SRI International

Technical Note 488 • May 1990

Domain-Independent Task Specification in the TACITUS Natural Language System

Prepared by:

Mabry Tyson
and
Jerry R. Hobbs
Artificial Intelligence Center
Computing and Engineering Sciences Division

The research was funded by the Defense Advanced Research Projects Agency under Office of Naval Research contract N00014-85-C-0013.

Domain-Independent Task Specification in the TACITUS Natural Language System

Mabry Tyson and Jerry R. Hobbs
Artificial Intelligence Center
SRI International

Abstract

Many seemingly very different application tasks for natural language systems can be viewed as a matter of inferring the instance of a prespecified schema from the information in the text and the knowledge base. We have defined and implemented a schema specification and recognition language for the TACITUS natural language system. This effort entailed adding operators sensitive to resource bounds to the first-order predicate calculus accepted by a theorem-prover. We give examples of the use of this schema language in a diagnostic task, an application involving data base entry from messages, and a script recognition task, and we consider further possible developments.

1 Interest Recognition as a Generalization

Natural language discourse functions in human life in a multitude of ways. Its uses in the computers systems of today are much more restricted, but still present us with a seemingly wide variety. Our contention, however, is that beneath this variety one can identify a central core common to most applications. By isolating this core and formalizing it in a concise fashion, one can begin to develop a formal account of the links between a natural language utterance and the roles it plays in the world, as determined by the interests of the hearer. On a practical plane, such an effort allows one to develop a module in which it is possible to specify with significant economy a wide variety of tasks for a natural language system. In this paper we describe our implementation of such a module for the TACITUS natural language system at SRI International.

Processing in the TACITUS system consists of two phases—an interpretation phase and an analysis phase. In the interpretation phase, an initial logical representation is produced for a sentence by parsing and semantic translation. This is then elaborated by a "local pragmatics" component which, in the current implementation, resolves referential expressions, interprets the implicit relation in compound nominals, resolves some syntactic ambiguities, and expands metonymies, and in the future will solve other local pragmatics problems such as the resolution of quantifier scope ambiguities as well as the recognition of some aspects of discourse structure. This component works by constructing logical expressions and calling on the KADS theorem prover[1] to prove or derive them using a scheme of abductive inference. The theorem prover makes use of axioms in a knowledge base of commonsense and domain knowledge. Except for the domain knowledge in the knowledge base, the interpretation phase is completely domain-independent.[2]

In the analysis phase, the interpreted texts are examined with respect to the system's application or task. Rather than writing specific code to perform the analysis, we have devised a **schema** representation to describe the analysis we wish to do. This declarative approach has allowed us to handle very different analysis tasks without reprogramming. In the knowledge base are named schemas which specify the task and can be used to perform the analysis. These are encoded in a schema representation language which is a small extension of first-order predicate calculus. This language is described in Section 2. In most applications, to perform the required task one has to prove or derive from the knowledge base and the information contained in the interpreted text some logical expression in the schema representation language, stated in terms of canonical predicates, and then produce some output action that is dependent on the proofs of that expression.

In order to investigate the generality of our approach to task specification, we have implemented three seemingly very different tasks involving three very different classes of texts. The first is a diagnostic task performed on the information conveyed in casualty reports, or CASREPS, about breakdowns in mechanical devices on board ships. After the text is interpreted, the user of the system may request a diagnosis of the cause of the problems reported in the message. The schema for this task is described in Section 3.1. The second task is data base entry from text. A news report about a terrorist incident is read and interpreted, and in the analysis phase, the

[1]See Stickel (1982, 1989).

[2]For a detailed description of the interpretation phase, see Hobbs and Martin (1987), and Hobbs et al. (1988).

system extracts information in the text that can be entered into a data base having a particular structure. This application is described in Section 3.2. The third application illustrates our approach to a very common style of text analysis in which the text is taken to instantiate a fairly rigid schema or script. The system seeks to determine exactly how the incidents reported in the texts map into these prior expectations. This mode of analysis is being implemented for RAINFORM messages, which are messages about submarine sightings and pursuits. It is described in Section 3.3.

In Section 4, we briefly discuss future research directions.

Before proceeding, we should note a feature of our representations. Events, conditions, and, more generally, **eventualities** are reified as objects that can have properties. Predicates ending with exclamation points, such as *Adequate*! take such eventualities as their first argument. Whereas *Adequate* (*lube-oil*$_1$) says that the lube oil is adequate, *Adequate*!(e, *lube-oil*$_1$) says that e is the condition of the lube oil's being adequate, or the lube oil's adequacy. These eventualities may or may not exist in the real world. If an eventuality e does exist in the real world, then the formula *Rexists*(e) is true. This is to be distinguished from the existential quantifier $\exists$ which asserts only existence in a Platonic universe, but not in the real world; it asserts only the existence of possible objects. It is possible for the eventualities to exist in modal contexts other than the real world, such as those expressed by the properties *Possible* and *Not-Rexists*.[3]

2 Schemas

A schema is a metalogical expression that is a first-order predicate calculus form annotated by nonlogical operators for search control and resource bounds. The task component of TACITUS parses the schema for these operators and makes repeated calls to the KADS theorem prover on (pure) first-order predicate calculus forms. The two nonlogical operators are PROVING and ENUMERATED-FOR-ALL.

2.1 The PROVING operator

Since the first-order predicate calculus is undecidable, an attempt to prove an arbitrary first-order predicate calculus formula may never terminate. While this limitation is discouraging, people manage to reason effectively

[3]See Hobbs (1985) for an elaboration on this notation.

despite the theoretical limits. In part this is because they limit the effort spent on problems and do the best they can within those limits. Hypotheses are formed based on the information known or determined within the limitations. Further investigation can then be done based on these hypotheses. If that does not pan out, the hypotheses can be rejected. Although full knowledge and proofs are desirable and in some cases necessary, it simply is not always possible.

KADS, our deduction engine, proves formulas in first-order predicate calculus. An oversimplified description of how KADS works is that it first skolemizes the formula, turning existentially quantified variables in goal expressions into free variables and making universally quantified variables into functions (with the free variables as arguments). The prover then tries to find bindings for those free variables that satisfy the resulting formula. If any such set of bindings is found, then the original formula has been proven.

In interpreting natural language texts, a single formula passed to the prover is rarely the entire problem. Interpretation requires a number of such calls. Moreover, the bindings made in a proof often are used by the system later in the interpretation process. If alternative bindings could have been used to prove the formula, then they may be needed later if the first set that was found leads to difficulties. KADS is able to continue to look for a proof and try further alternative variable bindings, even after it has found one valid set.

The nonlogical operator, **PROVING**, is used in controlling the theorem prover. An expression

(PROVING formula effort output-fn)

indicates to the the analysis module that it should instruct the prover to try to prove the formula *formula* using a maximum amount of effort *effort*. The results of that proof are then given to the output function *output-fn* to be processed. The output function typically displays the results to the user but may also, say, update a data base, send a mail message, or perform some other action, depending upon what the user has programmed it to do.

At each iteration in one of the inner loops, the theorem prover checks to see if the level of effort has been exceeded. If so, all sets of bindings that have been found for which the formula is true are returned. If none have been found, the proof has failed. If multiple proofs have been found, the analysis module is given multiple sets of variable bindings.

Our particular implementation allows great latitude in how the effort is described. Two obvious types of effort limitation are possible. One type yields repeatable results; the other does not. An example of the first type would be to express the effort limitations in, say, the number of unifications performed. Given the same axiom set and the same problem, the prover would always return the same results. An example of the second type would be to limit the proof attempt to take only a certain amount of real time. This type of limitation may yield different results on different runs. However, it has the advantage that it is easier to understand for users that are not experts in theorem proving. Since one of the reasons for limiting the deductive effort is to provide a responsive system, this type of limitation is often desirable.

The output function is called when the theorem prover has exhausted its resources or has determined that all the answers have been found. The function is called with the formula that was passed off to the theorem prover, the resources that were allowed, and the list of answers that were returned by the theorem prover. With the KADS theorem prover, each answer contains not only the set of substitutions that were used but also a representation of the proof. However, the output functions that we have needed so far only print messages based upon whether proofs were found and the substitutions required for them. They typically are short formatting functions that call upon another function to extract the substitutions from the answers.

2.2 The ENUMERATED-FOR-ALL Operator

The standard predicate logic quantifiers sometimes seem somewhat unnatural. Rather than simply proving existence, it is often much more natural to find an example. Rather than proving a predicate is true for all possible variables, it is more natural to verify that the predicate is true for all appropriate variable bindings.

Toward this end, we have implemented a quantifier which we call ENUMERATED-FOR-ALL. The syntax of this quantifier is

$$(\textit{ENUMERATED-FOR-ALL variables hypothesis conclusion})$$

The semantics is similar to that of

$$\forall(\textit{variables})\,[\textit{hypothesis} \supset \textit{conclusion}]$$

The difference is that, in the ENUMERATED-FOR-ALL case, the formula

$$\exists(\textit{variables})\ \textit{hypothesis}$$

is passed off to the prover to find all possible variable bindings for which the hypothesis is true. The resulting expression for the ENUMERATED-FOR-ALL would be

$$conclusion_1 \wedge conclusion_2 \wedge \ldots$$

Thus proving the ENUMERATED-FOR-ALL expression is reduced to proving this conjunction.[4]

As a simple example, consider

$$\begin{array}{l} (\mathit{ENUMERATED\text{-}FOR\text{-}ALL}\ (x) \\ \qquad [x = 2 \vee x = 3] \\ \qquad \mathit{Prime}(x)) \end{array}$$

The theorem prover would be called upon to prove

$$\exists (x)\, [x = 2 \vee x = 3]$$

and would return two sets of variable bindings. One would specify that x could be 2 and the other would specify x could be 3.[5] The result is that the ENUMERATED-FOR-ALL expression would be replaced by the expression $\mathit{Prime}(2) \wedge \mathit{Prime}(3)$.

2.3 Combining ENUMERATED-FOR-ALL and PROVING

The ENUMERATED-FOR-ALL and PROVING pseudo-operators can be combined, as in

$$\begin{array}{ll} (\mathit{PROVING}\ (\exists\ \mathit{varlist}_2 & (\mathit{ENUMERATED\text{-}FOR\text{-}ALL} \\ & \quad \mathit{varlist}_1 \\ & \quad (\mathit{PROVING}\ \mathit{hypothesis}\ \mathit{effort}_1\ \mathit{output\text{-}fn}_1) \\ & \quad \mathit{conclusion})) \\ \qquad \mathit{effort}_2 & \\ \qquad \mathit{output\text{-}fn}_2) & \end{array}$$

In this case, the theorem prover finds all satisfying variable binding sets for $\exists\,(\mathit{varlist}_1)\,\mathit{hypothesis}$ that it can within the bounds of effort_1. When the prover finishes, those sets of bindings are then passed to $\mathit{output\text{-}fn}_1$ and also applied to *conclusion*, and the conjunction of the resulting forms is then proved within the limitations of effort_2. Finally the bindings found in these proofs are processed by $\mathit{output\text{-}fn}_2$.

[4] This is also similar to Moore's restrictions on quantifiers (Moore, 1981).

[5] Note that each of $[2 = 2 \vee 2 = 3]$ and $[3 = 2 \vee 3 = 3]$ is true.

3 Example Applications

3.1 Diagnosis Task

In the application of the TACITUS system to the analysis of CASREPS, the system is given the domain-specific knowledge of what the various components of the mechanical assemblies are and how they are interconnected, both physically and functionally. The text given to TACITUS generally states the symptoms of the failure and possibly the results of investigations on board. The TACITUS system interprets the text and builds up data structures containing the information gathered from the text. The task component of TACITUS is then called upon to analyze that information.

The schema in Figure 1 is used to process the information. A search is made first for conditions (represented by event variables) that are abnormal but really exist and then for conditions that are normally present but do not really exist. Whether conditions are normal or not is pre-specified in the domain-specific axioms. During the interpretation phase of TACITUS, all conditions that are mentioned in or implied by the text are determined either to really exist or not. However, further deduction may be required during the analysis stage to propagate the existence or nonexistence to other conditions that are not directly mentioned in the text but can be deduced from the state of the world described by the text.

Several details are left out for the sake of clarity. The declaration (not shown) of this schema gives it a name so it can be identified. In this case, this particular schema was specified to be the default one to be done whenever the user asked to analyze the interpretation of the text. When the user asks for analysis, he may specify the name of a different schema to use. Secondly, the specification of the levels of effort have been removed. For instance, *effort*$_1$ is actually

```
(and (time-to-first-proof effort-for-problems)
     (time-to-next-proof (* 0.5 effort-for-problems))
     (ask-user t))
```

which specifies that KADS will be allowed to run on the first problem for an amount of time indicated by *effort-for-problems* if it finds no proof. If it has found a proof, an additional half again as much time will be allowed to find other proofs. If KADS does not find a proof, it will ask the user whether it should continue (if so, it acts as though it has used no resources up to that point). The user may specify the *effort-for-problems* when he asks for an analysis, but the schema declaration includes default values (in this case, 30 seconds).

```
1.  (PROVING
2.    (Some ($e_0$)
3.          (and ;; Look for those events that do exist but shouldn't
4.               (ENUMERATED-FOR-ALL
5.                  ($e_1$)
6.                  (PROVING (and (not (Normal $e_1$)) (Rexists $e_1$))
7.                           $effort_1$
8.                           casreps-problems-shouldnt-exist-print-fn)
9.                  (and (Could-Cause $e_0$ $e_1$)
10.                      (imply (Rexists $e_0$) (Repairable $e_0$))))
11.              ;; Look for those events that don't exist but should
12.              (ENUMERATED-FOR-ALL
13.                 ($e_2$)
14.                 (PROVING (and (not (Rexists $e_2$)) (Normal $e_2$))
15.                          $effort_2$
16.                          casreps-problems-should-exist-print-fn)
17.                 (and (Could-Prohibit $e_0$ $e_2$)
18.                      (imply (Rexists $e_0$) (Repairable $e_0$)))))
19.   $effort_3$
20.   casreps-causes-print-fn)))
```

Figure 1: Schema for the CASREPS Domain

Line 1 indicates that we will be looking for some variable e_0 (of type ev, meaning it is an event variable) that will be the repairable cause of the failure. Lines 6 through 8 are expanded into

$$\exists (e_1)\,[\neg Normal(e_1) \land Rexists(e_1)]$$

which will be passed to the prover with a level of effort $effort_1$. When that level of effort has been expended, the function *casreps-problems-shouldnt-exist-print-fn* informs the users of what conditions exist but normally do not. Then if, say, A and B were found by the prover to be two separate substitutions for e_1 that satisfy the formula, they are substituted into the expression in lines 9 and 10, giving

$$\begin{aligned} & Could\text{-}Cause(e_0, A) \land [Rexists(e_0) \supset Repairable(e_0)] \\ \land\ & Could\text{-}Cause(e_0, B) \land [Rexists(e_0) \supset Repairable(e_0)] \end{aligned}$$

Lines 12 through 18 would be handled similarly. If C and D are found to be valid substitutions for e_2, then the conjunction that begins on line 3 would become

$$\begin{aligned} & \mathit{Could\text{-}Cause}(e_0, A) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \\ \wedge\ & \mathit{Could\text{-}Cause}(e_0, B) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \\ \wedge\ & \mathit{Could\text{-}Prohibit}(e_0, C) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \\ \wedge\ & \mathit{Could\text{-}Prohibit}(e_0, D) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \end{aligned}$$

This would then be handed over to KADS with an effort limitation of *effort*$_3$ in the form of

$$\begin{aligned} \exists(e_0)(\ & \mathit{Could\text{-}Cause}(e_0, A) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \\ & \wedge \mathit{Could\text{-}Cause}(e_0, B) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \\ & \wedge \mathit{Could\text{-}Prohibit}(e_0, C) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)] \\ & \wedge \mathit{Could\text{-}Prohibit}(e_0, D) \wedge [\mathit{Rexists}(e_0) \supset \mathit{Repairable}(e_0)]). \end{aligned}$$

Note that we are looking for a single cause for all of the problems. Whatever bindings for e_0 that KADS finds are then printed by *casreps-causes-print-fn*.

The analysis of the text

```
Unable to maintain lube oil pressure to the starting air compressor.
Inspection of oil filter revealed metal particles.
```

results in the display of

```
An eventuality that shouldn't exist but does is
    X425                          (In! X425 metal-58 lube-oil1)
An eventuality that should exist but does not is
    adequate-ness1                (Adequate! adequate-ness1 pressure1)

An eventuality that could cause the problems is
    (Not-Rexists intact-ness1)    (Intact! intact-ness1 bearings1)
```

The output indicates that metal particles were found in the lube oil but should not have been while the pressure of the lube oil was inadequate. The only cause that was found that could explain both problems was that the "intactness" of some bearings didn't really exist, i.e., they were not intact. In the second sentence, the fact that metal particles were in the oil filter was derived in the interpretation phase. (Note that it is not explicit in the sentence.) The step from there to particles being in the oil was performed in the analysis phase.

3.2 Data Base Entry from Messages

Another important application for a natural language understanding system is to extract the information of interest contained in messages and enter it into a data base. As our ability to interpret messages increases, this application will come to take on greater significance. We have been experimenting with an implementation that analyzes news reports and enters specified information about terrorist attacks into a data base.

For example, suppose the sentence is

> Bombs have exploded at the offices of French-owned firms in Catalonia, causing serious damage.

The data base entry generated by the TACITUS system from this is:

Incident Type:	**Bombing**
Incident Country:	**Spain**
Responsible Organization:	—
Target Nationality:	**France**
Target Type:	**Commercial**
Property Damage:	**3**

where 3 is the code for serious damage.

We use a two-part strategy for this task. We first select a set of canonical predicates, corresponding in a one-to-one fashion to the fields in the data base. Thus, among the canonical predicates are *incident-type*, *incident-country*, and so on. The specification of the schema then involves attempting to prove, from the axioms in the knowledge base and the information provided by the interpretation of the sentence, expressions involving these predicates. When such expressions are found, an appropriate action is invoked. For now, we simply print out the result, but in a real system a data base entry routine would be called.

The schema we use is an expanded version of the schema in Figure 2. We first must find all instances e_1 of an incident (with its incident type) that we can find within resource limits $effort_1$. This is done in the hypothesis of the first ENUMERATED-FOR-ALL, lines 3 - 6. For each such e_1, we must see whether any of the canonical predicates expressing data base entries can be inferred. This happens in the calls to PROVING in lines 9-12, 15-18, etc. The dots in line 20 stand for further calls to prove expressions involving canonical predicates. For every such entry found, a call is made to the appropriate print function. A data base entry function could be placed here as well. The conclusions for the ENUMERATED-FOR-ALLs are all *TRUE*, because once

```
1.  (PROVING
2.    (ENUMERATED-FOR-ALL (e1)
3.      (PROVING
4.        (Some (it) (incident-type e1 it))
5.        effort1
6.        print-incident)
7.      (and
8.        (ENUMERATED-FOR-ALL (it)
9.          (PROVING
10.           (incident-type e1 it)
11.           effort1
12.           print-incident-type)
13.         TRUE)
14.       (ENUMERATED-FOR-ALL (tt)
15.         (PROVING
16.           (target-type e1 tt)
17.           effort1
18.           print-target-type)
19.         TRUE)
20.       ...))
21.     effort2
22.     print-sentence-finished)
```

Figure 2: Schema for the Data Base Domain

we print the information, there is nothing further we need to do with it in this application.

The link between the way people express themselves in messages and what the data base entry routines require is mediated by axioms. Among the axioms required for the above example are the following:

$$\begin{aligned} &\forall (B, E, E_3) \\ &\qquad \mathit{Bomb!}(E_3, B) \wedge \mathit{Explode!}(E, B) \wedge \mathit{Rexists}(E) \\ &\quad \supset \mathit{Incident\text{-}type}(E, \mathit{BOMB}) \end{aligned}$$

If B is a bomb and E is the event of its exploding and E really exists in the real world, then the incident type of E is $BOMB$.

$$\begin{array}{l} \forall (E_4, E, E_3, X) \\ \quad At!(E_4, E, X) \wedge Bomb!(E_3, B) \wedge Explode!(E, B) \wedge Rexists(E) \\ \supset \exists (E_5)\ Target!(E_5, X, E) \end{array}$$

If a bomb explodes at X, then X is the target of the exploding incident.

From such axioms as these we can show, for example, that since the firms are owned by the French, the offices are, and since the offices are, France is the target nationality.

The method for implementing a data base entry application is therefore first to construct a schema such as the one above, and then to define axioms that encode the relationships between these canonical predicates and the English words used in the message, or their corresponding predicates, and other predicates that occur in the axioms in the knowledge base. After the interpretation component has interpreted the message, the information in this interpretation and the axioms in the knowledge base are used to infer the canonical expressions in the schema.

3.3 Schema or Script Instantiation

Many times the texts of interest are very stylized or describe events or conditions that are very stereotypical. Traditionally in AI, researchers have used schemas or scripts in situations like this. "Understanding" the text is taken to mean determining how the described events instantiate the schema.[6]

We have begun to examine what are called RAINFORM messages with this kind of processing in mind. RAINFORM messages describe the sighting and pursuit of enemy submarines. A sample is the following:

> Visual sighting of periscope followed by attack with ASROC and torpedoes. Submarine went sinker.

The sequences of events described by these messages are generally very similar. A ship sights an enemy submarine or ship, approaches it, and attacks it, and the enemy vessel either counterattacks or tries to flee; in either case there may be damage, and in the latter case the enemy may escape.

For our purposes, we will assume the task is simply to show how the events described instantiate this schema, although in a real application we would want then to perform some further action. This task is, in a way,

[6]See, for example, Schank and Abelson (1977).

very similar to the data base entry task. We can describe the different steps of the schema in terms of canonical predicates and then try to infer these expressions.

One important use schemas or scripts have been put to is in the assumption of default values. Thus, the message might say, "Radar contact gained." Here the assumption would be that contact was with an enemy vessel. Our schema recognition module, working in conjunction with the abductive inference scheme in KADS, would handle this by attaching an assumability cost to parts of the schema. Then if it could not be proven within certain resources, it could simply be assumed.

4 Future Directions

We have worked out on paper the schemas for specifying two further tasks, in more or less detail–the first in more, the second in less. The first task is the translation of instructions for carrying out a procedure into a program in some formal or programming language. In structure, this resembles the data base entry task. The canonical predicates correspond to the constructions the target language makes available; the schema encodes the syntax of the target language; and axioms mediate between English expressions and target language constructs. It is interesting to speculate whether this approach could be extended to the case in which the target language is another natural language.

The second task is relating an utterance to a presumed plan of the speaker.[7] This bears a greater resemblance to the diagnostic task. Very roughly, for an utterance that is pragmatically an assertion, we must prove that there is, as a possible subgoal in the plan the speaker is presumed to be executing, the goal for the hearer to know the information that is asserted in the utterance. In doing this, we establish the relation of the utterance to that plan. Utterances that are pragmatically interrogatives and imperatives can be similarly characterized. One needs, of course, to have the axioms that will allow the system to reason about the speaker's plan.

Another area of future research we intend to pursue involves abolishing the current distinction in the TACITUS system between interpretation and analysis. In people, interpretation is interest-driven. We often hear only what we need to or what we want to. Our interests color our interpretations. Currently, interpretation in TACITUS amounts to proving a logical

[7]See, for example, Cohen and Perrault (1979) and Perrault and Allen (1980).

expression closely related to the logical form of the sentence, by means of an abductive inference scheme which is an extension of deduction. In this paper we have shown how schema recognition can be viewed in a very similar light. Therefore, we ought to be able to merge the two phases by attempting to prove the *conjunction* of the interpretation expression and the schema formula. Then the best interpretation of the text will no longer be the one that solves merely the *linguistic* problems most economically, but the one that solves those and at the same time relates the text to the hearer's interests most economically. Of course, many details need to be worked out before this idea turns into an implementation. Nevertheless, the intuition behind it—that to interpret an utterance is to integrate its information in the simplest and most coherent fashion with the rest of what one knows and cares about—seems right.

Acknowledgments

The authors have profited from discussions with Mark Stickel, Douglas Appelt, Douglas Edwards, and Douglas Moran about this work. The research was funded by the Defense Advanced Research Projects Agency under Office of Naval Research contract N00014-85-C-0013.

References

[1] Cohen, Philip, and C. Raymond Perrault, 1979. "Elements of a Plan-based Theory of Speech Acts", *Cognitive Science*, Vol. 3, No. 3, pp. 177-212.

[2] Hobbs, Jerry R., 1985. "Ontological Promiscuity", *Proceedings, 23rd Annual Meeting of the Association for Computational Linguistics*, pp. 61-69. Chicago, Illinois, July 1985.

[3] Hobbs, Jerry R., and Paul Martin 1987. "Local Pragmatics". *Proceedings, International Joint Conference on Artificial Intelligence*, pp. 520-523. Milano, Italy, August 1987.

[4] Hobbs, Jerry R., Mark Stickel, Paul Martin, and Douglas Edwards, 1988. "Interpretation as Abduction", to appear in *Proceedings, 26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, New York, June 1988.

[5] Moore, Robert C., 1981. "Problems in Logical Form", *Proceedings, 19th Annual Meeting of the Association for Computational Linguistics*, Stanford, California, pp. 117-124.

[6] Perrault, C. Raymond, and James F. Allen, 1980. "A Plan-Based Analysis of Indirect Speech Acts", *American Journal of Computational Linguistics*, Vol. 6, No. 3-4, pp. 167-182. (July-December).

[7] Schank, Roger, and Robert Abelson, 1977. *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum Associates. Inc., Hillsdale, New Jersey.

[8] Stickel, Mark E., 1982. "A Nonclausal Connection-Graph Theorem-Proving Program", *Proceedings, AAAI-82 National Conference on Artificial Intelligence*, Pittsburgh, Pennsylvania, pp. 229-233.

[9] Stickel, Mark E., 1989. "A Prolog Technology Theorem Prover: A New Exposition and Implementation in Prolog", Technical Note No. 464, SRI International, Menlo Park, California.