Formal Methods in the Design of Question-Answering Systems[1]

Erik Sandewall
Department of Computer Sciences,
University of Uppsala, Sweden

Recommended by N. Nilsson

ABSTRACT

This paper contributes to the discussion whether and how predicate calculus should be used as a deep structure in question-answering programs. The first part of the paper stresses that there are several possible ways of using predicate calculus, and argues that predicate calculus has significant advantages above competing deep structures if the way of using it is carefully selected. The second half gives hints on how various natural-language constructions can be encoded in a consistent way, and how axiom sets that define these encodings can be written and debugged.

1. The Rôle of a "Deep Structure" in Question-Answering Systems

The following are the major tasks that have to be performed in a computer question–answering system:

1. *Input translation.* This process performs morphological analysis, and a simple syntactic analysis which picks out important substructures (noun phrases, subordinate sentences, etc.). It also identifies words in the sentence with previous occurrences of the same word (represented e.g. as a node in the data base). Input translation may access the data base, so that the data base is used as a lexicon, but it does not integrate the sentence into the data base.

2. *Assimilation.* This step links the new sentence to the existing data base, i.e. it evaluates references to previous sentences, such as "that man", "his father", "for this reason", etc. A *sentence* is then simply defined as that which the input translation phase takes in one bite.

3. *Acceptance.* This step is only performed for assertions. It checks whether the new sentence can be added to the data base (checks for well-formed-ness, contradiction with previous facts, etc.). Possibly, the acceptance

[1] The research reported here was supported in part by the Swedish Natural Science Research Council (contract Dnr 2654–3).

Artificial Intelligence **2** (1971), 129–145

procedure may throw out some old facts from the data base to be able to accommodate the new fact without contradiction ("the system changes its mind").

4. *Answer-finding.* This step is only performed for questions. For closed questions (which can be answered by "Yes" or "No"), we search through the data base, to check whether the proposition of the given question follows from or contradicts the data base. In the first case, the answer is *yes*, in the second case, the answer is *no*. If neither consequence nor contradiction is found (because of insufficient data in the data base, or insufficient resources to carry on the search to the end), the answer is "*Don't know*". (If there is both consequence and contradiction, the data base must be modified). For open questions (WH questions), a similar search is done, and a similar analysis holds.

Other steps may sometimes be needed (e.g. output translation of answers to questions), but the above four steps are the most essential ones. I do not wish to imply that the "sizes" of the four steps given here are approximately equal, nor that these steps need to be done sequentially. For example, it may sometimes be suitable to do some assimilation before input translation has been completed, and in order to guide its course. The four steps are simply four different tasks that have to be done.

The overriding purpose of the input translation task is to transform the sentence to a form where assimilation, acceptance, and answer-finding can be performed more safely and conveniently than in the original sentence. We shall use the term "deep structure" for this form, i.e. for the result of the input translation phase. The deep structure which is used in transformational grammars is certainly one candidate for the deep structure of a question–answering system, but we must recognize that a computer scientist must select his deep structure on different criteria from those of the linguist. In a question-answering system, input translation is an investment of work which is expected to pay off in later steps. A more sophisticated deep structure, which requires more expensive input translation, is therefore warranted if and only if it simplifies later processing at least as much.

It follows, also, that the decision on exactly what is to be done during the input translation phase is a question of efficiency: in some systems, it is perhaps not efficient to reduce equivalent sentences (e.g. active and passive form) to a common, canonical deep structure, but better to verify the equivalence by logical deduction in the question-answering (internal processing) phase. If this is so, then it would be a reason for *not* using the deep structure of transformational grammars (as usually designed today) in such question-answering systems.

We can now make an important observation. Assimilation, acceptance, and answer-finding can be programmed quite easily if the deep structure (i.e.

the final result of input translation) and the components of the data base are formulas in first-order predicate calculus. We can then use existing methods in computational logic as a major subroutine in all these steps. The most important method is the resolution method; see Robinson [1] or Nilsson [2]. Using it, answer-finding for closed questions becomes trivial, and there already exist methods (Green [3]; Luckham and Nilsson [4]) which rely on resolution, and which answer open questions. Assimilation obviously consists of answering open questions (plus some administration), and acceptance consists mostly in verifying that the answer to certain closed questions is "I don't know". (At least, a simple form of acceptance tests can be performed in this fashion. This is further discussed in Palme [5]). Therefore, programming assimilation, acceptance, and answer-finding is facilitated considerably if we express the deep structure in first-order predicate calculus, and set up the necessary axioms. The present paper contributes to the discussion whether and how predicate calculus can be used in this way, and what the advantages and disadvantages would be.

2. Some Possible Ways of Using Predicate Calculus as Deep Structure

Unfortunately, there are misunderstandings current, as regards what can be done in predicate calculus. For example, in a recent working paper from a renowned university, we find the following statement: "The point is that in English, sentences often refer to other sentences, that this is a feature of meaning as well as syntax, and that first-order predicate calculus, which does not involve such references, fails to provide a translation for such sentences." This is not a singular example, although it is an unusually precise formulation; other authors often treat the matter with more handwaving. In this section, we shall show that the statement in the above quotation is wrong.

The heart of the matter is that the predicate calculus notation (like any notation) is only syntax, and that it remains for the user to define its semantics. This can be done in several different ways. Let us illustrate this with three possible translations into the predicate calculus of the sentence "Peter gives Fido to Mary":

1. *Verb is used as predicate symbol.* The translation is

$$\text{Gives(peter, fido, mary)}$$

where *Gives* is a three-place relation, and *peter*, *fido*, and *mary* are constants. Possibly, more arguments can be added to indicate when and where the action takes place.

2. *Prepositions and cases are used as predicate symbols.* The translation is

$$(\exists e)\ \text{Subject(peter, } e) \wedge \text{Verb(giving, } e) \wedge \text{Object(fido, } e) \wedge \text{Indirobject(mary, } e) \wedge \text{Holds}(e)$$

where e is interpreted as the event where Peter gives Fido to Mary. The first

four relations are used to describe this event, and the predicate *Holds* is used to state that the event does in fact take place. Thus in the translation of "John believes that Peter gives Fido to Mary", we would mark the top-level sentence, but not the subordinate sentence with *Holds*:

$$(\exists e)\ (\exists esub)\ \text{Subject}(\text{john},e) \wedge \text{Verb}(\text{believing},e) \wedge$$
$$\text{Object}(esub,e) \wedge \text{Holds}(e) \wedge$$
$$\text{Subject}(\text{peter},esub) \wedge \text{Verb}(\text{giving},esub) \wedge$$
$$\text{Object}(\text{fido},esub) \wedge \text{Indirobject}(\text{mary},esub).$$

3. *Prepositions and cases are used as function symbols.* The translation is

IS(peter, TO(OBJ(giving,fido), mary))

or (if the relation *IS* and the functions *TO* and *OBJ* are written infixed)

(peter IS ((giving OBJ fido) TO mary)).

This expression can be drawn as a binary tree (Fig. 1):

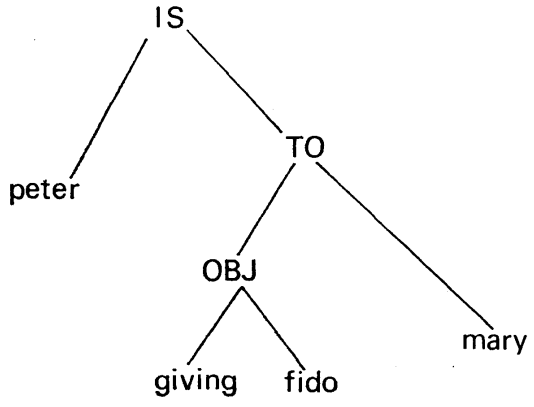

FIG. 1.

In this translation, as in the preceding one, we have two sorts: *objects* (peter, fido, mary), and *properties* (giving). The relation *IS* assumes its first argument to be an object, and its second argument to be a property. (It is a relation on objects times properties). Thus

peter IS giving

is a well-formed formula, and is interpreted as saying that Peter is giving something to somebody. Moreover, *TO* and *OBJ* are functions which map properties times objects into properties.

We shall deal with details in a later section. The point here is that all three translations are possible, and all three are syntactically proper predicate calculus. Admittedly, the later two translations may be interpreted as reformulations in predicate calculus of some other calculus. From a practical viewpoint, however, the important thing is whether existing methods and programs for computational logic are or are not applicable to the selected formalism. Since all three translations permit this, it is meaningful to talk about them as predicate calculus translations.

The first translation method is the most common one. It has been used,

e.g. by Bohnert and Backer [6] and Green [3]. The reason why it is so popular is probably that people easily identify the "predicate" concept of traditional grammar with the "predicate" concept of predicate calculus. However, this translation method also has some obvious limitations, which were presumably thought to apply to predicate calculus as a whole in the statement quoted above.

The second translation method has been used in several question-answering programs (see, e.g. Simmons [7] or Palme [5]). It also corresponds closely to Fillmore's [8] case grammar. As we have seen, this approach does permit one sentence to refer to another sentence (or, to be precise, to the event described in another sentence).

The third method bears some resemblance to the deep structure in transformational grammars. It is interesting to compare the tree in Fig. 1 with the conventional phrase-marker for the same natural-language sentence. Our tree is more compact, since non-terminal nodes are used to convey information *in* the sentence, rather than merely information *about* the sentence. Nested triples of roughly this form have been used, among others, by Simmons and Burger [9].

All three approaches have been tried. Unfortunately, the people who follow approaches (2) and (3) generally do not exploit the fact that their deep structure can be thought of in terms of first-order predicate calculus. The only exception I know of is Palme [5].

The second and third approach have another thing in common: they both assume a relatively small number of functions and relations, which serve to express primitives of our conceptual and linguistic framework (the relations *Subject*, *Verb*, etc. and the functions *OBJ*, *TO*, etc.). Dictionary items, on the other hand, are considered as constants in the calculus. This is in contrast to the first approach, where the list of functions and relations is necessarily very long, and open-ended. Therefore, only the second and third approach permit us to quantify over dictionary items.

I believe these are sufficient reasons to decide against using the first (verb-into-predicate-symbol) approach except for the most trivial programs. In the choice between the second and third approaches (and other possible approaches), I have no fixed opinion. Both are used and investigated in present research projects in Uppsala and Stockholm.

3. Advantages and Disadvantages of Axiomatic Deep Structures

One advantage of using predicate calculus for deep structure was stressed in the first section above: it enables us to use existing computational methods. However, predicate calculus has some further advantages, which it shares with formal systems (i.e. with axiomatization) in general. In this section, we

shall discuss those further advantages, and also say something about the disadvantages of predicate calculus deep structures.

The computer world abounds with data structures which are supposed to be useful for re-expressing "semantic information", "natural language", "deep structure", or whatever other synonym is used. Usually, it is easy and entertaining to set up such a data structure, and to write a program which translates some fragment of natural language into it.

The next problem, of course, is: how do we retrieve information from the data base that we have set up? The usual approach is to "write a retrieval program". In other words, one writes an answer-finding routine which first checks whether the answer to the question is explicitly stored in the data base, and which otherwise searches the data base, combining the given question plus some information in the data base into a sub-question, which is recursively given to the answer-finding routine. Such a retrieval program is by necessity rather *ad hoc*: its designer has to sit down, think, and decide which sub-questions are adequate for a given question.

This ad-hoc-ness is a disadvantage at the design stage: writing the retrieval program would be easier if one had some fixed guide-lines or some mathematical method by which to proceed. It gets to be an even bigger disadvantage when the retrieval program is to be tested and improved. For, suppose the retrieval program answers "Don't know" to a question, when the human user thinks the data base has all the information that was needed for answering the question by "Yes". Who is to blame? Is the retrieval program too dumb, so that more search time or more sophisticated search, or perhaps more sub-questions (higher branching factor) would solve the problem? Or could it be that the data structure is in some sense "insufficient", i.e. that the "information content" that remains after the input translation has added some entropy, is in fact too small to give a positive answer to the question?

One way to solve this problem is to write axioms. The design of the retrieval procedure is then performed in two steps:

(1) Formulate axioms which express *what conclusions can be drawn* from a certain fragment of the data base. An example of an axiom is

"if x is greater than y, and y is greater than z, then x is greater than z"

where each of the assumptions and the conclusion must be expressed in the notation of the selected data base.

(2) Use the axioms as a *guide-line for the retrieval program.* Every sub-question generator (if the retrieval program has such a structure) is then viewed as the implementation of an axiom. In a very simple case, an axiom of the form

$$A \wedge B \supset C$$

may be implemented as a piece of code which recognizes the internal

question or sub-question C, combines it with the explicit fact A in the data base, and generates the sub-question B.

If this approach is followed, and the resulting retrieval program cannot answer one particular question, there is a constructive action to take: we sit down with the date base and the given question, and try to prove the question statement *manually* from the information in the data base. If this succeeds, then obviously the retrieval program is to blame, and we have probably also gotten some hint of what one should change in it. If we do not succeed (and if we consider ourselves to be perfect theorem provers for small, test-case data bases), we must clearly add some more axioms. In attempting this, it could happen that every proposed axiom which would enable the system to answer Yes to the test-example question, also enables it to answer positively to questions which we (intuitively) understand are not supported by facts in the data base. If this is so, we can conclude that the data structure itself is inadequate for its intended purpose.

In this discussion, I have purposely avoided using the terminology of mathematical logic, because the merit of axiomatization needs mostly to be explained to those readers who do not know logic. It is clear, however, that anybody who intends to use axiomatization as a method for programming a retrieval program should learn at least the elements of logic, and familiarize himself with concepts such as interpretation, validity, completeness, etc.

If some usage of logical terminology is permitted here, let me add that a reasonable third step in the design of a retrieval procedure is

(3) Analyze the proposed retrieval program to determine whether it is sound (i.e. whether it only answers positively to questions which can be proved from the data base, using given axioms) and complete (i.e. whether it will answer positively to all questions which can be proved from the data base).

Such an analysis is something quite easy to do. Sandewall [10] does it for the "chaining" method in a data base of binary relations between nodes.

The axiomatization method is particularly convenient if the data base is expressed in predicate calculus notation. One advantage is that we then have a fixed external notation which adequately describes fragments of the data structure. (This is not necessarily the case when we use "home-made" structures.) Another advantage is that there is at least some knowledge available on when and how axioms should be written. A third advantage is that some axioms (namely those handling logical connectives and quantifiers) are standard, and can be taken from "the book". (To some extent, they are even built into the existing deduction methods.) If one uses his own data structure, he has to set up and debug these axioms himself.

The major advantage is, however, that methods which will do the retrieval for us already exist. This remark will immediately be qualified: these methods

exist in formal descriptions, and they have been tried on very small data bases. The primary method is of course the resolution method (Robinson [1]; Nilsson [2]). Deduction by resolution, starting from a small number of axioms can be done rather efficiently today, and the problem of how one should use resolution for (relatively) shallow deductions from a large data base is an important research topic. One can expect that much more will be known even before this paper is published. For further details about this, the reader is referred to Nilsson's excellent book [2].

The major *disadvantage* in using predicate calculus for question-answering today is that these techniques for doing shallow resolution deductions in a relatively large data base, have not yet been developed. In particular, we do not know how to exploit the semantic content of the formulas to direct the search for proof or refutation. On the other hand, hardly any research has been devoted to this problem, and I believe that techniques of this kind are within reach.

4. Encoding Natural-Language Constructions in Predicate Calculus

If we accept that predicate calculus should be used in the data base of a question-answering program, we must ask next what the major sub-problems are when this approach is used. The following seem to be the most important ones:

(a) Formulate pieces of our (linguistically based) conceptual framework in the selected version of predicate calculus. This includes concepts of time, space, cause-effect, adjective-noun composition, adjective comparison, etc.

(b) Formulate axioms which express the properties of the various concepts that were formalized under (a).

(c) Adapt the methods of computational logic for use in the data base of a question-answering system.

This section will give some hints on how task (a) can be performed, and the next section will deal with task (b). I am presently working on an approach to task (c).

One major rule-of-thumb for the concept formulation task was given in the last section, namely: write your axioms (or your retrieval program, in case you do not wish to axiomatize) in parallel with concept formulation, not after. They give you an important guide-line for selecting a useful notation.

Another rule is: do not attempt to formulate "raw" natural language constructions, but instead idealized constructions. For example, it would be senseless to look for axioms which characterize adjective-noun or noun-noun combinations in general. Instead, one should assume that such combinations

are ambiguous and that there are several underlying types of combination, which can be axiomatized separately. Examples:

"the red house"	(it is red, and it is a house)
"the little elephant"	(it is little *for an* elephant, but it may be big for an animal)
"the bad teacher"	(he is bad *as* teacher, but he may be good as father).

A third rule is: distinguish carefully between *factual* elements in the deep structure, which express new facts or actual questions, on one hand, and *referential* elements which serve for reference to previous text, on the other. Factual elements remain intact after the assimilation phase (cf. Section 1), whereas referential elements are eliminated in assimilation. Factual elements must be characterized by axioms; referential elements must be characterized by clear rules which define how these elements are to be used during assimilation.

We now show some examples which illustrate these rules of thumb, and which also may have some interest in themselves. We prefer to use the third approach of Section 2 (prepositions and cases used as function symbols), mainly because formulas tend to be much more legible than in the second approach (binary relations throughout). Most of the examples are taken from a recent paper (Sandewall [11]).

First some preliminaries:

Sorts. We need individuals of at least three sorts: *objects*, *properties*, and *events*. The individuals *peter*, *fido*, and *mary* are objects; the individual *giving* (in the example in a previous section) is a property. Event individuals will be introduced immediately.

Functions and relations all have one or two arguments. If there is one argument, the function (relation) is prefixed; if there are two arguments, it is infixed. For every function, the sort of each argument, and of the value is specified. For every relation, the sort of each argument is specified. This enables us to parse in an unambiguous fashion expressions which would otherwise be ambiguous. For example, the expression

peter **IS** giving OBJ fido TO mary

is unambiguous if **IS**, OBJ, and TO have the following sorts:

IS:	objects × properties
OBJ:	properties × objects → properties
TO:	properties × objects → properties

The construction of a simple algorithm (similar to precedence analysis) which does infix-to-prefix translation using sort restraints is an interesting programming problem.

We can now proceed to the discussion of some essential natural-language constructions.

Representation of attributes. For expressing, e.g. "John is a father" and 'John is the father of Peter", we assume a property "father", and a function

OF: properties × objects → properties.

The two examples can then be expressed as

john **IS** father

and

john **IS** father OF peter.

The same conventions and the same function OF can be used for other similar constructions, such as "son of", "telephone number of", and so on.

Representation of sentence kernels. These are handled like the example in Section 2 ("Peter gives Fido to Mary").

Representation of subordinate sentences. We introduce the functions

IZ: objects × properties → events

THAT: properties × events → properties.

These functions are used as in the following example: We express "Dick believes that John is the father of Peter" by

dick **IS** believing THAT (john IZ father OF peter).

The difference between **IS** and IZ is of course that **IS** is a *relation*, whereas IZ is a *function*, whose value (the 'event') is presumed to retain the information contained in its first and second argument. (In Sandewall [11], the function IZ is called WERE.)

Up to this point, we have treated factual constructions. Let us give an example of a referential construction as well:

A "definite article". In the expression which is generated by input translation, we permit sub-expressions of the form

The *p*

where *p* is an expression of the sort 'property'. We specify the following assimilation procedure:

(a) If the data base contains exactly one object *m* for which it is known (i.e. stated or deducible)

m **IS** *p*

then *m* is inserted instead of 'The *p*' in the input expression.

(b) If the data base contains no such *m*, then a new constant *m* is introduced, and the expression

m **IS** *p*

is considered for acceptance in the data base.

(If this expression contradicts the data base, then the acceptance routine is supposed to protest. This would presumably occur, e.g. if we try to introduce

m **IS** father OF matterhorn

when the data base contains information about the Matterhorn and about fathership).

(c) If the data base contains several such m, then a question is directed to the user: which p do you mean?

This very simple procedure obviously does not describe the actual usage of the definite article in natural language, and it is not intended to. It is, instead, a description of a logical operator.[2] For mnemonic reasons, we chose to call the operator "The", rather than "h_0" or "α", but the name is arbitrary. In a practical system, we would like to have a number of operators similar to "The". One of the challenges to the input translation module is then that it should be able to identify which of the logical operators is intended in one particular occurrence of a definite article in a sentence.

It is interesting to notice that assimilation of the operator "The" should not be performed if our system is to be used for automatic translation from one natural language to another, with input translation from the source language to predicate calculus, and output translation from predicate calculus to the target language. This is so because one wants to make the same kind of references in the target language as in the source language, although the rules for how various operators are expressed may vary considerably.

Sortwise, "The" is an operator

properties → objects.

This fact is useful in parsing formal expressions such as

The father OF peter **IS** father OF mary

or

321–5678 **IS** telephone-number OF The father OF peter.

The operator Any. One operator that will certainly be useful is

Any: properties → objects

which is used as in

Any boy **IS** male.

The assimilation procedure for "Any" must prescribe that this sentence is to be re-expressed internally as

$(\forall x)$ x **IS** boy $\supset$ x **IS** male.

In general, every sub-expression of the form 'Any p' is changed into a variable v, and the restriction

v **IS** p

is added to the formula. In principle, this expansion could have been done in input translation, since it does not require access to the data base and the

[2] We call it a logical operator, not a function. In fact, since we chose to use the term "deep structure" for the result of the input translation (before assimilation), and since it is the data base and the retrieval requests (after assimilation) that need to be in predicate calculus, we have been slightly imprecise when we proposed to "use predicate calculus for deep structure". Possibly, it would be a good idea to distinguish between deep structure *in transit* (before assimilation) and deep structure *in corpore* (after assimilation). The conventional deep structure of transformational grammars is then an *in transit* version, whereas predicate calculus is used *in corpore*. The notation in this section, which includes the operator "The", must then be characterized as a proto-predicate-calculus.

deduction procedures, but it seems more convenient and more natural to do the expansion in the assimilation stage. (One reason for this choice is the connection with automatic translation. We suggested above that the representation of the sentence after input translation but before assimilation should be the "intermediate language". One certainly wants to retain the operator "Any" in this intermediate language.)

Comparison of adjectives. We take for granted that adjectives are expressed as properties, so that we can write e.g.

peter **IS** tall.

To handle comparison, we then introduce functions

ER-THAN: properties × objects → properties
EST-AMONG: properties × properties → properties.

These are to be used as in

peter **IS** tall ER-THAN john
peter **IS** tall EST-AMONG (brother OF dick).

The first of these formulas is of course intended to say "Peter is taller than John", and the second one to say "Peter is the tallest of Dick's brothers". For the sake of the deduction procedure, we must decide on some conventions, e.g. whether the latter formula allows Peter to be one of two equally tall brothers, who both are taller than all the other of Dick's brothers.

We have already remarked that the infix notation for functions which is used in this paper is for the reader's convenience, and that one will probably prefer to have prefix notation internally in the computer's data base. We can increase the external convenience a trifle by permitting 'bifix' functions, so that we can write

More tall THAN john

synonymously with

tall ER-THAN john.

Translating bifix to prefix is no harder than translating infix to prefix, and can be done mechanically by a quite simple program. The bifix notation is excellent e.g. for the function

As . . . AS: properties × objects → properties

used as in

john **IS** As tall AS dick

with the obvious intended meaning.

The previously mentioned paper (Sandewall [11]) contains suggested formalizations of a number of other idealized natural-language constructions.

5. Hints on How to Axiomatize

Strangely enough, very little seems to have been written on this topic, in spite of the fact that formal logic is extensively used in mathematics, and axiomatization is a common preoccupation there. Some axiomatizations of

non-mathematical concepts can be found in McCarthy [12, 13, 14], Green [3], Sandewall [11], and Nilsson [2]. All of these are reports from A.I. projects.

For illustration, let us set up here some axioms which characterize comparison of adjectives as expressed in the data base notation of Section 3 of this paper. The following axioms are useful:

1. m **IS** More p THAN $k \wedge k$ **IS** More p THAN $n \supset$
   m **IS** More p THAN n
2. $\neg$ (m **IS** More p THAN m)
3. m **IS** More p THAN $k \supset m$ **IS** More p THAN Any (As p AS k)
4. m **IS** More p THAN $k \supset$ Any (As p AS m) **IS** More p THAN k
5. m **IS** As p AS $n \supset n$ **IS** As p AS m
6. m **IS** As p AS m
7. m **IS** Most p AMONG $q \equiv$ (m **IS** $q \wedge$ No q **IS** More p THAN m)

where the operator "No" is defined so that any expression

No q **IS** p

is assimilated into

$(\forall m)(\forall p)(\forall q) \quad \neg(m$ **IS** $p \wedge m$ **IS** $q)$

and where the universal quantifier on the variables (m, n, k, p and q) has consistently been omitted. Theorems which can be proved from these axioms include

m **IS** Most p AMONG $q \wedge n$ **IS** More p THAN $m \supset \neg(n$ **IS** $q)$.

The inference expressed in such theorems can therefore be obtained by a sequence of several successive sub-questions in any complete retrieval program based on the axioms 1 to 7.

How does one set up such a set of axioms? The following steps are generally useful:

(a) *Jot down* a number of observations about the environment that you wish to describe (in our example, comparison of adjectives). Be careful to express everything in correct predicate calculus.

(b) *Check for inconsistencies.* When you wrote down the observations, you expressed in a strict fashion your intuitive idea of the environment that is to be formalized. This intuitive idea may have varied from one axiom to the next. For example, in the treatment of comparison of adjectives, different suggested axioms might have expressed contradictory ideas about whether

m **IS** Most p AMONG q

permits some other q to be as p as m is. If you have been inconsistent in a matter like that, your axioms are likely to be inconsistent too.

(c) *Check for redundancy*, i.e. check whether some of the proposed axioms can be proved from some of the others. It is possibly, but not necessarily, a good idea for efficiency to remove redundant axioms. The idea tends to be better if the proof of one observation from the others is short.

(d) *Check for completeness*, i.e. check whether some axioms are missing. The general method to do this is to make up test examples of desired theorems (desired conclusions from the axioms), and to check manually that each of them can in fact be proved.

The test examples may be "positive" or "negative". Positive examples are those which are useful, in a direct way. For example, in an extended adjective comparison system, where one has an **OPPOSITE** relation on properties times properties, used as in

big **OPPOSITE** small

it is reasonable to check that from

m **IS** More *p* THAN *n*
k **IS** More *q* THAN *n*
p **OPPOSITE** *q*

follows

m **IS** More *p* THAN *k*

because this is a deduction that may be needed to answer some questions.

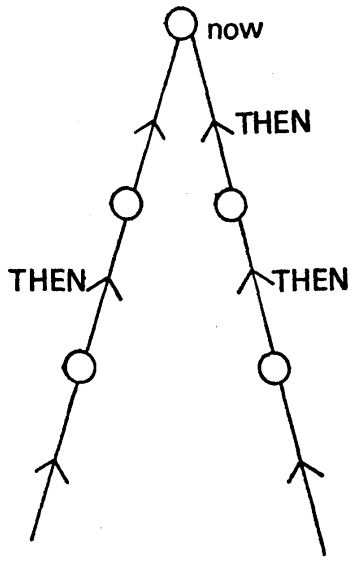

FIG. 2.

In finding negative examples, we look for "legal loopholes", or things that our axioms should outlaw but which they might fail to outlaw. For example, suppose we have agreed to describe time by using a binary relation **THEN** on events, used so that

e **THEN** *d*

says that both *e* and *d* have occurred, and *d* occurred after *e*. Suppose, further, that we have proposed some axioms, such as the obvious transitivity axiom. Then a reasonable "negative example" might be: does our set of axioms permit two converging strands of time, both of which end in "now", but such that neither "*e* **THEN** *d*" nor "*d* **THEN** *e*" holds if *d* and *e* are on opposite

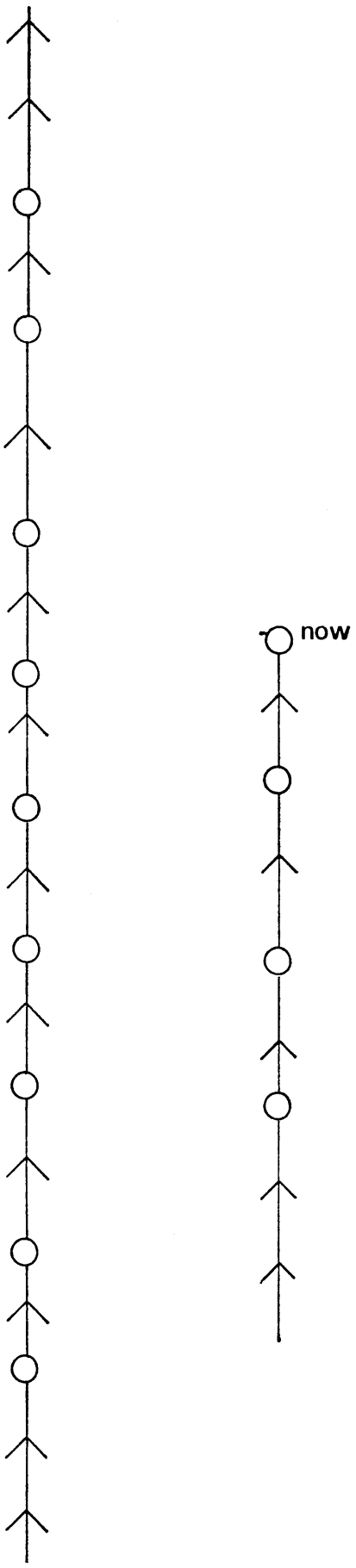

FIG. 3.

strands? (Fig. 2). This test example is of course equivalent to proving that

$$(\exists e)(\exists d) \quad \neg e \text{ THEN } d \wedge \neg d \text{ THEN } e$$

contradicts the axioms, i.e. to proving

$$(\forall e)(\forall d) \quad e \text{ THEN } d \vee d \text{ THEN } e.$$

If our set of axioms passes this test, our next "negative example" might be: does the set of axioms permit two parallel strands of time, one which is finite and ends in "now", and one which is infinite and unconnected to the first strand? (Fig. 3). It is left as an exercise to the reader to rewrite this negative example as a desired theorem.

It is interesting to notice that these four steps have counterparts in the methods for writing computer programs. In particular, the "positive" and "negative" test examples in step *d* correspond to "simulate the program and see if it works", and "look at the program and try to invent some case where it would not work", respectively. But when we compare a set of axioms with a computer program which performs the same task (strictly speaking, performs the task of a resolution-based retrieval program loaded with these axioms), we shall see that the axioms are much easier to write, easier to debug, and easier to integrate into large systems.

For a comparison in another direction, let us point out that if the predicate-calculus notation used in this report is analogous to the deep structure in transformational grammar, then the axioms are analogous to some of the optional transformations. We claim the same advantages for axioms in this comparison: they are easier to write, easier to debug, and more modular than conventional transformational grammars.

REFERENCES

1. Robinson, J. A. A machine oriented logic based on the resolution principle. *J. ACM*, **12** (1) (1965), 23–41.
2. Nilsson, N. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971 (to be published).
3. Green, C. *The Application of Theorem-Proving to Question-Answering Systems*. Ph.D dissertation, Stanford University, Electrical Engineering Dept., June 1969. Also printed as Stanford Artificial Intelligence Project Memo AI–96, June, 1969.
4. Luckham, D. and Nilsson, N. Extracting information from resolution proof trees. *Artificial Intell.*, **2** (1971) 27.
5. Palme, J. Making Computers Understand Natural Language, in *Artificial Intelligence and Heuristic Programming*, N. Findler and B. Meltzer (Eds.), Edinburgh University Press, 1971 (to be published).
6. Bohnert, H. G. and Backer, P. O. Automatic English-to-Logic Translation In a Simplified Model. *Research Paper RC–1744*, IBM Watson Research Center, Yorktown Heights, New York (1967).
7. Simmons, R. F. Computational Linguistics and Question-Answering Systems, in *Artificial Intelligence and Heuristic Programming*, N. Findler and B. Meltzer (Eds.), Edinburgh University Press, 1971 (to be published).
8. Fillmore, Ch. J. The case for case, in *Universals in Linguistic Theory*, E. Back, R. T. Harms (Eds.), Holt, Rinehart, and Winston, Inc., 1968.
9. Simmons, R. F. and Burger, J. F. A semantic analyzer for English sentences. *Mechanical Translation*, **11** (No. 1/2) (1968), 1–13.

10. Sandewall, E. J. A set-oriented Property-Structure Representation for Binary Relations, in *Machine Intelligence*, Vol. 5, B. Meltzer and D. Michie (Eds.), Edinburgh University Press, 1969.
11. Sandewall, E. J. Representing Natural-Language Information in Predicate Calculus, in *Machine Intelligence*, Vol. 6, B. Meltzer and D. Michie (Eds.), Edinburgh University Press, 1970.
12. McCarthy, J. Programs with Common Sense, in *Mechanization of Thought Processes*, Vol. 1, pp. 77–84, Proc. Symposium, National Physical Laboratory, London, Nov. 24–27 1958. Reprinted in *Semantic Information Processing*, M. Minsky (Ed.), MIT Press, Cambridge, Mass., 1968, pp. 403–410.
13. McCarthy, J. Situations, Actions and Causal Laws. Stanford University Artificial Intelligence Project Memo. No. 2, 1963. Reprinted in *Semantic Information Processing*, M. Minsky (Ed.), MIT Press, Cambridge, Mass., 1968, pp. 410–418.
14. McCarthy, J. and Hayes, P. Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence*, Vol. 4, B. Meltzer and D. Michie (Eds.), American Elsevier Publishing Co., Inc., New York, 1969, pp. 463–502.

Accepted, March 1971