-REQUEST]                          [[RELN  Ga-MODERATE]
 [AGEN !X03[[LABEL  *GUEST*]]]               [OBJE [[RELN  Da-IDENTICAL]
 [RECP !X02[[LABEL  *OFFICE*]]]                     [IDEN [[PARM !X01[]]
 [OBJE [[RELN  INFORMIF]                                  [RESTR [[RELN NAMED]
        [AGEN !X02]                                                [ENTITY !X01]
        [RECP !X03]                                                [IDEN Kaigi-Jimu-Kyoku-1]]]
        [OBJE [[RELN  Da-IDENTICAL]          [OBJE [[LABEL *OFFICE*]]]]]]
              [IDEN [[PARM !X01[]]
                    [RESTR [[RELN  NAMED]
                           [ENTITY !X01]
                           [IDEN  Kaigi-Jimu-Kyoku-1]]]]]
              [OBJE !X02]]]]]]
```

The NP system is preloaded with various hypothetical utterances and concepts. The system assumes *a priori* that the caller wants to open communications. Next, the actual conversation is processed. The *moshi-moshi* utterance is asserted as ACTUAL. The two alternatives for the *sochira* utterance, i.e. the *s-request* and *ga-moderate* structures, are asserted as POSSIBLE and the conjunction is set to be inconsistent. A backtracing processing demon attached to the plan-inferred node is executed, and the system infers the possible plan: `CONFIRMING-IDENTITY-OF-OTHER-1 <- IDENTITY-OF-OTHER-CONFIRMED-1 <- OPEN-COMMUNICATIONS <- S-REQUEST`, among others. Processing continues with *hai-affirmative* being asserted as ACTUAL; *sou-desu-confirmation* will be asserted next. At this point, if an external module queries the system as to whether *know-identity* is believed or not, the result is HYPOTHETICAL ("not yet"). After the *sou-desu-confirmation* is made, this belief will be POSSIBLE.

At the same time, the *ga-moderate* utterance implies a possible indirect-request. Since no action was predicted for an indirect-request, no plan is inferred from this possibility. The *ga-moderate* plan network is inconsistent with the network based on *s-request*, and the results will not mix. However, the system is able to simultaneously explore the results of both possibilities.

The plan recognition, prediction, and inference network for this example is shown in Figure 2. The display system currently has three levels of verbosity. In the network

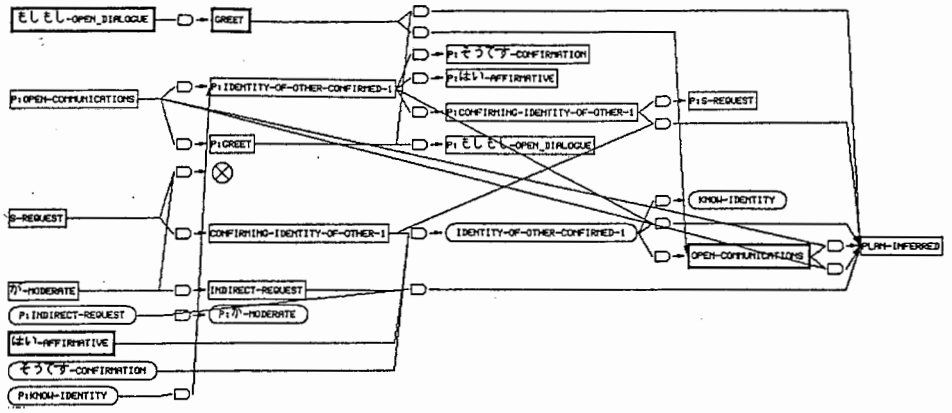


Figure 2: The Plan Recognition, Prediction, and Inference Network for the Example.

illustration, the full feature structure for each node is not shown. For clarity, only the content of the first slot is shown.

In a separate experiment, the system inferred plans in a 20-utterance conversation. During the first run, to represent worst-case behavior, no concepts were preinstantiated. The system had to derive all inferences at run-time, and took 147 seconds. During the second run, to represent optimal behavior, only the 99 ATMS-nodes and 168 implications used in the plans were preinstantiated. The system took 14 seconds to process the conversation. During the third and most realistic run, all of the relevant concepts plus considerable extra knowledge, in the form of 220 ATMS-nodes and 789 implications, were preinstantiated, and the system took 15 seconds to process the same conversation. The running times quoted do not include the time taken to load files nor to preinstantiate initialization concepts, and are expressed in elasped-time seconds on a Symbolics 3620.

17 Comparison with Previous Works.

An inference engine or a plan-inference system is significantly different from a rewriting system (e.g., [Kog89], [EZ89]), in that a rewriting system applies multiple rules to a single input, while both an inference engine and a plan-inference system apply multiple rules to a set of inputs.

Knoblock [Kno88] was the first to use an ATMS for a plan recognition system. His system also worked with multiple output hypotheses for the plan. However, it did not work with plan schemata having preconditions, decompositions and effects, and the corresponding effect-precondition chaining. Plan recognition was done only through the hierarchical decompositions, and the preconditions were used only as a filter for instantiating the actions. Knoblock made no plan prediction from high-level externally-specified goals, and no plan inference. Although Knoblock worked with single uncertain inputs, no multiple possible input sets were used.

Kautz [Kau87] was probably the first to work with multiple simultaneous output hypotheses (in the form of disjunctions). However, he apparently did not work with uncertain inputs nor with multiple possible inputs. Kautz also offered a theory of plan recognition and inference. Kautz used circumscription, and the presumption that all possible plans are known, to infer missing details. We make no such presumption. Kautz's system worked

with ordered events and time, which NP can not yet treat. No assumptions were represented, and input was hand-generated logical forms.

Pollack [Pol86a, Pol86b] explored the important issue of incorrect opinions of plans and the difference between the planning agent's and the observer's concepts of plans, and also worked with nested belief. Pollack worked with a 3-valued logic including "plausible" beliefs but apparently did not deal with multiple output possibilities, multiple inputs, nor uncertain observations.

Other significant plan recognition works are found in [CC89], [LA87], [All87], [AP80], [SA77], and [Wil86].

No previous plan-inference system known to the author has used feature structures as the basic data structure, has accepted input directly from a feature-structure parser, nor has been based on assumptions while using full plan schemata.

18 Discussion.

The NP system is intended to be used as part of an understanding module in an automatic interpretation system. NP is significant in working with feature structures, which allow direct communication with the parser, transfer, and generation modules. A practical plan-inference system must work with realistic parser input, as NP does.

A practical system should accept multiple possible parses and rank their likelihood for disambiguation. Plan inference, as specified, is inherently a logical process that results in assigning an assertion a value of {true, false} in other systems, or {hypothetical, possible, actual, inconsistent, null} in our system. However, a logic-based system cannot support the inherently analogical representation required for ranking systems. The current system has no method of ranking different possibilities, performing evidential reasoning, or determining the degree of probability of a situation. Thus, it cannot yet disambiguate between multiple possible inputs. To be practical, the current system must be supplemented with a well-founded evidential reasoning system. See [Pea88] for an independent exploration of this question.

The current system uses space, in the form of hypothetical assertions, to trade off against the time required to derive rule-based inferences during conversation processing. As always, there is a need to preinstantiate *all*, and preferably *only*, those concepts which will actually be used during processing. However, speed advantages will be realized even if only *some* of the conversation's concepts and their implications have been preinstantiated (e.g., all speech acts, or all domain plans). How to choose which concepts are to be preinstantiated is a crucial research question.

The current system has been designed to compute with inputs from an unordered set of assertions. This has a distinct advantage over systems that work with a strictly ordered representation set, such as those based on parsing technology, in that NFL does not have to perform large combinatorial searches to find conjunctions. The unordered set is a useful representation for working with problems dealing with *belief, ability, desire, decision,* and other modal operators. Belief sets are especially relevant. However, the unordered set (and the corresponding simple multiple-world representation) has known difficulties representing nonmonotonic actions and time [dK86a][LP89][WN88]. In addition, that the evolving

nature of conversations is not captured is a serious deficiency (but one shared by other previous plan recognition systems) which will have to be corrected. A representation such as multiple-action-worlds [MN86] or scripts [SA77] is required for more difficult problems.

The current system works only with what is actually or possibly present. Additional rules and processing demons are needed for problems in supplying implicit information, such as zero-anaphora resolution, ellipsis resolution, and other anaphora resolution. These areas are targets for future research.

Besides these, the current system is deficient in temporal representation, nested belief computations, mistaken belief, and expectation-based processing.

19 Critical Evaluation of the NP System

One of the most useful parts of researching a major computer system is to find out where things that were in the original design go wrong. This section will outline the lessons learned from NP.

The NP system was designed to be relatively simple, fast, and easy to work with. A major design decision was the use of monotonic actions, which allowed the corresponding timeless "big pot" model of plan recognition. All of the actions are defined and thrown together into a "big pot". There is no real difference between an action type and an action instance. As the actions are defined, they automatically combine together with each other in a hypothetical manner, forming chains of actions, consequences, and further actions. Then, when the actual execution starts, there is no need to interpret any rules, perform any pattern-matching, or search by expanding particular action instances in particular worlds–inferencing is performed simply by five-valued-logic marker-passing, which is extremely fast. In this way, the NP system is similar to a connectionist model. Of course, these things *can* be done dynamically by the system as needed, if the system had not thought about them hypothetically before; in this case, the NP system simply takes as much time to execute as a normal system.

Thus, the main advantage of this design is the fact that the system does not have to do any searching ("planning")–all possible useful plans have been hypothetically preconstructed in an automatic fashion.

The alternative is to allow the representation of nonmonotonic actions, as the B-SURE system does. However, in this case the user or some level of the system must explicitly perform searching and plan expansion in each significant possible world, which significantly slows down the system.

This is the main design feature of the NP system. Other features include:

1. Representing non-linear actions. Office doesn't know name -> office asks -> office knows name.

2. Representing future time in the system. I will send the form, you will get it, you will fill it out.

3. How to recognize future time.

4. Choosing or not choosing to do an action.

5. It seems that answers to questions are hard-wired. For instance, automatically the caller wants to say that the caller doesn't have the form.

6. The names of the caller and the office are not included in the input. But the names must be included in the representation, in order to understand that this is a two-person conversation, not one person talking to himself.

7. Knows-how-to X, Wants-to X, and Can X meta-actions are not well represented. X should be tied in with the statements.

8. Should effects follow from the small subactions or the large superactions? Or both? Should preconditions be attached to the small subactions or the large superactions?

9. What if you really want to represent that the agent didn't notice a small subaction had a precondition until he tried to execute it?

10. How to represent state changes? First, the office has the form. Then, the guest has the form. Then, the office has the form.

11. How will the system be used in machine translation?

12. What's the difference between a state and an action?

13. Nondeterministic actions need to be represented. For instance, the caller tries to find out about something can result in the caller knows (successful) or the caller doesn't know (failure). Success is not guaranteed.

14. How to represent ignorance of the agents? The computer knows that the caller does not know where the conference is.

15. How to represent incorrect knowledge of the agents? The computer knows that the caller knows where the conference is, but the computer knows that the caller is wrong.

16. How to represent unspecified incorrect knowledge and disagreements? The computer knows that the caller knows where the conference is, and the computer believes that the conference is somewhere else, but the computer does not know who is right.

17. How to represent ignorance of the computer? The computer does not know something/whether something is true. Knowledge of ignorance: The computer knows that the computer does not know something. Ignorance of ignorance: The computer does not know that the computer does not know something. BOTH are required.

18. Previous, during-, and post-conversation off-camera actions. First the caller decides to call, then dials the telephone. After thinking about things during the conversation, the caller decides to attend. Later, the caller fills out the form and sends it in. Non-observable state changes.

19. Preknowledge of unobservable facts. Caller calls the office implies that caller almost certainly knows the office's phone number. Caller asks for discount implies the caller planned to try to get the discount. But these can't be preconditions, because they are unobservable!!

20. Need to handle conjuction of states implying a situation. Also states implying other states directly. For instance, the caller knows English AND the conference is in English IMPLIES the caller can understand the conference. [Since NP was designed to reason with actions, the current version does not provide strong support for reasoning with states. Additions to the reasoning engine, or entering rules in NFL by hand, are required.]

21. Wants are attached to some actions but not to others.

22. Promises. "I'll send you the form soon" is NOT equal to (The Office sends the form), nor (The Office will-send the form) [could change her mind], nor (I stake-my-honor-on sending you the form). What is a promise?

23. Conjunctive actions. Doing several things at one time with one action. This should probably be represented by an upside-down decomposition.

24. Knowledge of the action implies some preconditions and some decompositions. This is backwards. Think about: Does "tell name and address" decompose into "tell name" (or is it a precondition)?

25. How to represent repeated utterances? "I will send you the form right away." [Since NP is a timeless system, there is no way to reassert something that has been asserted already.] This is a major problem with confirmations, such as "OK", "yes", and "I got it."

26. How in the world do we represent repeated or verified information? "My telephone number is 123-4567." "123-4567, right?" Did he hear it or didn't he? From a domain-planning standpoint, there is absolutely no justification for communicating the same information twice. Why does he need to say it again–with a computer, if it hears something once, then it believes it. What kind of logic can represent "maybe hearing something" or "hearing it but not believing that you probably heard it correctly"?

27. How to represent conditionals? "If you apply next week, it will be 200 dollars."

28. Thinking noises. "Well, let's see." "I understand." Although these play a DIALOGUE function, they do not help any at the DOMAIN PLAN level. How to reconcile these?

29. All actions must have been hypothetically thought of ahead of time. It is hard to integrate new actions or new preconditions into the network automatically. Hypothetically thinking of "The deadline is *next April*" requires considering and instantiating all possible dates [the current NP system does not know how to work well with variables at the ATMS level. All statements are composed of constants or Skolem variables.] What about preknowledge of unobservable facts? "I am a member of the Information Processing Society. Is there a discount?" There are not enough variables.

30. There's a real problem with specialized vocabulary in the current system [since everything is constants]. "I will contribute" vs. "I will give". These should be represented as the same conceptual node, but the current system only represents surface meaning and represents them separately, requiring two lines of reasoning.

31. Inferences of identity. Are two things the same? "The topic of the conference" is the same concept as "what the conference covers", but these are expressed in two different nodes right now.

32. How can the system deal with input that is helpful but not necessary? "We are also expecting psychologists to attend." There is no way to predict this utterance or understand why the person has to say it.

33. Choosing to do something. Roles of the participants. Why doesn't the office want to write a paper to present at the conference?

34. Answering the content of a question is the same as answering the question. This inference is not supported.

35. There are sometimes two ways to resolve a problem. For instance, with information, the caller could either (1) ask the office, or (2) wait for a form to come in the mail with the information on it. When does he do one and not the other? How does he decide?

36. How to represent and reason with an agent putting off the problem for a future date? This is a meta-decision action, and needs to be represented. For instance, "I'll wait for the announcement to come."

37. Generalities. "Know about X." "Know the details of Y."

38. Confirmations and verifications. Why do they happen at all? Why don't they happen ALL THE TIME, or more often than they do? What are the rules that govern when a confirmation is needed?

39. Potential problems and unknown problems.

40. A major problem is that the current system zips illocutionary acts up with the utterances, whereas these are actually two separate things. Break these apart and reason with them separately.

41. Agents are dynamic. Often, the caller will not know what he wants! Or, he might be exploring possibilities before he decides on what he wants. Or, his wants might depend on the situation.

42. Permission. Granting, denying. What rules?

43. There is no temporal ordering in the current system. This is a major problem.

44. How to handle convenient but not necessary preconditions? For instance, it is convenient for the caller to know about the conference in order to attend, but it is not necessary.

45. Standard ordering in conjunctions. Know date and place should be represented in the same concept as know place and date, but currently these are different.

46. The system works by abstracting out the important information and throwing away the unimportant information.

47. For most of the plans, the actual context of the slot is *not known*–all that is known is that the contents are known. How to represent (X knows (Y Knows-Ref A))?

48. Bottom-up vs. top-down action decomposition specifications.

49. How to represent the role of guessing? Also Japanese "deshou".

50. Repeating utterances. I'll send you the form right away.

51. Social smoothings and politenesses.

52. The current mechanism for recognizing goals is weak. Responsibility has been postponed.

53. How to represent volunteering? There is a difference between volunteering information and volunteering to do something.

54. Implicit acceptance of requests. "Please send me the form." "Give me your name and address."

55. Real-world reasoning: Specializations imply generals. For instance, "sentence no desu ga" implies "sentence desu"; "costs 46000 yen per person" implies "costs 46000 yen"; "there is a discount for members" plus "I am a member" implies "there is a discount for me".

56. Conditionals in statements. If vs. When.

57. PLans have to be hard-wired to the example data.

In the final analysis of NP, the system receives high marks for being a very good plan inference system. It understands plans well; it predicts what the person will *do*; and it recognizes when something has happened that it knows about.

However, all this is next to useless when it comes to solving the real problems associated with automatic interpretation, such as utterance disambiguation, "the" vs. "a" determination, understanding the deep meaning of "hai" or "wakarimashita", or interpreting "unagi-da" sentences [MT90]. A plan inference system does not have the machinery to solve these kinds of problems.

In order to build a system that is useful to ATR, it is necessary to design [Mye90] and implement [Mye92] a disambiguation system that is capable of weighting possibilities and deciding between two choices. Only then can the results of a plan inference system be made useful, by eventually contributing to such a system.

20 Assumptions and Understanding

20.1 What Is An Assumption?

An assumption is an assertion or concept that is believed by the system. However, rather than being a fact that is believed with certainty, an assumption has the uncertain belief value of **possible**. The system explicitly considers both the case that the assumption is believed true, and the case that the assumption is not believed. In addition, assumptions can be *nonmonotonic*. After an assumption is made, it is possible that later on the user can find that the assumption was mistaken; the assumption can then be *retracted*. The system builds *inference chains* of other concepts that are based on assumptions; the assumptions directly or indirectly imply belief in the inferred concepts. If an assumption is not believed, all the concepts that depend on the assumption are not believed either. Thus, retracting a single assumption can change the belief value of many concepts.

Currently, most dialog understanding systems start with the assumptions that the hearer and speaker always understand each other perfectly, that they automatically want to cooperate as much as possible, and that they have absolutely no other commitments outside of the conversation. Clearly some of these assumptions can occasionally be incorrect.

It is possible to have two or more assumptions that are mutually inconsistent. In this case, the system automatically constructs different *possible worlds* for each case [dK86a]. After this, whenever a new concept is added to the system, it is automatically added to all relevant possible worlds at the same time.

20.2 Why Are Assumptions Necessary for Understanding?

Communication is inherently an assumption-based process. People use language as a signal to communicate their ideas. However, it is never completely possible to directly know the concepts of another person. Instead, when attempting to understand a conversation, it is necessary to take a stance and rely on assumptions about the other person's thoughts [Den87]. In a dialog understanding system, there are at least two kinds of assumptions: assumptions that the speaker and hearer make (about the conversation and about domain facts) that must be modeled by the system, and assumptions that the system makes about the speaker and hearer.

In most cases, when the conversation is going well, these assumptions will be valid. However, in cases where the conversation fails temporarily, an assumption will be invalid and must be retracted. One of the people may have a mistaken assumption; the system must model this change in belief. The system may make a mistaken assumption about the dialog or about the participants; this assumption must be changed later. In addition, understanding recovery actions taken by the dialog participants after a mistake is aided by recognizing which assumptions are incorrect. It is necessary to explicitly represent the assumptions used in understanding in order to be able to represent and work with such problems.

20.3 NFL vs FS

One problem that rule-based systems have as opposed to feature-structure systems is that rules typically want to match all of an assertion, while feature-structure systems can deal with parts and subparts of assertions, leaving the rest unprocessed. This was partially taken care of by incorporating "?rest" absorption variables into the feature structure patterns for the rules. However, since the ATMS matches assertions directly, it is important to discard irrelevent information before saving assumptions in the ATMS, so that true matches will not go unrecognized because of differences in the irrelevent information.

20.4 ATMS

Since the ATMS only operates on atms-nodes and implications, it is possible to store any type of data in the atms-nodes (including FSs), and have the system perform inferences with this data. However, because of the nature of the ATMS, it almost always does not make sense to store any assertions containing variables into the atms-nodes; the user should store only assertions containing constants or Skolem constants. For this reason, raw rules or rule patterns should not be put into the ATMS; only instantiated rule patterns. However, because the ATMS works with constant nodes, there is no expensive unification or pattern matching to be done. Truth maintenance or following chains of inferences consists mainly of activation propagation, which is done by setting flags in bit vectors and is quite fast.

The system interprets the results of the ATMS by assigning a five-valued logic to each atms-node assertion, consisting of the uncertain belief values ACTUAL, POSSIBLE, HYPOTHETICAL, INCONSISTENT, or NULL. The main values that are currently used are HYPOTHETICAL and ACTUAL.

The belief value of a particular node is indexed to a *possible world*; the same node can be BELIEVED TRUE in one possible world and NOT BELIEVED in a different possible world at the same time.

21 The ATMS

Belief Justification The belief in any assertion can be explained or justified by the system in terms of its underlying assumptions.

The ATMS can explain why a concept is believed in any one possible world. It does this by conceptually backward-chaining on the active justifications for that possible world, until the contributing assumptions are reached. This is necessary because each node has many hypothetical justifications; however, only a few of them will be active for any particular assumption set. Also, for different possible worlds, different nodes will be active; it is necessary to search for the active justifying nodes in a particular possible world. The actual implementation of this is much faster: each node is labeled with the different sets of assumptions that justify it–thus, explanation is basically a single look-up operation. Explanation is important for recognizing plan inferences and explaining predictions.

The input to the initialized system consists of a list of forms that are asserted sequentially. The forms represent the surface syntactic meaning of sequential utterances in observed conversations. The system input forms have no variables.

22 Example Application: Representation of Illocutionary Force

Illocutionary force requires assumptions because the speaker could possibly mean any one of several different things when an utterance is stated.[7] Understanding the illocutionary force behind an utterance consists of recognizing that a particular illocutionary act has taken place. This is done in the system by assuming the meaning behind an utterance; the conjuction of the assumed meaning plus the actual utterance act implies that the illocutionary act has occurred.

For example, take the utterance "Can I write down your name?". This could have three possible meanings: it could be a simple question concerning ability (the most literal interpretation); it could be a request for permission; or, it could be an indirect question for the information. Assuming the first meaning is true, the person has just performed an Ability-Question illocutionary act. The conjunction of the second meaning and the utterance act implies the occurrence of a Request-For-Permission act. Finally, if the third possible meaning is believed, in conjunction with the actual utterance it implies belief in the possibility that an Indirect-Question-Act has occurred.[8]

Once these assumptions have been explicitly represented, the system can use them as justifications, retract inconsistent assumptions, and work with them in other ways.

23 Recognition of Plan Inferences.

It is necessary for the system to recognize when a possible or actual action matches a prediction of the same action. This is most easily done using the explanation facility of the ATMS. The system builds a single special node, "PLAN-INFERRED", with an interrupt routine attached to it. Every time the system creates an action belief node, it also creates a predicted-action belief node for that action, and a third, special "interested in this action" assumption. These three nodes are set to imply the PLAN-INFERRED node. When both the possible action and the predicted action are simultaneously believed, the PLAN-INFERRED node becomes believed as well (since the "interested" node is also believed). This triggers the interrupt routine, which uses the explanation facility to find out *why* it is believed, out of the hundreds of implications pointing to it. The routine then reports the plan inference match, and the resulting plan. Finally, since this match has been reported, it is no longer of interest; the routine sets the "interested" node to nogood, which disables belief in the PLAN-INFERRED node and re-arms the interrupt.

[7] In addition, sometimes utterances can purposefully have more than one meaning. The system can represent both mutually exclusive interpretations and dual interpretations. However, it must know hypothetically which are which ahead of time.

[8] This example has been simplified for illustration purposes. For example, the mutually inconsistent interpretation constraints and the preconditions have been left out.

24 Review of Theory–Types of Knowledge

This section of the manual briefly digresses into an review of different theoretical types of knowledge. This is important in understanding the application of NP and the ATMS to actual problems. More about this theory has been said elsewhere; however, the terms are important enough to the NP system that this brief review is offered here.

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

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

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

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

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

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

Actual concepts can also combine with hypothetical rules to produce further actual concepts. Again, using the previous hypothetical example, the actual concept "An answer is expected" is produced (more formally, "It is actually true that right now an answer is expected"). [9]

The ATMS Representation. The ATMS represents the different kinds of knowledge in different ways. Hypothetical knowledge is represented by the ATMS-nodes. But, unless there is a reason to BELIEVE the knowledge, it remains hypothetical, and is not used by

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

the system. Possible knowledge is represented by the BELIEVED/NOT BELIEVED paradigm. If a node is an assumption, then it represents possible knowledge; ATMS-nodes that are implied by assumptions and therefore also come to be BELIEVED in some contexts also represent possible knowledge. Actual knowledge is represented by premises, which are always believed in all possible worlds.

25 Conclusion.

As a first step towards an integrated understanding system, this paper has presented **NP**, a plan-inference system, and **NFL**, a rule-based inference engine. Both systems use feature structures as their basic representation method, allowing direct interface with a parser, a transfer module, and a language generation module. Both systems are based on the use of assumptions. This allows NP to accept uncertain and multiple possible inputs, and to represent multiple possible inferred plans. Multiple possible input capability is important for disambiguating speech recognition results. The current version of the system is not yet able to support nonmonotonic actions nor reasoning in time. The system uses preinstantiated hypothetically-known inferences to save run-time processing. The NP system successfully recognizes feature-structure plans in expected parser output from actual dialogs in the ATR corpus.

A Implementation of NFL

To review, there are two main user data-structures employed by NFL: the nfl-fact, and the nfl-rule. Nfl-rules are divided into antecedent patterns, consequent patterns, and the special, optional effect-consequent patterns. All patterns have variables. When the conjunction of a rule's antecedent patterns consistently match assorted facts, the rule's consequent and effect-consequent patterns are instantiated.

NFL is an inference engine that uses an unordered list of "facts" and a set of "rules" to draw conclusions and find new assertions. In this aspect, NFL is quite similar to an ordinary inference engine; it has many features similar to a standard inference engine. There is a stack, which contains instantiations of rule firings to be examined. The top instantiation on the stack is examined for consistency; if its antecedent pattern instantiations are consistent, then the consequents are asserted. The rules' antecedents consist of a conjunction of patterns. However, NFL differs in three important aspects:

- Instead of logical forms, feature structures are used to represent both facts and patterns.
- When a rule fires, that firing is customarily asserted into the ATMS to be remembered. This is done by asserting the particular instantiated patterns of the rule (including the antecedent, consequent, and effect consequent patterns) into the ATMS, along with appropriate implications.
- The user has the option of disabling propagation inside NFL of the derived consequents of a rule. In other words, it is possible for the new resulting facts not to get inserted as new NFL-facts, but only to get used by the ATMS system.

These features are reflected in the implementation of the NFL system.

The actual data structures used by the system are slightly different. There is a structure for an nfl-rule, and a structure for an nfl-pattern; facts are simply raw feature-structures, and so do not need an explicit separate data-structure. Both the nfl-rules and the nfl-patterns are stored in respective hash-tables. There currently is no hash-table for facts, as all of the used facts are stored in the ATMS. There is also a processing stack that gets clocked.

The nfl-rule structure consists of a print-function, a list of antecedent nfl-patterns, a list of consequent feature-structure patterns, a list of effect-consequent feature-structure patterns, a documentation string, and a priority. The rule priority is used to sort the rule onto the system execution heap. Currently, all of the rules have the same priority, and there is no explicit provision for setting this priority. (Priorities are not especially significant in a monotonic system.) However, the priority-based stack insertion has been implemented, and if the user were to set rule priorities by hand (e.g. before any facts were asserted), this would work properly.

The nfl-pattern structure consists of a print-function, a data slot (used for documentation), a list of dotted-lists of matching node and bindings pairs (used for consistency checks), and a list of rules the pattern belongs to.

The current system assumes that the user first resets the NFL system, next enters all of the rules required for the system, and finally starts entering facts. In the current version, no provision has been made for remembering facts, and then checking a newly entered rule against all previously known facts. This restriction seems to be reasonable for a plan inference system. Currently, there is no penalty for adding rules after some facts have been added; the rule simply does not examine these. It would not be difficult to add the ability to add new rules if this were required.

Resetting the system clears out the nfl-rules and nfl-patterns hash-tables, resets the stack, and clears the counters. It also clears out the RWS system, which could cause problems if another system is using RWS.

The user starts by specifying a series of rules to the NFL system. Each rule gets entered into an nfl-rule data structure, and filed in the hash-table (currently under its ID number). The antecedents get entered as a list of nfl-patterns.

When a new nfl-pattern is created, first the pattern should be checked to see whether it is isomorphic with a previous pattern. If it is, the old pattern would be used instead, for efficiency. Since doing this check correctly involves theoretical problems with normalizing variable names, currently this nonessential efficiency check is bypassed. Next, the pattern object gets created, initialized with the feature-structure data, and filed in the nfl-pattern hash-table (again, currently under its ID number). The pattern's rule is pushed onto the pattern's rule list. Finally, the pattern, along with its ID number and its name-tag, gets specially asserted into the RWS system.

The RWS assertion, used to interface with the RWS system, is perhaps the most difficult part of the system. The NFL system does not use the normal RWS utilities, rather it uses a special custom routine to assert RWS rules. The routine dynamically builds a RWS rule at run-time, using macros. It is important to get the macro-expansion evaluation level correct when the software gets changed–make sure that the actual macro assertions are evaluated at run-time, and not the customary compile-time. Note that there is a difference between compiled and interpreted code on this point; errorful code will run correctly while interpreted. The current system does work when the code is compiled. ...Normal RWS rules consist of an antecedent feature structure, and a consequent routine that instantiates and returns an appropriate consequent to the RWS routine when that RWS rule fires. The NFL RWS routines instead accept the bindings list from the recognized antecedent, push it on a special NFL answer stack as a side-effect, and return a null answer to RWS. This has the desirable effect of bypassing any RWS-system-specific requirements for instantiated answers. The bindings list has an objectionable "?input" variable automatically inserted on it, so this is removed before the bindings are returned to NFL. Since the RWS system changes, this particular interface procedure has to be adjusted every time a new, modified version of RWS comes out.

After the user has finished asserting rules into the NFL system, the user next asserts facts one by one to the system, in the form of a specified feature structure. The fact is first printed out, if the `*nfl-debug*` flag is on. Next, if the `*nfl-dont-repropagate*` flag is on, the feature structure is checked against the ATMS node table, using `find-node`. In this case, if the fact has been entered into the ATMS already, the rest of the process is disregarded. Otherwise, the feature structure is submitted to the RWS for pattern-matching, and the results (in the form of a list of bindings/pattern-ID dotted lists) are

returned to the NFL system. The pattern IDs are used to look up the matching nfl-patterns, and the bindings are pushed onto the pattern's node.bindings slot. All of the rules in the pattern's rule slot are reactivated. After this, currently the activated rules stack is clocked by the `nfl-fact` routine until it's empty. If a more user-oriented system is desired, this feature should be removed and the user should be allowed to clock the stack as desired.

Submission to the RWS system consists of running the submitted fact feature-structure against the pattern-recognition rules that were submitted into the RWS by NFL already. A special global interface variable is used to keep and return a stack of all of the results from RWS to NFL. If no rules match, this variable is NIL.

Activating a rule consists of forming all of the permutations between the one pattern instantiation that activated the rule, and all of the other binding sets associated with the other antecedent patterns. A set of one bindings from each pattern consists of an instantiation. The rule instantiations are sorted into the processing stack to be checked, based on the rule's priority.

Clocking the stack once consists of checking (only) the single instantiation on the top of the stack (whether it turns out to be consistent or not), and then firing the rule if it's consistent. In any case the stack is popped, and the instantiation is discarded.

Checking an instantiation on top of the stack is currently done by throwing the bindings in a "comparison pot", one by one. First, a variable is tested against the pot by performing an `assoc`. If the variable was not in the pot already, it is put in the pot, along with its binding. If the variable *was* in the pot, the current variable's binding is tested against the pot variable's binding, to see whether they are equivalent or not. Since the bindings are (cyclic) feature structures, there are some philosophical questions as to what constitutes equivalency. Currently `rws:FS-equal` is used for this test, which is not as strict as eq. It is unknown whether variable names are (or should be) significant in this test. If the bindings are equivalent, no action is performed, as the variable is in the pot already. If the bindings are not equivalent, the algorithm signals an inconsistency rejection, and terminates. If the algorithm manages to check all of the bindings without returning an inconsistency, the algorithm completes successfully and signals consistency. This algorithm works well, but is rather brute-force and conceivably could be improved for speed by replacement with some more clever algorithm.

When a consistent rule fires, both all of the consequences' and all of the effect consequences' feature structures are instantiated using a *copy* of the variable bindings, and each resulting feature structure is asserted into the ATMS as a hypothetical node. If the flag `*nfl-propagate*` is true, each of these feature structures is asserted, as it is created, as an nfl-fact. Of course, when an nfl-fact is asserted, it could fire off more rules and clock the stack. Note that this depth-first instantiation method could have important implications if the system is used for nonmonotonic applications in the future; this might have to be trivially replaced with a breadth-first instantiation method. In addition to the system creating ATMS-nodes for the rule's antecedents and consequents, implications are created from the conjunction of the rule's antecedent nodes to each of the rule's consequent nodes. Also, if there are any effect consequents, implications are created from the last consequent to each of the effect-consequent nodes. This is useful for implementing action networks.

Currently there are no functions for supporting retraction of facts. Thus, the NFL system is currently monotonic. It would be possible to write routines in NFL that would

retract NFL facts, however there is a question as to what should be done with the ATMS nodes and/or the corresponding implications. A node cannot be retracted when it is hypothetical, it can only be deleted (to a NULL belief status). Although this could be done, it is unclear whether it would be desirable or not. Perhaps the best thing would be to retract all justified (outgoing) implications from a retracted node, or to build a system centered around retracting rules rather than nodes. It might also be useful to re-examine the philosophy of hypothetical reasoning versus actual reasoning.

Currently, the current system only implements hypothetical reasoning. It is then left to the user system to assert actual or possible concepts directly into the ATMS. This is a clean breakdown, and results in a useful and understandable system.

However, NFL maintains only one pool of hypothetical facts. One consequence of this is that NFL explores only one possible universe, in effect exploring multiple possible worlds at the same time but not keeping the distinction between them. If there are no implications in the ATMS that imply the nogood node this is not a problem. Otherwise, the NFL system might waste some time exploring combinations of facts that belong to disjoint possible worlds, i.e. that are inconsistent.

Places for possible future improvement include: reworking the pattern matcher (either RWS or some other system) so that it matches multiple facts and patterns concurrently, using trees for representation; put in the test for pattern repeats in multiple rules; create and implement a better consistency checker; and, put in the new-rules-against-facts comparison. Eventually, the rules should also have locally compiled programs to represent the consequents, instead of being interpreted. There is also no support for a backward-chaining inference engine.

The results of the system are that all consistent rule firings are instantiated as implication networks in the ATMS. In addition, if the flag `*nfl-propagate*` is on, the system repropagates the results of the rule firings back through the NFL system. The resulting system offers an implementation of an inference engine based on feature structures.

B Implementation of the ATMS

As a brief review, from the user's viewpoint, there are three kinds of nodes: ATMS-nodes, premises, and assumptions. There is also one kind of connection between nodes, the implication (or, "justification"). Finally, there are the environments, which consist of sets of assumptions. Use of the system consists of creating nodes, and then creating implications to link them together. The user can also indicate inconsistent nodes or sets of nodes. Environments can then be referenced, to see what assumptions are required in a particular possible world, and which possible worlds are inconsistent. This section provides a brief review; more information can be found in the ATMS manual [Mye89b].

B.1 Implementation Data Structures

The actual data-structures that are used to accomplish this are somewhat different from the user conceptualization. There are four types of structures in the implementation: the ATMS-node, the assumption-tag, the implication, and the environment. Premises are implemented

as a special case of the ATMS-node. An assumption is implemented as an ATMS-node together with an assumption-tag, with a single-antecedent implication pointing to the ATMS-node from the assumption-tag. All of these objects are implemented as structures for speed.

B.1.1 ATMS-node structure

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

B.1.2 The Assumption-Tag Structure

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

B.1.3 The Implication Structure

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

B.1.4 The Environment Structure

An environment has the fields *nodes, nogood-p, size, ID,* and *assum-bits.* In addition, it has an associated print-function. The nodes field contains the context of the environment, i.e. a list of all of the nodes that have this environment in their label. The nogood-p field is nil unless the environment is nogood; this provides a quick check, although it is

not strictly necessary. The size field provides a count as to the number of assumptions in this environment; it is used to order the environment in lists. The ID field contains a non-negative integer to identify the environment, that is unique among the environments. And, the assum-bits field contains a special bit-array that has a bit set for the number of each assumption that composes the environment.

B.2 Firing Processing Demons

ATMS processing demons are implemented using a user-specified routine that is stored in the appropriate ATMS node, and a special check in the `OR-label` routine that gets called when a node's justifying implication is reprocessed. If the node turns from OUT to IN, the stored routine is `eval`'ed.

B.3 Efficiency Considerations

Data structures are implemented with Lisp structures, instead of flavor objects. This results in faster access time. Some previous ATMSs have based their propagation on nodes, requiring a node to recompute its label from its justifications and their antecedent nodes when a change is propagated. This involves unnecessary computation. The ATR ATMS bases propagation on implications, which is faster. As explained above, the propagated change contributed by an implication is unioned into the label of the implication's consequent; there is no need to examine the sister implications contributing to the consequent. This results in significant time savings (around 10x in one benchmark) when one node is justified by many different implications. Some previous ATMSs have represented their labels using lists, which require list computations. This ATMS uses bit vectors to represent labels; as a result, label computations are extremely fast. In particular, the important subsumption test is represented as two accesses and a single bit-vector operation, resulting in extremely efficient operation on the Symbolics Lisp Machine.

Efficiency questions also center around the porting of the ATMS to other machines, such as the SUN. A previous version of the ATMS used extensible bit-vectors, which, although fast on the Symbolics, are extremely slow on the SUN. The current system uses static bit-vectors that get copied. In addition, the environment bins stored in assoc lists under `*env-bins*` and `*nogood-bins*` that were previously themselves implemented as assoc lists, have been reimplemented using hash tables. This also resulted in a 2.3x speedup in benchmarks for very large user systems.

C Version History of NP

Version 1.0 was based on logical forms, and used the FLAIL inference engine. Instantiation was top-down, which meant that no new information could be instantiated in the lower levels of the hierarchy–the action descriptions had to contain enough variables and constants to completely support the decompositions. This was a disadvantage. The NP actions were monotonic.

Version 2.0 was a complete rewrite that converted the NP system from using logical forms to the use of feature structures. Instead of using an inference engine, the system used the rewriting facilities of RWS along with a plan-schema pre-interpreter, a set of *instruction rules*, and an instruction post-interpreter. The pre-interpreter built up a set of instructions and put them in the single consequent of an RWS rule. When the rule executed, the instructions were instantiated and returned. The post-interpreter interpreted these instructions and built a corresponding ATMS network. Although this method worked, it was clumsy. Instantiation was performed bottom-up, which had the advantage that information could be discarded when going from the low levels of the hierarchy up to the high levels. However, instantiation was performed with a set of instructions–recognizing the action from the decompositions was performed separately from implying the effects from the action. This meant that the action description had to contain enough information to instantiate the effects, a disadvantage. In addition, because the decompositions were necessarily segmented into separate rewriting rules, two different decompositions to the same action might unnecessarily reinstantiate the identical implications network, resulting in duplicate implications.

Version 3.0 was another rewrite that introduced NFL and phased out most of the use of the RWS. Instead of all of the RWS being used for recognizing patterns, rewriting the consequents, and returning the results as an instruction rule to be interpreted, only part of the RWS was used, just for recognizing patterns for NFL. The NFL inference engine instantiated actions bottom-up. Since all of the decompositions were present in the rule, the information required for the effects could be derived from all of the decompositions together, and it was no longer necessary to include deriving information in the action description. The system was invoked in a single pass using a complex series of six arguments. Multiple alternatives were not explicitly supported. Three levels of verbosity were added to the graphics display: small, medium, and large.

Version 3.1 cleaned up the user interface by introducing the "-utt" user commands. The system was invoked incrementally, in an interactive fashion, reflecting a more realistic method of use. In addition, the "alternative-utt" command was introduced to explicitly support possible alternative inputs. The graphics was converted to bold-outline actual assertions, instead of reversing them as white-on-black.

Version 3.2 cleaned up some other minor items. Assumptions can now accept probabilities, although they are not used by the system.

Version 3.3 saw NP installed as a Lisp System. The NP system can now be invoked with the Load System NP command. The "pairwise-inconsistent" functions were added for the ATMS.

D Example Listing of Plan Input

```
;;; -*- Syntax: Common-Lisp; Base: 10; Mode: TFS -*-

      ;;;PLANS 4      File LM01:>myers>np1-plans4-CAN.lisp




;;;
;;;   "CAN" ABILITY/POSSIBILITY
;;;




;;  ?variables
;;  @coref-tag[DEF], @coref-tag.

;;;        HISTORY
;        March 9 '90      Some variables had a question-mark in the MIDDLE.
;                  The reader was choking on them.  Don't use question-marks
;                  inside variable/constant names.
;                  "Meet extra feature value"---no [[ after the feature, only [.
;
;

(make-FS-action

  [[action  +/Q/1/DEKIRU]   ;Can (I) do X? and X is a commissive, means "X".

   [prec1        [[RELN  Is-a]
                  [ARG1  ?verb]
                  [ARG2        Commissive]] ]

   [dèc1         [[RELN  S-REQUEST]
                  [AGEN ?questioner]
                  [RECP ?answerer]
                  [OBJE [[RELN  INFORMIF]
                         [AGEN ?answerer]
                         [RECP ?questioner]
                         [OBJE [[RELN  できる-POSSIBLE]
                                [AGEN ?questioner]
                                [OBJE [[RELN ?verb]
                                       ?rest]]]]]]]]

   [eff1         [[RELN ?verb]
```

```
                ?rest]]

  ]
)


(make-FS-action

  [[action  +/Q/1/DEKIRU]        ;Can I X?

   [dec1          [[RELN  S-REQUEST]
                   [AGEN ?questioner]
                   [RECP ?answerer]
                   [OBJE [[RELN  INFORMIF]
                          [AGEN ?answerer]
                          [RECP ?questioner]
                          [OBJE [[RELN  できる-POSSIBLE]
                                 [AGEN ?questioner]
                                 [OBJE ?action]]]]]]]

  ]
)


(make-FS-action

  [[action  +/Q/2/DEKIRU]        ;Can you X?

   [dec1          [[RELN  S-REQUEST]
                  [AGEN ?questioner]
                  [RECP ?answerer]
                  [OBJE [[RELN  INFORMIF]
                         [AGEN ?answerer]
                         [RECP ?questioner]
                         [OBJE [[RELN  できる-POSSIBLE]
                                [AGEN ?answerer]
                                [OBJE ?action]]]]]]]

  ]
)


(make-FS-action

  [[action  +/Q/3/DEKIRU]        ;Can he/she X?

   [prec1         [[RELN  DIFFERENT]
```

```
                [ARG-1 ?answerer]
                [ARG-2 ?third-person]]]

   [prec2       [[RELN  DIFFERENT]
                [ARG-1 ?questioner]
                [ARG-2 ?third-person]]]

   [dec1        [[RELN  S-REQUEST]
                [AGEN ?questioner]
                [RECP ?answerer]
                [OBJE [[RELN  INFORMIF]
                       [AGEN ?answerer]
                       [RECP ?questioner]
                       [OBJE [[RELN  できる-POSSIBLE]
                              [AGEN ?third-person]
                              [OBJE ?action]]]]]]]

  ]
)


(make-FS-action

  [[action  +/Q/U/DEKIRU]        ;Can [0 someone] X?

   [dec10       [[RELN  S-REQUEST]
                [AGEN ?questioner]
                [RECP ?answerer]
                [OBJE [[RELN  INFORMIF]
                       [AGEN ?answerer]
                       [RECP ?questioner]
                       [OBJE [[RELN  できる-POSSIBLE]
                             ;This may need a ? or a !.
                              [AGEN @agent[]]
                              [OBJE [[RELN ?verb]
                                     [AGEN @agent]
                                     ?rest]]]]]]]]


   [eff1        [[RELN S-REQUEST]        ;Please X.
                [AGEN ?questioner]
                [RECP ?answerer]
                [OBJE [[RELN ?verb]
                       [AGEN ?answerer]
                       ?rest]]]]

  [eff2                [[RELN DESIRE]           ;I want [0 someone] to X.
```

```
                [AGEN ?questioner]
                [OBJE [[RELN ?verb]
                       [AGEN ?agent]
                       ?rest]]]]

  ]
)


(make-FS-action

  [[action  -/Q/1/DEKIRU]        ;Can't I X?

   [dec1          [[RELN  S-REQUEST]
                  [AGEN ?questioner]
                  [RECP ?answerer]
                  [OBJE [[RELN  INFORMIF]
                         [AGEN ?answerer]
                         [RECP ?questioner]
                         [OBJE [[RELN        NEGATE]
                                [OBJE [[RELN  できる -POSSIBLE]
                                       [AGEN ?questioner]
                                       [OBJE ?action]]]]]]]]]

  ]
)


(make-FS-action

  [[action  -/Q/2/DEKIRU]        ;Can't you X?

   [dec1          [[RELN  S-REQUEST]
                  [AGEN ?questioner]
                  [RECP ?answerer]
                  [OBJE [[RELN  INFORMIF]
                         [AGEN ?answerer]
                         [RECP ?questioner]
                         [OBJE [[RELN        NEGATE]
                                [OBJE [[RELN  できる -POSSIBLE]
                                       [AGEN ?answerer]
                                       [OBJE ?action]]]]]]]]]

  ]
)
```

```
(make-FS-action

  [[action  -/Q/3/DEKIRU]        ;Can't he/she X?

   [prec1         [[RELN  DIFFERENT]
                  [ARG-1 ?answerer]
                  [ARG-2 ?third-person]]]

   [prec2         [[RELN  DIFFERENT]
                  [ARG-1 ?questioner]
                  [ARG-2 ?third-person]]]

   [dec1          [[RELN  S-REQUEST]
                  [AGEN ?questioner]
                  [RECP ?answerer]
                  [OBJE [[RELN  INFORMIF]
                         [AGEN ?answerer]
                         [RECP ?questioner]
                         [OBJE [[RELN        NEGATE]
                                [OBJE [[RELN  できる-POSSIBLE]
                                       [AGEN ?third-person]
                                       [OBJE ?action]]]]]]]]]

  ]
)


(make-FS-action

  [[action  -/Q/U/DEKIRU]        ;Can't [0 someone] X?

   [dec1          [[RELN  S-REQUEST]
                  [AGEN ?questioner]
                  [RECP ?answerer]
                  [OBJE [[RELN  INFORMIF]
                         [AGEN ?answerer]
                         [RECP ?questioner]
                         [OBJE [[RELN        NEGATE]
                                [OBJE [[RELN  できる-POSSIBLE]
                                       ;This may need a ? or a !.
                                       [AGEN @agent[]]
                                       [OBJE ?action]]]]]]]]]

  ]
)
```

```
(make-FS-action

  [[action  +/./1/DEKIRU]        ;I can X.

   [dec1          [[RELN  できる-POSSIBLE]
                        [AGEN ?questioner]
                        [OBJE ?action]]  ]

  ]
)


(make-FS-action

  [[action  +/./2/DEKIRU]        ;You can X.

               ;Unknown error here--unfinished?
   [dec1                  [[RELN  できる-POSSIBLE]
                               [AGEN ?answerer]
                          .    [OBJE ?action]]]

  ]
)


(make-FS-action

  [[action  +/./3/DEKIRU]        ;He/She can X.

   [prec1         [[RELN  DIFFERENT]
                  [ARG-1 ?answerer]
                  [ARG-2 ?third-person]]]

   [prec2         [[RELN  DIFFERENT]
                  [ARG-1 ?questioner]
                  [ARG-2 ?third-person]]]

   [dec1         [[RELN  できる-POSSIBLE]
                       [AGEN ?third-person]
                  [OBJE ?action]] ]

  ]
)


(make-FS-action
```

```
  [[action  +/./U/DEKIRU]        ;[0 someone] can X.

   [dec1        [[RELN  できる-POSSIBLE]
                 ;This may need a . or a !.
                 [AGEN @agent[]]
                 [OBJE ?action]
                ?rest]]

  ]
)


(make-FS-action

  [[action  -/./1/DEKIRU]        ;I can't X.

   [dec1        [[RELN        NEGATE]
                 [OBJE [[RELN  できる-POSSIBLE]
                        [AGEN ?questioner]
                        [OBJE ?action]
                        ?rest]]]]

  ]
)


(make-FS-action

  [[action  -/./2/DEKIRU]        ;You can't X.

   [dec1        [[RELN        NEGATE]
                 [OBJE [[RELN  できる-POSSIBLE]
                        [AGEN ?answerer]
                        [OBJE ?action]
                        ?rest]]]]

  ]
)


(make-FS-action

  [[action  -/./3/DEKIRU]        ;He/She can't X.

   [prec1       [[RELN  DIFFERENT]
                 [ARG-1 ?answerer]
```

```
                [ARG-2 ?third-person]]]

   [prec2       [[RELN  DIFFERENT]
                [ARG-1 ?questioner]
                [ARG-2 ?third-person]]]

   [dec1       [[RELN        NEGATE]
                [OBJE [[RELN  できる -POSSIBLE]
                       [AGEN ?third-person]
                       [OBJE ?action]
                       ?rest]]]]
  ]
)


(make-FS-action

  [[action  -/./U/DEKIRU]        ;[0 someone] can't X.

   [dec1       [[RELN        NEGATE]
                [OBJE [[RELN  できる -POSSIBLE]
                       ;This may need a . or a !.
                       [AGEN @agent[]]
                       [OBJE ?action]
                       ?rest]]]]
  ]
)
```

E Example of a Conversation that is Input to the Program

```
;;; -*- Base: 10; Syntax: Common-Lisp; Mode: TFS -*-
;;;
;;; CONV5-EX     LM01:>NP>example-conversations>conv5-ex.lisp
;;;
;;;
;;;   Expected output results for conversation 5
;;;
;;;  HISTORY:
;;;  Nov 21 '89  Changed all ?'s to !'s for Hasegawa.
;;;  Bug; changed all ++u's to ++ u.  Changed read-fs to rws:read-fs.
;;;  Switched over to NP-input, reset-NP-input.  5 errors in input file.
;;;  "meet extra feature value"  == you left out a [[ square bracket
;;;  after this slot, or other [] mismatch.
;;;  "Can't redefine tag" == Two !X[] !X[]'s encountered.
;;;        Take square brackets off second one.
;;;  "illegal tag definition" == you forgot the [] after the !X the first time.
;;;  Converted SPEAKER and HEARER to OFFICE and GUEST.
;;;  Let's be consistent!  POSSIBLE should always take AGEN.
;;;        Changed EXPR to AGEN in three places.
;;;  Nov 22 '90   Copied over to NP:>example-conversations.



;-----------------------------------------------------------
(NP-input "はい"

[[RELN  はい-AFFIRMATIVE]
 [AGEN [[LABEL  *OFFICE*]]]
 [RECP [[LABEL  *GUEST*]]]]

)
;-----------------------------------------------------------
(NP-input "こちらは会議事務局でございます"

[[RELN  だ-IDENTICAL]
 [IDEN [[PARM !X01[]]
        [RESTR [[RELN  NAMED]
                [ENTITY !X01]
                [IDEN  会議事務局-1]]]]]
 [OBJE [[LABEL  *OFFICE*]]]
 [HEAR [[LABEL  *GUEST*]]]]
)
;-----------------------------------------------------------
```

```
(NP-input "ちょっとお願いがあるのですが"

[[RELN  が-MODERATE]
 [OBJE [[RELN  ある-1]
        [OBJE [[PARM !X01[]]
               [RESTR [[RELN  お願い-1]
       [AGEN []]
       [RECP []]
       [OBJE []]
                        [ENTITY !X01]]]]]
[INFMANN [[PARM !X02[]]
  [RESTR [[RELN  ちょっと-1]
  [ENTITY !X02]]]]]]]]

)
;-------------------------------------------------------------------------
(NP-input "私は会議に申込みをした者です"

[[RELN  だ-IDENTICAL]
 [OBJE [[LABEL  *GUEST*]]]
 [IDEN [[PARM !X04[[PARM !X01[]]
                   [RESTR [[RELN  者-1]
                           [ENTITY !X01]]]]]
        [RESTR [[RELN  た-PERFECTIVE]
                [OBJE [[RELN  する-2]
                       [AGEN !X04]
                       [SLOC [[PARM !X02[]]
                              [RESTR [[RELN  会議-1]
                                      [ENTITY !X02]]]]]
                       [OBJE [[PARM !X03[]]
                              [RESTR [[RELN 申込み-1]
      [AGEN []]
      [OBJE []]
      [SLOC []]
                                          [ENTITY !X03]]]]]]]]]]]]]]]

)
;-------------------------------------------------------------------------
(NP-input "参加を取り消したいのですが"

[[RELN  が-MODERATE]
 [OBJE [[RELN  たい-DESIRE] ;Weird IDEN taken out.
[EXPR !X02[]]
[OBJE [[RELN  取り消す-1]
       [AGEN !X02]
       [OBJE [[PARM !X01[]]
      [RESTR [[RELN  参加-1]
```

```
      [AGEN []]
      [SLOC []]
      [ENTITY !X01]]]]]]]]]]

)
;-------------------------------------------------------------------------
(NP-input "お名前をお伺いできますでしょうか"

[[RELN  S-REQUEST]
 [AGEN !X01[[LABEL  *OFFICE*]]]
 [RECP !X02[[LABEL  *GUEST*]]]
 [OBJE [[RELN  INFORMIF]
        [AGEN !X02]
        [RECP !X01]
        [OBJE [[RELN  できる-POSSIBLE]
       [AGEN !X04[]] ;Changed EXPR to AGEN
       [OBJE [[RELN  聞く-1]
      [AGEN !X04]
      [RECP []]
      [OBJE [[PARM !X03[]]
     [RESTR [[RELN  名前-1]
     [ENTITY !X03]]]]]]]]]]]

)
;-------------------------------------------------------------------------
(NP-input "はい"

[[RELN  はい-AFFIRMATIVE]
 [AGEN [[LABEL  *GUEST*]]]
 [RECP [[LABEL  *OFFICE*]]]]

)
;-------------------------------------------------------------------------
(NP-input "ベル研のジムワイベルです"

[[RELN  だ-IDENTICAL]
 [IDEN [[PARM !X03[[PARM !X01[]]
                  [RESTR [[RELN  NAMED]
                          [ENTITY !X01]
                          [IDEN ジムワイベル]]]]]
        [RESTR [[RELN  の-連体修飾]
                [ARG-1 [[PARM !X02[]]
                        [RESTR [[RELN  NAMED]
                                [ENTITY !X02]
                                [IDEN  ベル研-1]]]]]
                [ARG-2 !X03]]]]]
 [OBJE []]]
```

```
)
;-----------------------------------------------------------
(NP-input "既に登録料の８万５千円を振り込まれておられますね"

[[RELN  ね-CONFIRMATION]
 [OBJE [[RELN  ている-PROGRESSIVE]
[AGEN !X01[]]
        [OBJE [[RELN れる-RESPECT]
       [OBJE [[RELN  振り込む-1]
      [AGEN !X01]
      [SLOC []]
      [OBJE [[PARM !X04[[PARM !X02[]]
[RESTR [[RELN  ８万５千円-1]
[ENTITY !X02]]]]]
     [RESTR [[RELN  の-連体修飾]
     [ARG-1 [[PARM !X03[]]
     [RESTR [[RELN  登録料-1]
     [ENTITY !X03]]]]]
     [ARG-2 !X04]]]]]]]
       [TLOC [[PARM !X05[]]
      [RESTR [[RELN  既に-1]
      [ENTITY !X05]]]]]]]]]]

)
;-------------------------------------------------------------
(NP-input "はい"

[[RELN  はい-AFFIRMATIVE]
 [AGEN [[LABEL  *GUEST*]]]
 [RECP [[LABEL  *OFFICE*]]]]

)
;---------------------------------------------------------------
(NP-input "そうです"

[[RELN  そうです-CONFIRMATION]
 [AGEN [[LABEL  *GUEST*]]]
 [RECP [[LABEL  *OFFICE*]]]]

)
;-----------------------------------------------------
(NP-input "参加料を払い戻して頂けますか"

[[RELN  S-REQUEST]
 [AGEN !X04[[LABEL  *GUEST*]]]
 [RECP !X05[[LABEL  *OFFICE*]]]
```

```
 [OBJE [[RELN  INFORMIF]
        [AGEN !X05]
        [RECP !X04]
        [OBJE [[RELN  もらえる-POSSIBLE]
               [AGEN !X03[]]
               [OBJE [[RELN  もらえる-RECEIVE_FAVOR]
                      [AGEN !X03]
                      [RECP !X02[]]
                      [OBJE [[RELN  払い戻す-1]
                             [AGEN !X02]
                             [OBJE [[PARM !X01[]]
                                    [RESTR [[RELN  参加料-1]
                                            [ENTITY !X01]]]]]]]]]]]]]

)
;-----------------------------------------------------------------
(NP-input "お気の毒ですができません"

[[RELN  NEGATE]
 [OBJE [[RELN  できる-POSSIBLE]
        [OBJE []]
        [AGEN []]]]
 [INFMANN [[RELN が-DISJUNCT]
           [OBJE [[RELN だ-IDENTICAL]
                  [IDEN [[PARM !X01[]]
                         [RESTR [[RELN  気の毒-1]
                                 [ENTITY !X01]]]]]
                  [OBJE []]]]]]]

)
;-----------------------------------------------------------------
(NP-input "案内書にも書いていますが"

[[RELN  が-MODERATE]
 [OBJE [[RELN  ている-STATIVE]
        [AGEN !X02[]]
        [OBJE [[RELN  書く-1]
               [AGEN !X02]
               [SLOC [[PARM !X01[]]
     [RESTR [[RELN  案内書-1]
     [ENTITY !X01]]]]]
      [OBJE []]]]]]]

)
;------------------------------------------------------------
(NP-input "9月27日以後の取り消しに対する払い戻しできません"
```

```
[[RELN  NEGATE]
 [OBJE [[RELN  できる -POSSIBLE]
        [AGEN []] ;EXPR changed to AGEN
        [OBJE [[PARM !X05[[PARM !X04[]]
  [RESTR [[RELN  払い戻し -1]
  [AGEN []]
  [OBJE []]
  [ENTITY !X04]]]]]
       [RESTR [[RELN  対する -1]
       [AGEN !X05]
       [OBJE [[PARM !X03[]]
      [RESTR [[RELN  取り消し -1]
      [AGEN []]
      [OBJE []]
      [TLOC [[PARM !X02[]]
     [RESTR [[RELN  以後 -1]
     [ENTITY !X02]
     [COMP-OBJE [[PARM !X01[]]
 [RESTR [[RELN 9 月 27 日 -1]
 [ENTITY !X01]]]]]]]]]]   ;] moved here from next line
      [ENTITY !X03]]]]]]]]]]]]

)
;-------------------------------------------------------------------------------
(NP-input "後日プログラムと予稿集をお送り致します"

[[RELN  送る -1]
 [AGEN []]
 [RECP []]
 [TLOC [[PARM !X03[]]
        [RESTR [[RELN  後日 -1]
                [ENTITY !X03]]]]]
 [OBJE [[RELN  と -COORDINATE]
        [ARG-1 [[PARM !X01[]]
                [RESTR [[RELN  プログラム -1]
[AGEN []]
[OBJE []]
                       [ENTITY !X01]]]]]
        [ARG-2 [[PARM !X02[]]
                [RESTR [[RELN  予稿集 -1]
[ENTITY !X02]]]]]]]]

)
;-------------------------------------------------------------
(NP-input "では誰かが私の代わりに参加することはできますか"

[[RELN  S-REQUEST]
```

```
 [AGEN !X04[[LABEL  *GUEST*]]]
 [RECP !X06[[LABEL  *OFFICE*]]]
 [OBJE [[RELN  INFORMIF]
        [AGEN !X06]
        [RECP !X04]
        [OBJE [[RELN  できる-POSSIBLE]
               [AGEN [[PARM !X02[]] ;EXPR changed to AGEN
                      [RESTR [[RELN  誰か-1]
                              [ENTITY !X02]]]]]
               [OBJE [[RELN  参加する-1]
                      [AGEN !X02]
                      [SLOC []]
                      [PURP [[PARM !X05[[PARM !X03[]]
                                            [RESTR [[RELN  代わり-1]
[AGEN []]
                                                    [OBJE !X04]
[ENTITY !X03]]]]]
     [RESTR [[RELN  の-連体修飾]
     [ARG-1 !X04]
     [ARG-2 !X05]]]]]]]]]]
 [INFMANN [[PARM !X01[]]
           [RESTR [[RELN  では-1]
   [ENTITY !X01]]]]]]

)
;-----------------------------------------------------------
(NP-input "それは別に問題ありません"

[[RELN  NEGATE]
 [MANN [[RELN  別だ-1]
        [COMP []]
        [OBJE !X01[]]]] ;[] moved here from down under PARM.
 [OBJE [[RELN  問題ある-1]
        [SLOC [[PARM !X01]
               [RESTR [[RELN  それ-1]
                       [ENTITY !X01]]]]]]]]


)
;-----------------------------------------------------
(NP-input "代理人が参加する場合はあらかじめこちらまでお知らせ下さい"

[[RELN  下さい-REQUEST]
 [AGEN !X01[[LABEL  *OFFICE*]]]
 [RECP !X02[[LABEL  *GUEST*]]]
 [MANN [[PARM !X03[]]
        [RESTR [[RELN  あらかじめ-1]
```

```
                [ENTITY !X03]]]]]
 [OBJE [[RELN せる-CAUSATIVE] ;RELN Inserted
        [AGEN !X01]
        [RECP !X02]
        [OBJE [[RELN 知る-1]
               [AGEN !X02]
               [OBJE []]]]]]
 [COND [[PARM !X05[]]
        [RESTR [[RELN  場合-CONDITIONAL]
                [ENTITY !X05]
                [IDEN [[RELN  参加する-1]
                       [AGEN [[PARM !X04[]]
                              [RESTR [[RELN  代理人-1]
                                      [ENTITY !X04]]]]]
                       [SLOC []]]]]]]]

)
;-----------------------------------------------------
(NP-input "分かりました"

[[RELN  た-PERFECTIVE]
 [OBJE [[RELN  分かる-1]
        [EXPR []]
        [OBJE []]]]]

)
;-----------------------------------------------------
(NP-input "代理人が決まりましたらお知らせ致します"

[[RELN  せる-CAUSATIVE]
 [AGEN []]
 [RECP !X03[]]
 [OBJE [[RELN 知る-1]
        [AGEN !X03]
        [OBJE []]
        [COND [[PARM !X02[]]
               [RESTR [[RELN  たら-CONDITIONAL]
                       [ENTITY !X02]
                       [IDEN [[RELN  決まる-1]
                              [OBJE [[PARM !X01[]]
                                     [RESTR [[RELN  代理人-1]
                                             [ENTITY !X01]]]]]]]]]]]]]

)
;---------------------------------------------------------------------------
(NP-input "では失礼します"
```

```
[[RELN  失礼する -CLOSE_DIALOGUE]
 [AGEN [[LABEL  *GUEST*]]]
 [RECP [[LABEL  *OFFICE*]]]
 [INFMANN [[PARM !X01[]]
          [RESTR [[RELN  では -1]
                  [ENTITY !X01]]]]]]
```

)

F Example Output of the Program

This section shows an example of the results of running the plan recognition system on a slightly shorted version of conversation 4. Each of the white boxes is a concept that has been inferred. The black boxes are the system's predictions (prefaced by "P:"), which are not used in this example for clarity.

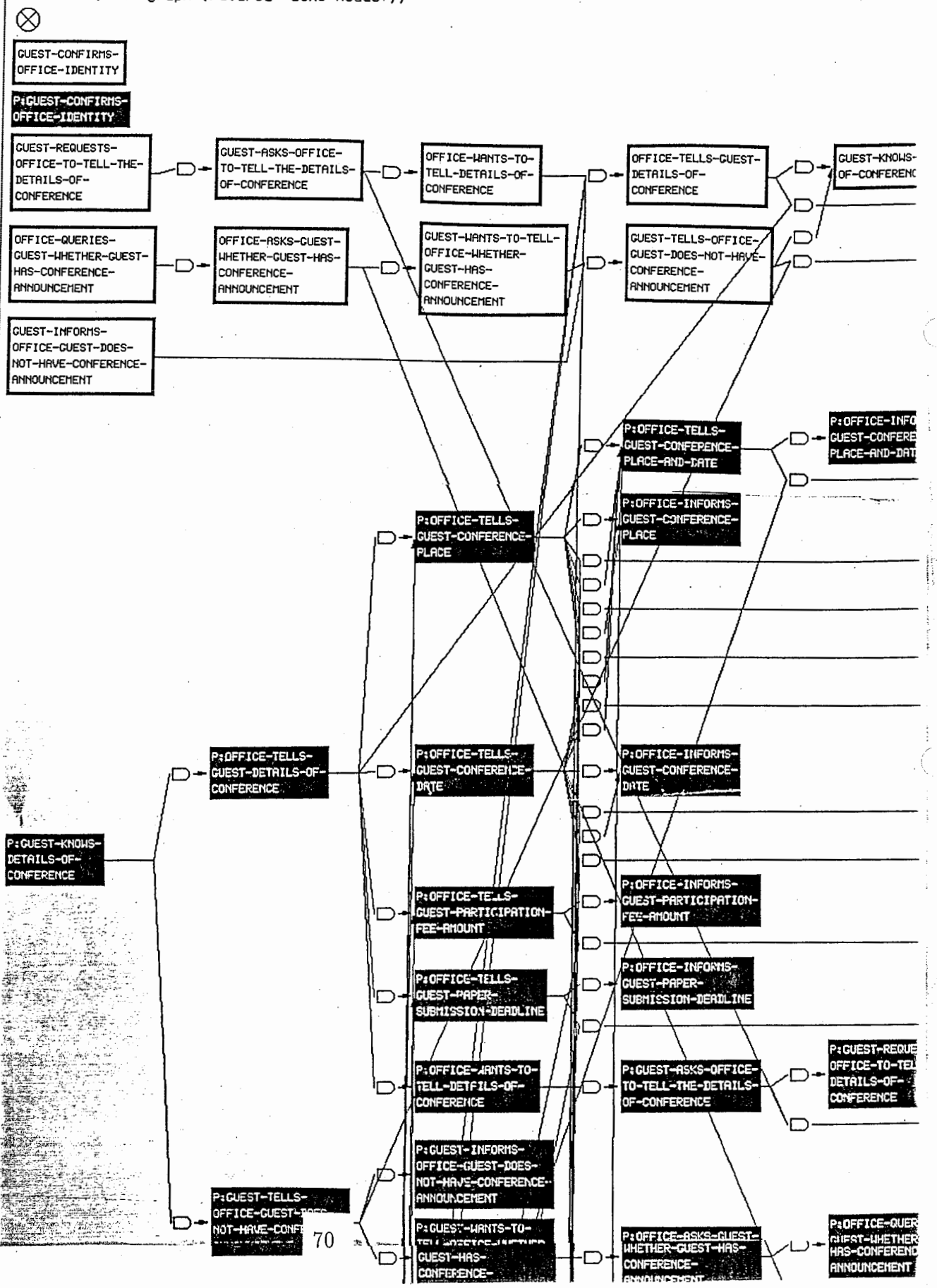
There are a number of observations that can be made on the output. Basically, the system comes up with a massive data-base that tells which concepts are believed. However, this data-base is so large that it is very difficult to work with, and the form of the data (mostly domain and communication plans) is also difficult to work with. The entire graph takes up approximately 12 large screens, or an area 3'x2'; it is impossible to see it all at the same time, and printing it out and pasting it together is a chore. Even on a lisp machine it takes about 20 minutes to draw. The graph needs an intelligent browser that would allow people to look at sections of it.

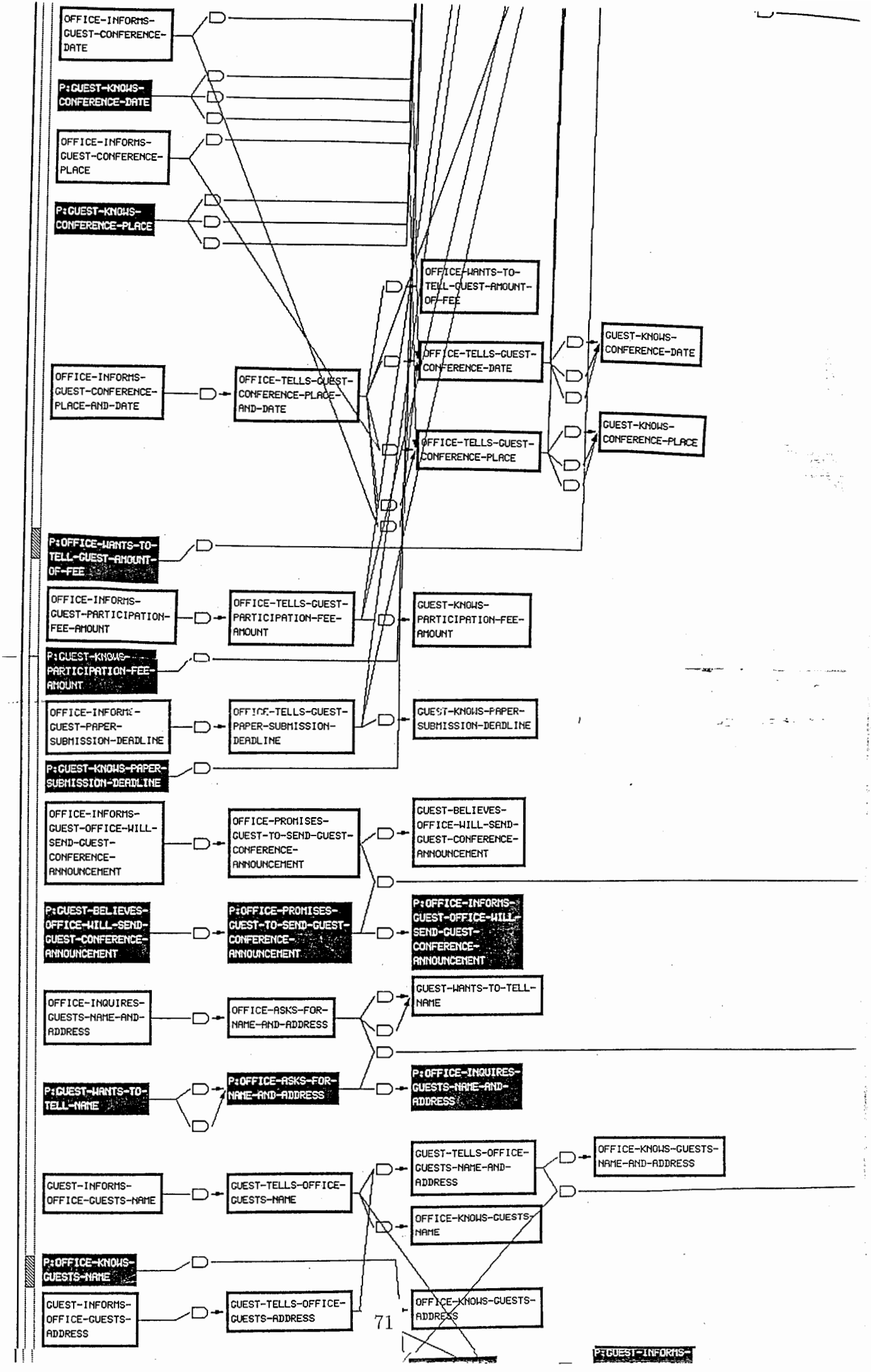
In addition, the astute reader will notice that the nodes do not contain feature structures, but rather long atomic names. This is because the feature structures that the system normally works with are much too long to display. Instead, a researcher has to have two versions of code–a short version that uses only tiny feature structures for research (the first RELN feature is used for the display output in this case), and a long version for actual development and system integration. This also needs to be improved.

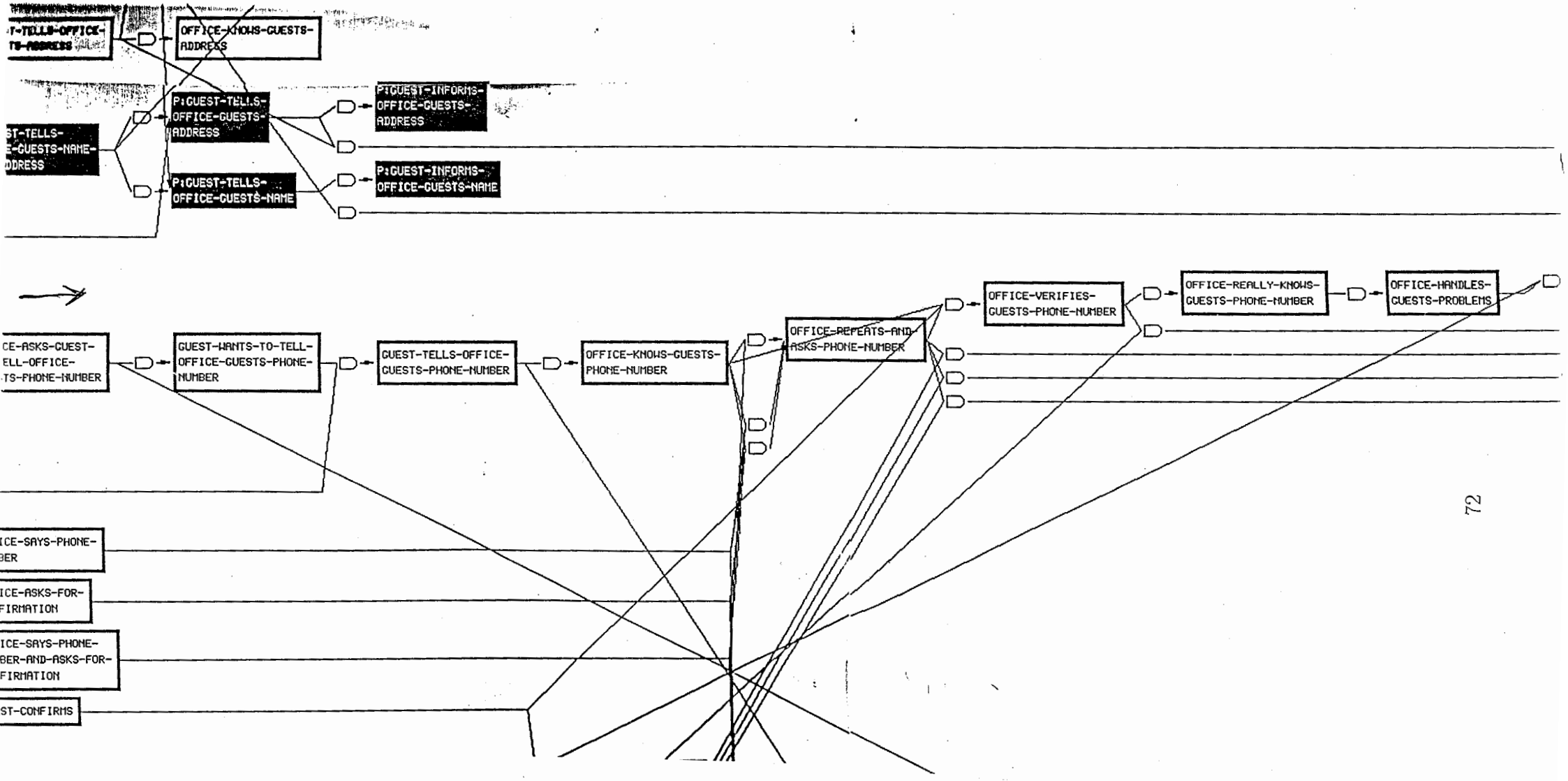
Note that the system easily handles logical conjunctions, such as "the office knows the guest's name and telephone number", which is supported by "the office knows the guest's name" and "the office knows the guest's telephone number".

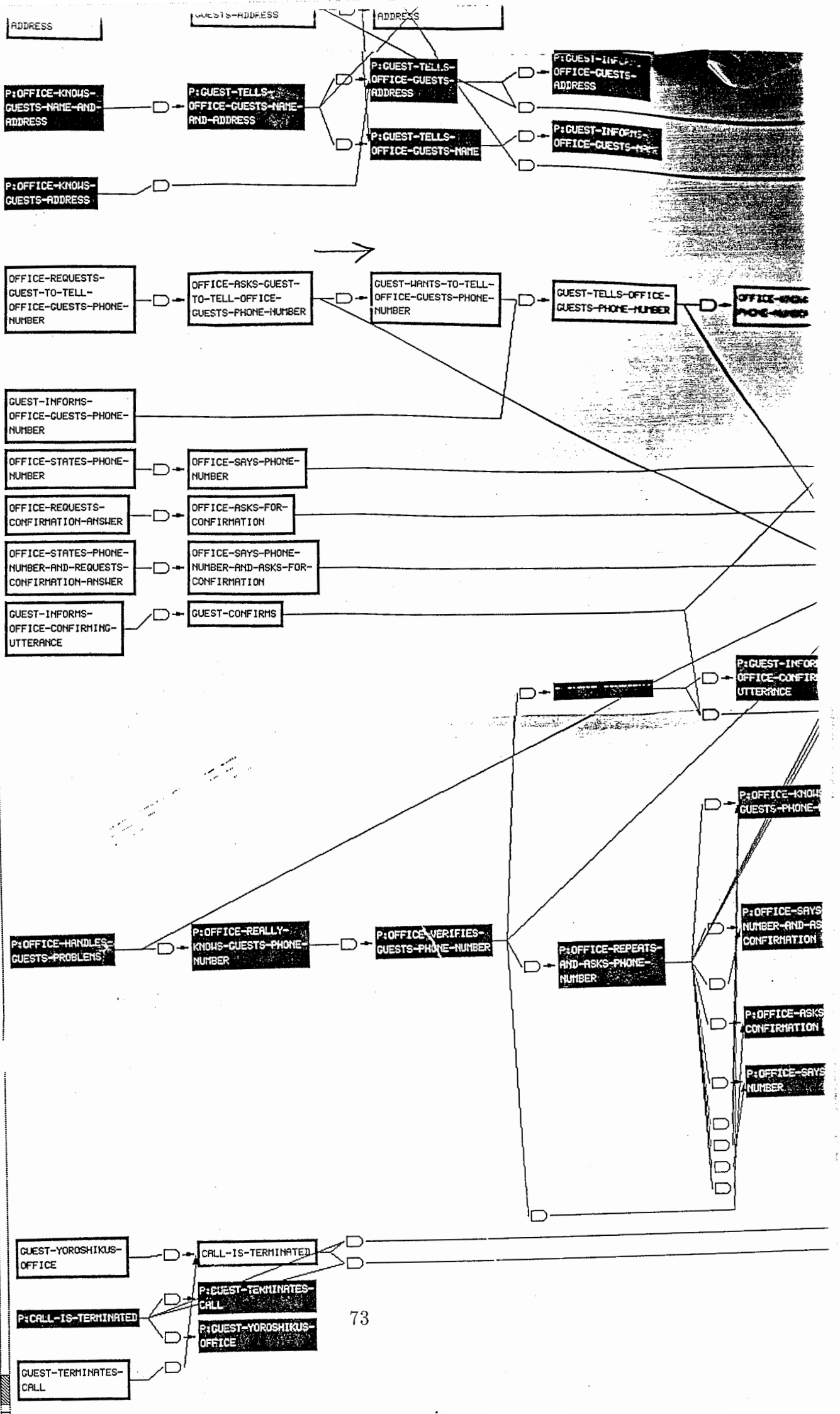
The results illustrate the previous evaluation of the system. The system performs plan recognition, prediction, and plan inference in a marvelous manner, but the resulting data-base requires at least an intelligent disambiguation system and other reasoning machinery in order to be usable in solving practical problems in machine translation.

```
Processing utterance:    My phone number is 372-8081.
Processing utterance:    372-8081.
Processing utterance:    Is that right?
Processing utterance:    372-8081, right?
Processing utterance:    That's right.
Processing utterance:    Yoroshiku.
Processing utterance:    Well, goodbye then.
Evaluation of (ACTIVATE-DEMO) took 25.053543 seconds of elapsed time including:
  0.717 seconds processing sequence breaks,
  1.367 seconds in the storage system (including 0.243 seconds waiting for pages):
    0.815 seconds processing 2513 page faults including 15 fetches,
    0.552 seconds in creating and destroying pages, and
    0.000 seconds in miscellaneous storage system tasks.
The garbage collector has flipped; so no consing was measured.
NIL
Command: (draw-graph (reverse *atms-nodes*))
```




OFFICE-INFORMS-GUEST-CONFERENCE-DATE
P:GUEST-KNOWS-CONFERENCE-DATE
OFFICE-INFORMS-GUEST-CONFERENCE-PLACE
P:GUEST-KNOWS-CONFERENCE-PLACE
OFFICE-WANTS-TO-TELL-GUEST-AMOUNT-OF-FEE
OFFICE-TELLS-GUEST-CONFERENCE-DATE
GUEST-KNOWS-CONFERENCE-DATE
OFFICE-INFORMS-GUEST-CONFERENCE-PLACE-AND-DATE
OFFICE-TELLS-GUEST-CONFERENCE-PLACE-AND-DATE
OFFICE-TELLS-GUEST-CONFERENCE-PLACE
GUEST-KNOWS-CONFERENCE-PLACE
P:OFFICE-WANTS-TO-TELL-GUEST-AMOUNT-OF-FEE
OFFICE-INFORMS-GUEST-PARTICIPATION-FEE-AMOUNT
OFFICE-TELLS-GUEST-PARTICIPATION-FEE-AMOUNT
GUEST-KNOWS-PARTICIPATION-FEE-AMOUNT
P:GUEST-KNOWS-PARTICIPATION-FEE-AMOUNT
OFFICE-INFORMS-GUEST-PAPER-SUBMISSION-DEADLINE
OFFICE-TELLS-GUEST-PAPER-SUBMISSION-DEADLINE
GUEST-KNOWS-PAPER-SUBMISSION-DEADLINE
P:GUEST-KNOWS-PAPER-SUBMISSION-DEADLINE
OFFICE-INFORMS-GUEST-OFFICE-WILL-SEND-GUEST-CONFERENCE-ANNOUNCEMENT
OFFICE-PROMISES-GUEST-TO-SEND-GUEST-CONFERENCE-ANNOUNCEMENT
GUEST-BELIEVES-OFFICE-WILL-SEND-GUEST-CONFERENCE-ANNOUNCEMENT
P:GUEST-BELIEVES-OFFICE-WILL-SEND-GUEST-CONFERENCE-ANNOUNCEMENT
P:OFFICE-PROMISES-GUEST-TO-SEND-GUEST-CONFERENCE-ANNOUNCEMENT
P:OFFICE-INFORMS-GUEST-OFFICE-WILL-SEND-GUEST-CONFERENCE-ANNOUNCEMENT
OFFICE-INQUIRES-GUESTS-NAME-AND-ADDRESS
OFFICE-ASKS-FOR-NAME-AND-ADDRESS
GUEST-WANTS-TO-TELL-NAME
P:GUEST-WANTS-TO-TELL-NAME
P:OFFICE-ASKS-FOR-NAME-AND-ADDRESS
P:OFFICE-INQUIRES-GUESTS-NAME-AND-ADDRESS
GUEST-INFORMS-OFFICE-GUESTS-NAME
GUEST-TELLS-OFFICE-GUESTS-NAME
GUEST-TELLS-OFFICE-GUESTS-NAME-AND-ADDRESS
OFFICE-KNOWS-GUESTS-NAME-AND-ADDRESS
OFFICE-KNOWS-GUESTS-NAME
P:OFFICE-KNOWS-GUESTS-NAME
GUEST-INFORMS-OFFICE-GUESTS-ADDRESS
GUEST-TELLS-OFFICE-GUESTS-ADDRESS
OFFICE-KNOWS-GUESTS-ADDRESS
P:GUEST-INFORMS-


OFFICE-KNOWS-GUESTS-ADDRESS
P:GUEST-TELLS-OFFICE-GUESTS-ADDRESS
P:GUEST-INFORMS-OFFICE-GUESTS-ADDRESS
P:GUEST-TELLS-OFFICE-GUESTS-NAME
P:GUEST-INFORMS-OFFICE-GUESTS-NAME
GUEST-WANTS-TO-TELL-OFFICE-GUESTS-PHONE-NUMBER
GUEST-TELLS-OFFICE-GUESTS-PHONE-NUMBER
OFFICE-KNOWS-GUESTS-PHONE-NUMBER
OFFICE-REPEATS-AND-ASKS-PHONE-NUMBER
OFFICE-VERIFIES-GUESTS-PHONE-NUMBER
OFFICE-REALLY-KNOWS-GUESTS-PHONE-NUMBER
OFFICE-HANDLES-GUESTS-PROBLEMS


ADDRESS
GUESTS-ADDRESS
ADDRESS
P:GUEST-TELLS-OFFICE-GUESTS-ADDRESS
P:OFFICE-KNOWS-GUESTS-NAME-AND-ADDRESS
P:GUEST-TELLS-OFFICE-GUESTS-NAME-AND-ADDRESS
P:GUEST-TELLS-OFFICE-GUESTS-NAME
P:OFFICE-KNOWS-GUESTS-ADDRESS
OFFICE-REQUESTS-GUEST-TO-TELL-OFFICE-GUESTS-PHONE-NUMBER
OFFICE-ASKS-GUEST-TO-TELL-OFFICE-GUESTS-PHONE-NUMBER
GUEST-WANTS-TO-TELL-OFFICE-GUESTS-PHONE-NUMBER
GUEST-TELLS-OFFICE-GUESTS-PHONE-NUMBER
GUEST-INFORMS-OFFICE-GUESTS-PHONE-NUMBER
OFFICE-STATES-PHONE-NUMBER
OFFICE-SAYS-PHONE-NUMBER
OFFICE-REQUESTS-CONFIRMATION-ANSWER
OFFICE-ASKS-FOR-CONFIRMATION
OFFICE-STATES-PHONE-NUMBER-AND-REQUESTS-CONFIRMATION-ANSWER
OFFICE-SAYS-PHONE-NUMBER-AND-ASKS-FOR-CONFIRMATION
GUEST-INFORMS-OFFICE-CONFIRMING-UTTERANCE
GUEST-CONFIRMS
P:OFFICE-HANDLES-GUESTS-PROBLEMS
P:OFFICE-REALLY-KNOWS-GUESTS-PHONE-NUMBER
P:OFFICE-VERIFIES-GUESTS-PHONE-NUMBER
P:OFFICE-REPEATS-AND-ASKS-PHONE-NUMBER
GUEST-YOROSHIKUS-OFFICE
CALL-IS-TERMINATED
P:GUEST-TERMINATES-CALL
P:CALL-IS-TERMINATED
P:GUEST-YOROSHIKUS-OFFICE
GUEST-TERMINATES-CALL

G 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. Optional arguments: Assumption-implication data, and the assumption probability (not used): `(assume-this-node node data prob)`.

(assumption data) Constructs and returns an Assumption node storing the given information. For future expansion, it is possible to assign a probability number to the assumption when it is created, by calling `(assumption data prob)`. Currently, the probabilities are not used otherwise by the system.

(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. Assigns an ID number to that node. 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.

(del-atms-node name-or-node) Hard-deletes an atms-node.

(del-env environment) Hard-deletes an environment. Not supported yet.

(del-implic implication) Hard-deletes an implication.

(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.

(find-node data) Finds the ATMS node that stores the given data. Returns NIL if the node was not there. Assumes "uniquification" is on.

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-node data antecedent-node1 A2 ...) Constructs and returns an implication. This function is mostly for human users. Same as `(justification ...)`. The consequents and the antecedents can either be atms-nodes or data. The system will check each consequent and antecedent node to make sure that it is in fact a node; if not, it will use the old node containing that data, or it will create a new atms-node for that data if necessary.

(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.

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

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

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

(implication-list consequent-node data (list antecedent-node1 A2 ...))
Constructs and returns an implication. This function is useful when you have a variable containing a list of the antecedents. The consequents and the antecedents can either be atms-nodes or data. The system will check each consequent and antecedent node to make sure that it is in fact a node; if not, it will use the old node containing that data, or it will create a new atms-node for that data if necessary.

(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 env) Same as `(nogood env)`. Poisons the given environment.

(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. It is now possible to call this routine several times on the same node, and install several different actions; when the node becomes IN, all of the actions are performed. The action should be of the form '(funcname arg1 arg2). Most of the time, one of the args will be the node itself. If the args are not constants, they must be evaluated: '(funcname ,node ,arg2). The function can have any number of nodes; the literal is simply stored and evaluated later.

(instantiate-goal ¡FSfact¿) States that the fact is a possible goal. Turns on all predicted nodes implied by that goal. The fact must be an internal FS.

(instantiate-known-goal ¡FSfact¿) States that the fact is an actual goal. Turns on all predicted nodes implied by that goal. The fact must be an internal FS.

(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.

(NP-Action [plan-FS]) Declares a plan schema to the system. The schema should be an explicit feature-structure. The semicolon character, ";", supports to-end-of-line comments, even inside the feature structure. It is important that the action have at least one precondition or decomposition; otherwise, it will never be instantiated and will be useless. The current version is UNABLE to accept extra features in the data to be matched, that are not described in the plan feature structure.

(NP-Input "documentation-string" [data-FS]) Declares an input data assertion to the system. The schema should be an explicit feature-structure. The semicolon character, ";", supports to-end-of-line comments, even inside the feature structure.

NP-to-LF This is a master global flag. When it is set to T before the system starts running, the system will print out logical forms for each atms-node that becomes IN (POSSIBLE or ACTUAL).

(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.

(pairwise-inconsistent node-or-data1 node-or-data2 ...) Sets each node in the set to be inconsistent with each other single node.

(pairwise-nogood node-or-data1 node-or-data2 ...) Sets each node in the set to be inconsistent with each other single node. Same as pairwise-inconsistent.

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

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

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

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

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

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

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

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

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

(print-assum assum) Prints an assumption.

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

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

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

(print-implic implic) Prints a given implication.

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

(print-env env) Prints an environment.

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

(print-node node) Individual item printing functions.

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

(reset-atms) Clears the system out.

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

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

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

(sys-implication consequent-node data antecedent-node1 A2 ...) Constructs and returns an implication. This function is mostly for computer users. Assumes that the consequents and antecedents are nodes already, and does not check for legality. This results in significant speed gains, at the cost of extra safety.

(sys-pairwise-inconsistent node1 node2 ...) Sets each node in the set to be inconsistent with each other single node. Does not check to make sure that the given nodes are in fact nodes.

(sys-pairwise-nogood node1 node2 ...) Sets each node in the set to be inconsistent with each other single node. Does not check to make sure that the given nodes are in fact nodes. Same as sys-pairwise-inconsistent.

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

(unassume name-or-node) Turns a node from an assumption back into a hypothetical node.

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

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

[AP80] James F. Allen and C. Raymond Perrault. Analyzing intention in utterances. *Artificial Intelligence*, 15:143–178, 1980.

[BFKM85] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5.* Addison-Wesley Publishing Co., Menlo Park, CA, 1985.

[CC89] Sandra Carberry and Kathleen D. Cebulka. Capturing rational behavior in natural language information systems. In Anthony G. Cohn, editor, *Proceedings of the Seventh Conference of the Society for the Study of Artifical Intelligence and Simulation of Behavior*, pages 153–163, Los Altos, CA, 1989. Morgan Kaufmann Publishers, Inc.

[Den87] Daniel C. Dennett. *The Intentional Stance.* The MIT Press, Cambridge, Mass., 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.

[EZ89] Martin C. Emele and Remi Zajac. Retif: A rewriting system for typed feature structures. Technical Report TR-1-0071, ATR Interpreting Telephony Research Laboratories, Kyoto, Japan, 1989.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.

[Gol70] Alvin I. Goldman. *A Theory of Human Action.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1970.

[Has89] Toshiro Hasegawa. The feature structure rewriting system manual. Technical Report TR-1-0093, ATR Interpreting Telephony Research Laboratories, Kyoto, Japan, 1989. (in Japanese).

[IKYA89] Hitoshi Iida, Kiyoshi Kogure, Kei Yoshimoto, and Teruaki Aizawa. An experimental spoken natural dialogue translation system using a lexicon-driven grammar. In *Computer World 89 in Osaka*, 1989.

[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.

[Kau87] Henry Kautz. A circumscriptive theory of plan recognition. In Philip R. Cohen and Martha E. Pollack, editors, *Symposium on Intentions and Plans in Communication and Discourse*. SRI International, Monterey, CA, March 1987.

[Kno88] Craig A. Knoblock. Data-driven plan recognition. CS Dept., Carnegie-Mellon University, Pittsburgh, PA, March 1988.

[Kog89] Kiyoshi Kogure. Parsing japanese spoken sentences based on hpsg. In *International Workshop on Parsing Technologies-89*, 1989.

[KU89] Akira Kurematsu and Yoshihiro Ueda. Generation in dialogue translation. In *Machine Translation Workshop at Univ of Manchester*, 1989.

[LA87] Diane J. Litman and James F. Allen. A plan recognition model for subdialogues in conversation. *Cognitive Science*, 11:163–200, 1987.

[LP89] Vladimir Lifschitz and Ed Pednault. Ijcai'89 tutorial on reasoning about actions and change. In *IJCAI'89*, Detroit, MI, 1989. Tutorial No. MP2.

[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.

[MT90] John K. Myers and Takashi Toyoshima. Known current problems in automatic interpretation: Challenges for language understanding. Technical Report TR-I-0128, ATR Interpreting Telephony Research Laboratories, Kyoto, Japan, January 1990.

[Mye88] John K. Myers. The necessity of intentions under fallible execution. ATR International, Kyoto, Japan, December 1988.

[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.

[Mye90] John K. Myers. A design for a disambiguation-based dialog understanding system. Technical Report TR-I-0189, ATR Interpreting Telephony Research Laboratories, Kyoto, Japan, November 1990.

[Mye92] John K. Myers. An agent-based approach to natural-language understanding of conversations for an interpreting telephone. In *International Symposium on Natural Language Understanding and AI (NLU +AI), as a part of the International Symposia on Information Sciences (ISKIT'92)*, pages 211–218, Kyushuu Institute of Technology, July 1992.

[Nag89] Masaaki Nagata. Expected semantic parser output for conversations 1-5. ATR corpus, August 1989. (feature structures in Japanese).

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[Pol86a] Martha E. Pollack. *Inferring Domain Plans in Question-Answering.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1986.

[Pol86b] Martha E. Pollack. A model of plan inference that distinguishes between the beliefs of actors and observers. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions & Plans: Proceedings of the 1986 Workshop*, Los Altos, CA, 1986. Morgan Kaufmann Publishers, Inc.

[SA77] Roger Schank and Robert Abelson. *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1977.

[Shi86] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CLSI, Stanford, CA, 1986.

[Wil86] Robert Wilensky. Points: A theory of the structure of stories in memory. In Barbara J. Grosz, Karen Sparck Jones, and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 459–473. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[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.