AN AUTOMATICALLY COMPILABLE RECOGNITION NETWORK FOR STRUCTURED PATTERNS

Frederick Hayes-Roth and David J. Mostow
Computer Science Department[1]
Carnegie-Mellon University
Pittsburgh, Pa. 15213

ABSTRACT

A new method for efficient recognition of general relational structures is described *and* compared with existing methods. Patterns to be recognized are defined by templates consisting of a set of predicate calculus relations. Productions are representable by associating actions with templates. A network for recognizing occurrences of any of the template patterns in data may be automatically compiled. The compiled network is economical in the sense that conjunctive products (subsets) of relations common to several templates are represented in and computed by the network only once. The recognition network operates in a bottom-up fashion, in which all possibilities for pattern matches *are* evaluated simultaneously. The distribution of the recognition process throughout the network means that it can readily be decomposed into parallel processes for use on a multi-processor machine. The method is expected to be especially useful in errorful domains (e.g., vision, speech) where parallel treatment of alternative hypotheses is desired. The network is illustrated with an example from the current syntax and semantics module in the Hearsay II speech understanding system.

INTRODUCTION

The work described in this paper was motivated by certain problems involved in the task of recognizing general structured patterns and, in particular, the problem of parsing continous spoken speech. From the point of view of the language parser, an essential quality of speech is its errorful nature. Ambiguities in acoustic segmentation, phonetic labelling, word hypothesization, and semantic interpretation necessitate understanding systems which can deal efficiently with multiple alternative hypotheses about each portion of the input. [11] The usual methods of dealing with such multiple hypotheses typically entail an expensive search through a combinatorial space, since they consider only one hypothesis for each portion of input at a time, and then exploit contextual relationships to eliminate certain combinations of adjacent hypotheses as impossible. The data structure and associated recognition procedure described in this paper can be thought of as effectively reversing this process by first exploiting context -- thereby eliminating all but a few combinations from consideration — and then testing contextuatly related hypotheses for adjacency. Since the contextual information is statically embedded in the data structure itself, comparatively little work needs to be done at recognition time. This work requires only the computation of a few, simple operations rather than a complex search. Moreover, the method provides an efficient way to handle the spurious insertions and deletions characteristic of speech.

TEMPLATE CRAMMARS

In this section, we define template grammars for recognizing relational structures. A template normal form (TNF) for template grammars is defined. An algorithm described elsewhere [7] translates a given template grammar into an equivalent TNF grammar which is economical in that it maximally exploits repeated subtemplates in the original grammar. The construction of an automatically compilable recognition network (ACORN) from a TNF grammar is described in the next section. The definitions we use are tailored to natural language understanding, but are immediately generalizable to other applications (e.g., vision).

A relation $r(x_1, ..., x_n)$ is an n-ary predicate corresponding to some element or pattern in the language. For example, the relation $tell(x_1, x_2)$ holds if the word "tell" occurs in the input utterance beginning at time x_1 and ending at time x_2. In general, x_1 and x_2 are temporal arguments specifying the time interval containing a recognized occurrence of the relation, and $x_3, ..., x_n$ are additional attributes of the occurrence. A relation is called primitive if it corresponds to a primitive element (terminal symbol) of the language, non-primitive if it corresponds to a pattern of elements (non-terminal symbol), and top-level if it corresponds to a complete pattern (sentential form).

A template T is a Boolean combination of relations r_i, $i=1, ..., |T|$, restricted as follows. It must be either a disjunction

$$r_1(x_1, ...,x_n) \vee r_2(x_1, ..., x_n) \vee ...$$

$$\vee\, r_d(x_1, ..., x_n),\ |T| = d \geq 1,$$

or a conjunction

$$r_1(x_{1_1}, ..., x_{n_1}) \wedge ... \wedge r_p(x_{1_p}, ..., x_{n_p}) \wedge$$

$$\neg r_{p+1}(x_{1_{p+1}}, ..., x_{n_{p+1}}) \wedge ... \wedge$$

$$\neg r_q(x_{1_q}, ..., x_{n_q}),\ q \geq p \geq 1.$$

In the first case (disjunction), the symbolic arguments $(x_1, ..., x_n)$ are the same for each r_i, $i = 1, ..., d$. In the second case, a weaker condition must be satisfied: the relations must have enough symbolic arguments in common for the template to be connected, that is, for any partition of the p+q relations $r_1, ..., r_{p+q}$ into two non-empty sets A and B, there must exist relations $r_a(x_{1_a}, ..., x_{n_a}) \in A$, $r_b(x_{1_b}, ..., x_{n_b}) \in B$ with $\{x_{1_a}, ..., x_{n_a}\} \cap \{x_{1_b}, ..., x_{n_b}\} \neq \emptyset$.

A template grammar is a set of rules of the form [<template> => <relation>; <action>]. The action optionally associated with each rule specifies what should be done in the event that an instance of the template is recognized in the input and the rule is invoked. Thus a template grammar is actually a production system of the sort described by Newell [12].

Table 1. Sample Grammar (G_{AP})

1. [$Ford(t_1, t_2)$ => $TOPIC(t_1, t_2, expr)$; expr←"FORD"]

2. [$Rockefeller(t_1, t_2) \vee Rocky(t_1, t_2)$ => $TOPIC(t_1, t_2, expr)$; expr←"ROCKEFELLER"]

3. [$Kissinger(t_1, t_2)$ => $TOPIC(t_1, t_2, expr)$; expr←"KISSINGER"]

1 This research was supported in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

4. { or(t_1, l_2) $\wedge$ TOPIC(t_2, t_3, expr) => TOPIC*(t_1, t_3, expr);]

5. [TOPIC(t_1, l_2, expr_1) $\wedge$ TOPIC*(t_2, t_3, expr_2) => TOPIC*(t_1, t_3, expr); expr ← expr_1 ∪ expr_2]

6. [UTTERANCE(t_1, t_4) $\wedge$ about(t_2, t_3) $\wedge$ TOPIC(t_3, t_4, expr) => TOPIC*(t_3, t_4, expr);]

7. [UTTERANCE(t_1, t_6) $\wedge$ tell(t_1, t_2) $\wedge$ me(t_2, t_3) $\wedge$ nothing(t_3, t_4) $\wedge$ about(t_4, t_5) $\wedge$TOPIC*(t_5, t_6, expr) => REJECT(t_1, t_6, expr); SUPPRESS(expr)]

8. [UTTERANCE(t_1, t_6) $\wedge$ tell(t_1, t_2) $\wedge$ me(t_2, t_3) $\wedge$ ¬nothing(t_3, t_4) $\wedge$ about(t_4, t_5) $\wedge$TOPIC*(t_5, t_6, expr) => REQUEST(t_1, t_6, expr); RETRIEVE(expr)]

As an example, consider the sample template grammar GAp (Table 1) which is part of a much larger grammar for analyzing spoken queries to a wire-service news retrieval system. [4] GAp's top-level relations are REQUEST and REJECT, An instance of REQUEST is the utterance "Tell me all about Rocky " An instance of REJECT is the utterance "Tell me nothing about Ford, Rockefeller, or Kissinger." The primitive relation UTTERANCES(l_1, t_2), used in rules 6, 7, and 8, simply signifies that the entire utterance spans the time interval [t_1.t_2]; this makes the beginning and ending times of the utterance accessible as arguments to other relations, without violating the framework of the template grammar. Rule 2 illustrates the use of features and actions. The feature, expr, of TOPIC is the semantic expression eventually passed to the actual news retrieval routine. The action of Rule 2 gives expr the value "ROCKEFELLER." Rule 5 is an example of recursion. It handles phrases of the form "topicj, topic2, •-, topic$_{n\_j}$, or topic$_n$" The action of Rule 5 forms a compound semantic expression from the expressions associated with its individual constituents. Thus the instance "Ford, Rockefeller, or Kissinger** of the relation TOPIC* has expr - {"FORD", "ROCKEFELLER", "KISSINGER"}. Rule 6 shows how context sensitivity can be embedded in a template grammar. It states that *any* instance of TOPIC which occurs at the end of an utterance, and whose left context is ABOUT, constitutes an instance of TOPIC*. Rule 8 illustrates the use of negation. It states that any utterance of the form "Tell me ... about X" is a request for information about X unless the gap "..." contains the word "nothing." Thus "Tell me about Ford," "Tell me all about Ford," and "Tell me everything you know about Ford," are all instances of REQUEST. This illustrates the capacity of a template grammar to ignore redundant portions of the input.

A template grammar is in <u>template normal form</u> (TNF) if the following conditions are satisfied:

(1) The template of each rule has one of the following types:

<relation_1> v <relation_2> v ... v <relation_d>, $d \geq 1$ (disjunctive type)

<relation_1> $\wedge$ <relation_2> (conjunctive type)

<relation_1> $\wedge$ ¬ <relation_2> (negative type)

The relations in a disjunctive template have the same symbolic arguments; the relations in a conjunctive or negative template are connected.

(2) Every non-primitive relation appears on the right side of exactly one rule. Hence we can define the <u>type</u> of a relation to be the type of its unique defining template; a primitive relation is simply said to be of primitive type.

It is clear that any template grammar G can be translated into an equivalent grammar G* in TNF by means of adding new relations and rules. The task of the automatic translator is to do this in such a way as to minimize the number of new relations added. The algorithm we employ is described in [7], The result of applying the algorithm to the sample grammar of Table 1 is grammar G^p , shown in Table 2. Mnemonic conventions used in GAp* are these: "+" indicates concatenation; "-" indicates temporal overlap; parenthetical phrases indicate temporal contexts; and "/k" distinguishes different TNF relations arising from occurrences of a single relation in various different rules of the original grammar.

Table 2. Sample Grammar in TNF (G_{AP}^*)

1*. [Ford(t_1, l_2) => TOPIC/1(t_1, t_2, expr); expr←"FORD"]

2*. [Rockefeller(t_1, t_2) v Rocky(t_1, t_2) => TOPIC/2(t_1, t_2, expr); expr←"ROCKEFELLER"]

3*. [Kissinger(t_1, l_2) => TOPIC/3(t_1, t_2, expr); expr←"KISSINGER"]

4*. { or(t_1, t_2) $\wedge$ TOPIC(t_2, t_3, expr) => TOPIC*/4(t_1, t_3, expr);]

5*. [TOPIC(t_1, t_2, expr_1) $\wedge$ TOPIC*(t_2, t_3, expr_2) => TOPIC*/5(t_1, t_3, expr); expr ← expr_1 ∪ expr_2]

6**. [UTTERANCE(t_1, t_3) $\wedge$ (ABOUT)TOPIC(t_2, t_3, expr) => TOPIC*/6(l_2, t_3, expr);].

7****. [TELL•ME-UTTERANCE-ABOUT•TOPIC* (t_2, t_3, t_1, t_4, expr) $\wedge$ nothing(t_2, t_3) => REJECT(t_1, t_4); SUPPRESS(expr)]

8****. [TELL•ME-UTTERANCE-ABOUT•TOPIC* (t_2, t_3, t_1, t_4, expr) $\wedge$ ¬ nothing(t_2, t_3) => REQUEST(t_1, t_4); RETRIEVE(expr)]

9. [tell(t_1, t_2) $\wedge$ me(t_2, t_3) => TELL•ME(t_1, t_3);]

10. [about(t_1, t_2) $\wedge$ TOPIC*(t_2, t_3) => ABOUT•TOPIC*(t_1, t_3);]

11. [TELL•ME(t_1, t_2) $\wedge$ UTTERANCE(t_1, t_3) => TELL•ME-UTTERANCE(t_2, t_3, t_1);]

12. { TELL•ME-UTTERANCE(t_2, t_4, t_1) $\wedge$ ABOUT•TOPIC*(t_3, t_4, expr) => TELL•ME-UTTERANCE-ABOUT•TOPIC* (t_2, t_3, t_1, t_4, expr);]

13. [about(t_1, t_2) $\wedge$ TOPIC(t_2, t_3, expr) => (ABOUT)TOPIC(t_2, t_3, expr);]

14. [TOPIC/1(t_1, t_2, expr) v TOPIC/2(t_1, t_2, expr) v TOPIC/3(t_1, t_2, expr) => TOPIC(t_1, t_2, expr);]

15. { TOPIC*/4(t_1, t_2, expr) v TOPIC*/5(t_1, t_2, expr) v TOPIC*/6(t_1, t_2, expr) => TOPIC(t_1, t_2, expr);]

THE REC06NITI0N NETWORK

Given a template grammar in TNF, a corresponding recognition network (ACORN), as first described in [6], is constructed as follows. For each relation r appearing in the TNF grammar, there is a unique node, node(r), in the network. (Hence minimizing the number of relations in the TNF grammar is equivalent to minimizing the number of nodes in the network.) For every rule [T-> r; A], an. arc is drawn from node(sj) to node(r) (or each relation Sj in the template T. Each node(Sj) is said to be a constituent of node(r), and node(r) a derivative of node(S|). A node may have zero, one, or more derivatives. The recognition network for the sample grammar GAp, constructed from the TNF grammar GAp*, is shown in Figure 1.

Node(r) contains various information: its type (i.e., the type of relation r); the action A in the rule [T => r; A], if any; and the correspondence between the arguments of relation r and the arguments of its constituent relations Sj. This correspondence consists of two parts, a set of tests and a generator. The tests represent any requirements for agreement between the arguments supplied by the constituents node(Sj). The generator is a list of the arguments which are to be supplied in turn to the derivatives of node(r). The arguments *are* encoded according to a canonical numbering scheme best described by an example. Consider node(TELL»ME). Its constituents are node(TELL), which supplies arguments t_1, t_2, and node(ME), which supplies arguments t_3,t_4 Let concatenated argument list (t_1,$t_2$$t_3$,$t_4$). Then node(TELL*ME) can specify its arguments by their indices in L. Thus node(TELL+MErs only test is L*(2)* = L(3), denoted by "2:3" below node(TELI*ME> in the network. (See Figure 1.) Similarly, node(TELL*ME)'s generator is the list < L(I), L<4)), denoted by "(1, 4)^M above node(TELL»ME> in the network. Arguments which are not supplied by a node*s constituents but instead originate at the node itself are specified by negative indices. For example, node(T0P!C/2Vs generator is denoted by "(1, 2, -1),^{H} the -1 specifies the argument expr, which originates at node(T0PIC/2). The action stored in node(T0PIC/2) assigns this argument the value "ROCKEFELLER"

All of the recognition network components described so far are static. There is also associated with each node(r) a dynamic instance list 1L. Each instance in the instance list of node(r) represents a single recognized occurrence (instantiation) of the relation r in the input utterance. An instance has several components: a unique identification number I; the time interval [xj, x^] containing the occurrence; the values X3. . . . x_p of additional attributes of the occurrence; and a support set SS containing one or two instance identification numbers. An instance is denoted I:(xi, ... *$_n$; SS). During the recognition process, instances are created and deleted dynamically.

The recognition process is bottom-up, as follows. Initially all instance lists are empty. A lexical analyzer is invoked and begins to scan for occurrences of primitive relations in the input utterance. Since the lexical analyzer receives imperfect, incomplete information from the phonetic labelling routine, the best it can do is to identify possible occurrences. When it finds a possible occurrence of a relation r, it adds a new element to the instance list of node(r) containing the appropriate information. To understand the recognition process, imagine each node(r) as having a demon. The node(r) demon continuously monitors the instance list of each constituent node(s_t) of node(r). Whenever a new instance is added to the instance list of node(Sj), the node(r) demon adds a reference to this new instance to its node(Sj) add set. Similarly, whenever an existing instance of s_f is deleted, the node(r) demon saves a copy of it in its node(Sj) delete set. Add sets and delete sets are referred to collectively as change sets. [9] The demon then activates (wakes) node(r) itself by invoking code pointed to by node(r).

When node(r) is activated, it updates its instance list according to the information in its constituents' instance lists and change sets. If node(r) can derive (construct) any new instances from instances of its constituents, it does so, adding the new instances to its instance list. The support set of each instance contains the identification numbers of the instances from which it has been derived. Node(r) deletes from its instance list any instances supported by (derived from) the defunct instances listed in its constituents' delete sets. The exact way in which all this is done depends, of course, on the type of node(r).

If node(r) is disjunctive, then it has d constituents node(s_1), ..., node(s_d). For each instance I:(x_1, ..., x_n; SS) in a node(s_i) add set, node(r) adds a new element I_{new}:(z_1, ..., z_k; {I}) to its own instance list, computing z_1,..., z_k from the values of x_1, ..., x_n according to the generator stored in node(r). I_{new}'s support set is {I} because the instance I_{new} of r is derived from (supported by, dependent on) the instance I of r's constituent relation s_i. For each defunct instance I in a node(s_i) delete set, node(r) deletes all instances I_{old}:(z_1, ..., z_k; SS) supported by I, i.e., such that $I \in SS$. (Actually, for disjunctive r, all instances of r will have support sets of size one, so $I \in SS$ iff $SS = \{I\}$. However, for conjunctive r, $|SS| = 2$; hence the set notation).

If node(r) is conjunctive, then it has exactly two constituents, node(s_1) and node(s_2), with respective instance lists IL_1 and IL_2, add sets AS_1 and AS_2, and delete sets DS_1 and DS_2. First node(r) deletes any of its instances I_{old}:(z_1, ..., z_k; SS) which were derived from instances in DS_1 or DS_2, i.e., those for which $SS \cap (DS_1 \cup DS_2) \neq \emptyset$. Then node(r) looks for new instance pairs I_1:(x_1, ..., x_n; SS_1) in IL_1 and I_2:(y_1, ..., y_m; SS_2) in IL_2 such that (x_1, ..., x_n) matches (y_1, ..., y_m) according to the tests stored in node(r). For each such matching pair, node(r) adds a new element I_{new}:(z_1, ..., z_k; {I_1, I_2}) to its instance list, using its generator to select z_1, ..., z_k from x_1, ..., x_n, y_1, ..., y_m. It is sufficient to check only those pairs of instances I_1, I_2 of which one or both are new, or more formally, such that either $I_1 \in AS_1$ and $I_2 \in IL_2$ or $I_1 \in IL_1$ and $I_2 \in AS_2$. For example, suppose the input utterance is "Tell me nothing about Rockefeller," and the lexical analyzer finds an instance I_1:(0, 18; ...) of tell and an instance I_2:(18, 23; ...) of me. Then the test stored in node(TELL•ME) becomes 18 = 18, which is true, so node(TELL•ME) adds a new instance I_{new}:(0, 23; {I_1, I_2}) to its instance list to represent the occurrence of "tell me" in the concatenated time interval [0, 23]. (Time is measured in centiseconds since the beginning of the utterance.) Now suppose the lexical analyzer mistakenly identifies the syllable "fell" in "Rockefeller" as the word "tell," and adds an instance I_3:(257, 269; ...) to node(tell)'s instance list. This may happen, for example, if the phonetic labeller correctly identifies the "F" in "Rockefeller" as an unvoiced consonant but can't tell if it's an "F," a "T," or a "P." No harm is done, however, since when node(TELL•ME) matches I_3 against I_2, the test 269 = 18 fails, and no new instance of TELL•ME is derived from I_3. This example shows how the ACORN automatically weeds out spurious instances hypothesized by the lexical analyzer on the basis of incomplete phonetic information.

Finally, if node(r) is negative, then it has two constituents, node(s_1) and node(s_2), where $r = (s_1 \wedge \neg s_2)$. Let IL_1, IL_2, AS_1, AS_2, DS_1, DS_2 be the instance lists, add sets, and delete sets of node(s_1) and node(s_2). First node(r) deletes any of its instances I_{old}:(z_1, ..., z_k; SS) derived from defunct instances in DS_1, i.e., those for which $SS \cap DS_1 \neq \emptyset$. Then node(r) looks for any instance pairs I_1:(x_1, ..., x_n; SS_1) in IL_1 and I_2:(y_1, ..., y_m; SS_2) in AS_2 such that (x_1, ..., x_n) matches (y_1, ..., y_m) according to the tests stored in node(r). For each such pair, node(r) deletes all of its instances I_{old}:(z_1, ..., z_k; SS) which depended on I_1, i.e., such that $I_1 \in SS$. This is done since each such I_{old}, previously an instance of $(s_1 \wedge \neg s_2)$, is now invalidated by a new instance of s_2. Adding instances of node(r) is also a bit tricky, and proceeds as follows. First node(r) constructs the set IS of all instances $I_1 \in IL_1$ which match some I_2 in DS_2. Then node(r) looks for all instances I_1:(x_1, ..., x_n; SS_1) in $AS_1 \cup IS$ which match none of the instances in IL_2. For each such I_1, node(r) adds a new instance I_{new}:(z_1, ..., z_k; {I_1}) to its instance list.

To illustrate this, let us continue with our sample utterance, "Tell me nothing about Rockefeller." Suppose that at some point the lexical analyzer has recognized all the words in the utterance except the word "nothing," and node(TELL•ME-UTTERANCE-ABOUT•TOPIC*) has I_4:(23, 41, 0, 274, "ROCKEFELLER"; ...) on its instance list. Since the instance list of node(nothing) is empty, node(REQUEST) will have an instance I_5:(0, 274, "ROCKEFELLER"; {I_4}) on its instance list. Now suppose that the lexical analyzer finally recognizes the word "nothing," and puts the instance I_6:(23, 41; ...) on node(nothing)'s instance list. This activates both of node(nothing)'s derivatives. Node(REJECT) matches I_6 against I_4, tests 23 = 23 and 41 = 41, and accordingly adds a new instance I_7:(0, 274, "ROCKEFELLER"; {I_4, I_6}) to its instance list. Node(REQUEST) matches I_6 against I_4, tests 23 = 23 and 41 = 41, and accordingly deletes I_5:(0, 274, "ROCKEFELLER"; {I_4}) from its instance list. This example shows how information is accumulated and corrected dynamically during the ACORN recognition process. It also illustrates the ACORN's state-saving nature and its sharing of information between top-level nodes.

Once node(r) has examined its constituents' change sets and, if appropriate, revised its own instance list, it goes back to sleep. Meanwhile, the demons sitting on the derivatives of node(r) have been watching its instance list and, when changes occur, activate their nodes. This chain reaction continues, fuelled by new instances generated by the lexical analyzer, until the lexical analyzer has stopped, all nodes are asleep, and all change sets are empty.

At this point each instance I:(x_1, ..., x_n; SS) of a non-primitive node, node(r), may be interpreted as a partial parse of the interval [x_1, x_2], with relevant syntactic and semantic features given by x_3, ..., x_n. For example, when the recognition of our sample utterance terminates, the instance I:(41, 274, "ROCKEFELLER"; ...) of ABOUT•TOPIC* may be considered to be a partial parse of the input interval [41, 274] containing "about Rockefeller." Parse trees can easily be reconstructed from the information contained in the support sets. Parses of the entire utterance are given by instances of top-level nodes. Thus the instance I_7:(0, 274, "ROCKEFELLER"; {I_4, I_6}) of REJECT constitutes a total parse of the sample utterance, and supplies the semantic feature, expr, required by the action SUPPRESS(expr).

RELATIONSHIP TO EXISTING PARSERS AND PATTERN-MATCHERS

The original motivation which led to the ACORN concept was the development of a general automatic recognition system for spoken utterances, visual scenes, and other structured patterns in which context is a fruitful source of information. Since the speech understanding ACORN treats an utterance as a relational structure, it is related both to natural language parsers and to general pattern-matching mechanisms.

The ACORN's closest relative among natural language parsers is PARRY [2], a program which simulates a paranoid individual being interviewed by a psychiatrist. PARRY employs a large library of stored concept sequence templates which are compared with segments of typewritten input sentences. Generalization is achieved by rules which rewrite words as synonymous concepts, delete unrecognized words and, if necessary, delete one recognized word at a time until a template is matched. While the *approach* underlying PARRY is very successful with typed input, it appears to be too risky for spoken input. Unlike the "perfect" input which PARRY receives, the input to the syntax module of a speech understanding system such as Hearsay II [9] is highly imperfect. PARRY can say, with confidence, "this portion of the input is such-and-such (e.K., the word "oh"), so III ignore it;" Hearsay II can only say "if this portion of the input is "oh," I can ignore it; but if it's really the word "no," then HI need it." An ACORN can be thought of as a non-deterministic version of a PARRY-Itke system in which aft possible parses *are* followed simultaneously in parallel. On the other hand, an ACORN is capable of recognizing general graph structures and is more powerful than any context-sensitive language parser (string recognizer).

Woods' augmented transition network (ATN) [14] is a mechanism for parsing natural language. It works top-down, uses backtracking, and produces a formal parse of the input sentence. In contrast, an ACORN works bottom-up, does no backtracking, and extracts only those features of the utterance which are relevant to the particular application. An ACORN can be thought of as a state-saving, bottom-up version of an ATN.

Miller [10] has proposed a parser for spoken English which builds multiple partial parse trees and employs a complicated and heuristic search to combine them. An ACORN differs from Miller's parser in handling all combinations simultaneously rather than sequentially, and in the simplicity of the matching operations it uses.

Current artificial intelligence programming systems such as PLANNER [8], QA4 [13], and SAIL [3] can match a given relational template against a data base. However, the method they use is an exhaustive, iterative, and associative search. If several templates are to be matched against the data base, they must be matched one at a time In contrast, the associative matching operation performed by ACORNs effectively tests all the relations of all the templates simultaneously.

The ACORN's nearest relative among general pattern-matching methods is hierarchical synthesis [1]. Consider the task of matching a template, such as a schematic representation of a building, against an input set of line segments. A recognition algorithm employing hierarchical synthesis replaces the single, many-component template for "building" with a hierarchy of templates for "doors," "windows," "stories," etc. A higher-level template can be matched only if its lower-level constituents *are.* Hierarchical synthesis considerably reduces recognition time for two reasons. First, it can exploit the repetition of subtemplates by recognizing all instances of a single subpattern just once. Second, before considering whether or not the entire pattern specified by a template is present, it can insure that all necessary subpatterns are present.

However, hierarchical synthesis as described in [I] depends on a hierarchy defined a priori by the user. This limitation is transcended by Hayes-Roth's interference matching method [5], which does hierarchical synthesis in parallel in all possible directions, thereby obviating the need for a predefined hierarchy. In interference matching, a template is represented as a set of relations. Each relation is a predicate with one or more symbolic variables. The input is also a set of relations, whose arguments *are* constants. A partial match consists of an assignment of input constants to the symbolic variables of a subset of template relations which makes them all true. Interference matching works by finding partial matches and combining them into complete matches.

Like interference matching, the ACORN method is an improved version of hierarchical synthesis in that it requires no predefined hierarchy. The ACORN compiler itself determines an economical hierarchy, and embeds it in the form of a recognition network Hierarchy selection can be factored out into a separate compilation phase because the choice of hierarchy depends only on the templates *and* not on the individual input utterance. In interference matching, on the other hand, hierarchy selection depends on the input pattern, and is therefore a part of the recognition process. Thus the

ACORN method combines the convenience of automatic hierarchy selection with the efficiency which comes from using a predefined hierarchy in the recognition process.

In real-world applications, input is matched against several top-level templates. Current methods of hierarchical synthesis and interference matching involve matching the input against one template at a time. Such an approach is clearly undesirable for tasks such as speech recognition, which may involve large numbers of templates. The ACORN compiler takes a whole set of templates and produces a single, unified recognition network for it; common subtemplates *are* shared not just <u>within</u> top-level templates but also <u>between</u> them. An instance of a subtemplate ts recognized just once -- not separately for each top-level template in which it occurs. Hence recognition time depends not on the total number of templates, but just on the number of templates which match some portion of the input. This property is encouraging, since the number of templates required to recognize a significant subset of English would probably be several thousand.

In sum, an ACORN can be looked at as a bottom-up version of an ATN; a parallel and non-determinislic version of a PARRY-like system; a general pattern-matcher; or an improved mechanism tor rMerarchital synthesis, with automatic hierarchy selection and subtemplate sharing between templates.

<u>APPLICAT</u>IONS. <u>IMPLICATIONS</u> ANP EXTENSIONS

In order for art ACORN to be efficient, the templates and input data characteristic of the chosen problem domain should tend to be asymmetric, so that a template will usually match a given portion of the input in at most one way. Let us illustrate with a negative example Suppose the template we wish to match js K_5 (a , c, d, e), the complete graph on five vertices, represented by the conjunction of relations line(a, b) A hne(a, c) A ... A line. Then any occurrence of K5 (as a subgraph, say) in the input corresponds to 5! =120 instances of T,since there are 5! ifferent ways to bind the variables a, b, c, d, e to the five vertices of the K_5in the input. For symmetries on a larger scale, the problem grows combinatorially worse Clearly, *an*ACORN would be inefficient in such a domain, since it would insist on finding all instances of every template.

Fortunately, many problem domains do not exhibit this bothersome property. Speech, in particular, is highly asymmetric, partly because it is embedded in a one-dimensional ordered temporal domain. If tell (t_1,t_2) is true, then $t_1 < t_2$, so $tell(t_2, t_1)$ cannot be true. Symmetries at a higher level can occur only if there is more than one syntactically and semanticaliy valid way to group the input words into phrases, i.e,, if the input is inherently ambiguous.

What are the advantages of ACORNs for speech understanding? The bottom-up template-oriented approach is especially conducive to handling natural, idiomatic, conversational natural language robustly. Consider the problem in spoken speech of spurious insertions such as "oh," "urn," "er." We wish to treat them the same as silences. We do this by adding rules like [oh(t_1,.t_2)=> SILENCE(t_1, t_2);] to our template grammar, and relaxing the test $t_2=t_3$ for temporal adjacency between two relation instances, such as tell(t_1, t_2) and me(t_3, t_4), to compute $t_2=t_3$ v SILENCEttp (t_2, t_3).

This example also illustrates the reason for non-**deterministic application of Colby's methods in a speech understanding system. Even if a spurious insertion is recognized, the corresponding portion of the input must not be discarded, since it may have been recognized incorrectly. If an ACORN recognizes an instance of "oh" in the interval $[t_1, t_2]$, it puts the instance $(t_1, t_2; ...)$ on the instance list of SILENCE, without discarding any information. That way, if the interval** actually contains the word "no," it is still there for the lexical analyzer to find. In contrast, when PARRY ignores information, it throws it away altogether.

Another phenomenon common to conversational speech is the idiomatic expression, e.g "How *$re* you?" Using an ACORN, we can simply include explicit template rules for such expressions, <u>e.g.,</u>

{how•are•you(t_1, t_2) =>
GREETING(t_1, t_2); REPLY("Fine, how are you?")],

thereby short-circuiting the detailed syntactic parse which would be attempted by a more formal system such as Woods'.

The two techniques just described can be combined. Certain idioms such as "by the way" carry essentially no useful information and can be treated as spurious insertions by rules like

[by(t_1, t_2) $\wedge$ the(t_2, t_3) $\wedge$ way(t_3, t_4) =>
SILENCE(t_1, t_4);]

Some expressions occur either as meaningless idioms or as meaningful phrases, depending *on* context. Consider, for example, the utterance "I see, could 1 see the midnight digest?," which occurred in an actual experimental protocol The first occurrence of "J see" is idiomatic and can be ignored; the second is essential to the meaning of the utterance. An ACORN, in processing this utterance, would recognize both occurrences as instances of SILENCE, without discarding any information. The first occurrence would be ignored, as desired, but the second one would still be available to match other templates.

Spurious deletions can also be handled by ACORNs. To handle spurious deletions, we want to permit partial matching of templates We can do this within the ACORN framework simply by adding extra templates corresponding to commonly occurring partial matches of the original templates. The obvious weakness of this method is that it requires a <u>priori</u> Knowledge of which deletions are likely to occur. The success of the method might require many iterations over a large corpus of lest utterances, with new templates added as needed. Hopefully this process would converge, after a reasonable number of such iterations, to acceptable performance with respect to handling spurious deletions. (This method of "massive iteration" seems to have worked successfully for PARRY.)

Partial templates can be used for another purpose as well. Although the bottom-up approach has several advantages, as described above, it is useful to have certain properties associated with top-down processing One such property is the ability to focus the attention of lower-level modules on critical portions of input. Another is the ability to hypothesize words from above, for lower-level modules to confirm or reject. Although we earlier referred to a lexical analyzer which finds all instances of primitive relations (words) in the input utterance, this would in practice be too expensive. The actual Hearsay II system seeks to constrain hypothecation as, much as possible; to do this it applies high-level information to cut down the number of plausible words matched against each portion of the input. Thus it is desirable to have a speech understanding ACORN generate intermediate partial information telling the lower level modules which portions of the input they should concentrate on processing, and which words *are* likely to occur at a given place in the input, *on* the basis of the already recognized portions of the surrounding context.

This top-down extension to the basic bottom-up mechanism requires knowledge about the predictive value of partial templates. For example, we know that "What time" often occurs in the phrase "What time is it?" We can incorporate this information in an ACORN by including a rule

[what(t_1, t_2) $\wedge$ time(t_2, t_3) =>
WHAT•TIME(t_1, t_3); TEST(t_3, t_{any}; "is it")],

where TEST is the action invoked upon recognition of the template The effect of the TEST is to look for the missing instance of "is it" starting at the time t_3 in the input utterance. If it is found, it is added to the instance list for is+it, leading to the desired completion of the full template "What time is it."

In the above example, a partial template was used to predict downwards in the network. Partial templates can also be good upward predictors For example, given an instance of the partial template T_1 = "time is it," the probability $P(T_2|T_1)$ that it occurs as part of the template T_2= "What time is it" may approach certainty. If $P(T_2|T_1)$ is high enough, say .99, we may wish to save processing time by simply predicating that T_2 does in fact occur. Such a scheme is currently being implemented.

EVALUATION AND CONCLUSIONS

A full evaluation of the ACORN method must of course await experience with large-scale implementations. In the meantime, there are several properties we observe from the current, partial implementation.

(1) The recognizer is efficient.

(2) It ts extremely easy to modify, since changes are restricted to The template grammar.

(3) Using an ACORN makes it possible to dispense with a formal parse.

(4) Even when an ACORN cannot fully parse an utterance, it can still provide a partial parse.

(5) ACORNs are organized so as to factor recognition processing into simple, universal, and independent operations performed at the nodes. This has made them trivial to implement and, in addition, makes them well-suited to parallel execution on a multiprocessor.

Finally, we expect ACORNs to have a broad range of applications, since they seem well-suited to recognizing any sort of relational pattern which manifests few symmetries. Both spoken utterances and real-world scenes appear to be in this class. At this point, we have implemented one ACORN processor for the syntax and semantics in speech (SASS) module of Hearsay II. Another ACORN processor has been built for recognizing the occurrence of inferred patterns (abstractions) in pattern learning training data. The abstractions themselves *are* produced by a program called Sprouter which grows a minimal ACORN to recognize all subtemplates common to two or more relational patterns. [5] From these experiences, it seems that ACORNs may provide an effective mechanism for general recognition.

REFERENCES

1. Barrow, H.G., Ambler, A.P., ft Burstall, R.M. Some techniques for recognising structures in pictures. In S. Watanabe (Ed.), Frontiers of pattern recognition. New York: Academic Press, 1972.

2 Colby, KM, Faught, B., ft Parkison, R.C. Pattern-matching rules for the recognition of natural language dialogue expressions. Memo AIM-234. Stanford: Stanford Artificial Intelligence Laboratory, Stanford University, 1974.

3. Feldman, J.A., ft Rovnar, F. An Algol-based associative language. Communications of the ACM, 1969, 12, 439-449.

4. Frost, M. The news service system. Operating Note SAILON 72-2. Stanford: Stanford Artificial Intelligence Laboratory, Stanford University, 1974

5. Hayes-Roth, F. An optimal network representation and other mechanisms for the recognition of structured events. Proceedings of the Second International Joint Conference on Pattern Recognition, 1974.

6. Hayes-Roth, F. The representation of structured events and efficient procedures for their recognition. Pittsburgh: Department of Computer Science, Carnegie-Mellon University, 1974.

7. Hayes-Roth, F., and Mostow, D.J. An automatically compilable recognition network for structured patterns. Pittsburgh: Department of Computer Science, Carnegie-Mellon University, 1975.

8. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Cambridge: MIT Project MAC, 1972.

9. Lesser, V. R, Fennel, R. D, Erman, L. D., ft Reddy, D. R. Organization of the HEARSAY II speech understanding system. Proceedings IEEE Symposium on Speech Understanding, 1974.

10. Miller, P.L. A locally-organized parser for spoken input. Communications of the ACM, 1974, 11, 621-630.

11. Newell, A., Barnett, J., Forgie, J., Green, C, Klatt, D., Licklider, J.C.R., Munson, J., Reddy, R., ft Woods, W. Speech understanding systems: final report of a study group. New York: American Elsevier, 1973.

12. Newell, A. Production systems: models of control structures. In W.C. Chase (Ed), Visual information processing. New York: Academic Press, 1973.

13. Ruhfson, J.F., Derksen, J.A., ft Waldmger, R.J. QA4: a procedural calculus for intuitive reasoning. Menlo Park: Stanford Research Institute, 1972.

14. Woods, W.A. Transition network diagrams for natural language analysis. Communications o[the. ACM, 1970, 13, 591-606.

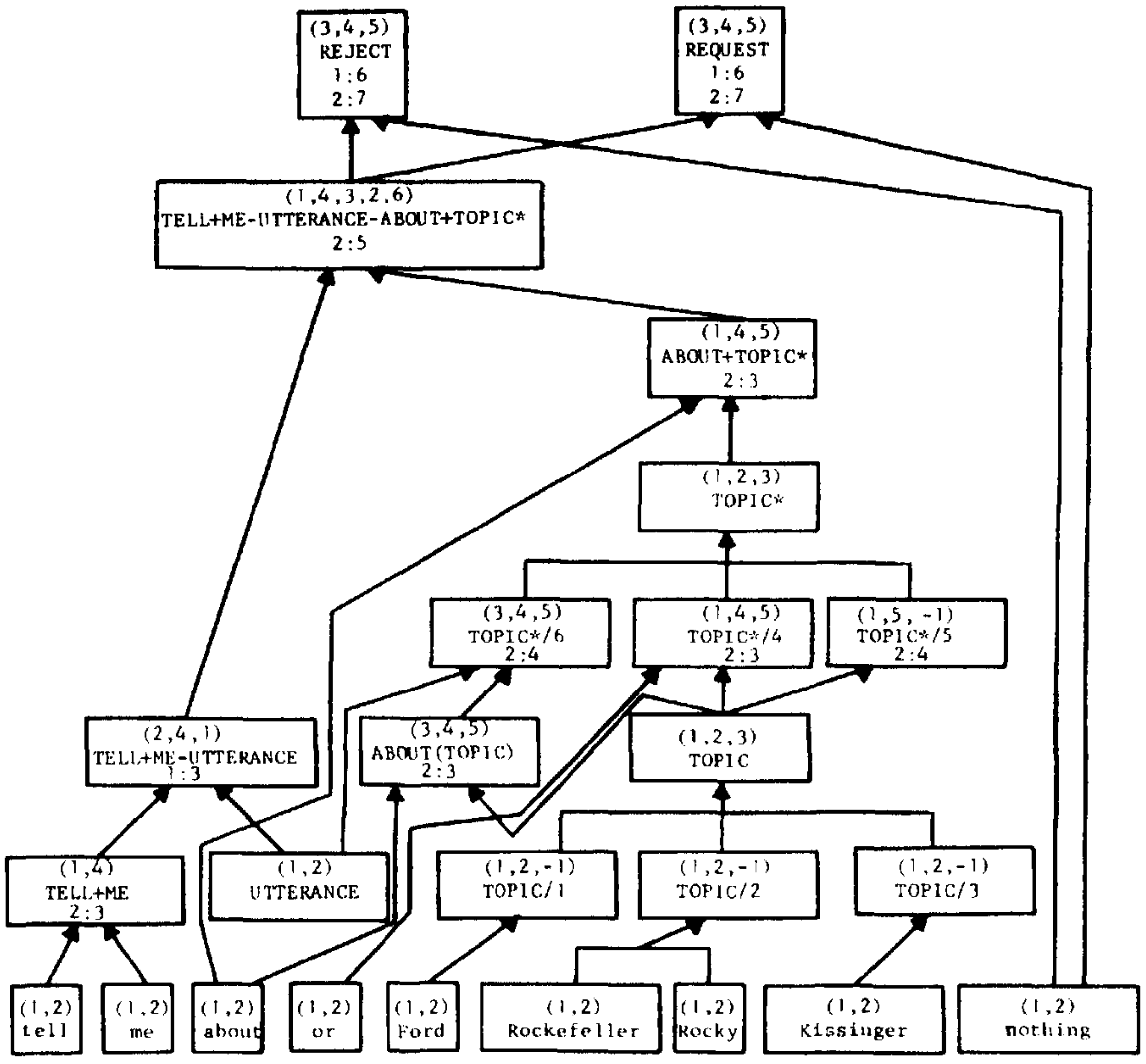

Figure 1.

Sample Recognition Network (an ACORN). See text for an explanation of the tests i:j below the nodes and the generators (i_1, ..., i_k) above them.