A set-oriented property-structure representation for binary relations, SPB

by

Erik J Sandewall

UPPSALA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCES

UPPSALA UNIVERSITY
COMPUTER SCIENCES DEPARTMENT
REPORT NR 24
AUGUST, 1969

also:
SWEDISH QUESTION-ANSWERING PROJECT
MEMO 6

A set-oriented property-structure representation for binary relations, SPB

by

Erik J Sandewall

Paper presented at the Machine Intelligence 5
workshop in Edinburgh, Scotland
September 15 - 20, 1969

0. Introduction

Early question-answering systems often used ad hoc representations for their data bases, and corresponding ad hoc deduction methods for the question answering. Many systems, such as the SIR system ({Raphael 1964a}) represent binary relations on property-lists. This term will be defined below. In recent years, it has been argued (e.g. in {Slagle 1965b} and {Green 1968a}) that a dialect of predicate calculus should be used instead. This would have two advantages: (1) predicate calculus is a richer language, i.e. more things can be said in it; (2) for predicate calculus, one knows reasonably efficient proof procedures, e.g. resolution (see {Robinson 1965a} or {Nilsson 1969a}).

The distinction between these two approaches is not clear-cut, and it is tempting to try to incorporate the use of property-lists to speed up resolution. For example, one may find it useful as the data base grows to construct, for each object symbol c, a chained list of all literals or clauses where c occurs. This chained list is then (in a sense) a property-list for c. However, if we consider it as given that certain information is to be retrieved on property-lists, then it is not obvious that the best way to use this information is to feed it into the resolution black-box. It is natural to look for deduction procedures which utilize efficiently the kind of information that is stored and can be retrieved from property-lists, and which can be used instead of or together with conventional theorem-proving methods.

In this paper, we shall suggest one such alternative, namely a "clean" property structure where binary relations uRv are stored as address references. We shall show how formulas that involve quantifiers can sometimes be expressed simply by giving a certain structure to the "R". We shall also specify a fast deduction procedure for this notation. - An ultimate goal for work along these lines is that property-lists shall no longer be considered as an ad hoc notation.

1. Property-structures and property-lists

The purpose of this section is to define what we mean by a property-structure and a property-list, and to give some notation. It will not contain any new results.

Let $U = \{u,v,w,\ldots\}$ be a set of objects, and let $\Delta = \{P,Q,\ldots\}$ be a set of binary relations on U (i.e. subsets of $U \times U$). Consider the problem to design a computer program which has partial knowledge of these relations; which can accept more information about the relations and store it in a data base, and which can also accept questions about these relations and answer them from the data base. In the simplest case, assertions and questions are given as triples $\langle u,P,v \rangle$, meaning " $\langle u,v \rangle$ is a member of P" viz. "is $\langle u,v \rangle$ a member of P?". We want to organize the data base in such a way that new assertions can be added easily, and answers to questions can be retrieved easily and quickly.

Let us first discuss the use of such systems. Several previous question-answering programs are essentially of this type. In Lindsay's program ({Lindsay 1962a}), U is a set of people and families, and the relations are "u is the husband in the family v", "u is offspring in the family v", etc. Raphael encountered the same problem ({Raphael 1964a}) with U being a set of objects and people, and the relations being e.g. "u is physically part of w", "v is the owner of x", etc. Levien saw it ({Levien 1965a}) with U being a set of people, meetings, institutions, and documents, and relations such as "u is the author of v", "u is employed by w", etc. We met the same kind of problem ourselves when we decided to translate natural-language sentences like "A gives B to C" into an expression

$$(\exists\ x)\ \ R_1(x,A) \wedge R_2(x,\text{Give}) \wedge R_3(x,B) \wedge R_4(x,C)$$

where x is the activity described by the sentence, R_1 can crudely be described as an "activity-to-its-subject" relation, R_2 is an "activity-to-its-verb" relation, etc.

One standard way of representing binary relations in the computer is through property-structures. We formally define a property-structure as a mapping

$$\sigma : \quad U \rightarrow 2^{\Delta \times U}$$

i.e. a mapping which assigns a set of pairs Rv to each member u of $U^{(*)}$. A property-structure σ corresponds to a set φ of facts iff

$$uRv \in \varphi \equiv Rv \in \sigma(u)$$

If σ is a property-structure and $Rv \in \sigma(u)$, we shall say that u has the property Rv in σ.

To represent a property-structure σ in memory, one usually does as follows: a unique cell is associated with each member of U. A cell which is so associated is called an atom. The atom associated with u will itself be called u. A property Rv is represented as an indicator for R plus the address of v. All the properties that an atom u has are stored in such a way that they can be accessed from u as easily as possible. This may be done using a chained list (in which case each $\sigma(u)$ is represented as a property-list), through hash-coding, or by other means.

It is usually necessary to represent each fact through two properties. If $\mathcal{R}$ is the reverse relation of R, then a fact uRv (equivalent to $v\mathcal{R}u$) is represented in a property-structure σ through

$$Rv \in \sigma(u)$$

and

$$\mathcal{R}u \in \sigma(v)$$

If R is symmetric, then R and $\mathcal{R}$ are of course the same relation.

Figure 1 illustrates how $\varphi = \{uPv, vRw\}$ can be represented as a property-list structure. Arrows stand for address references.

Let us contrast the property-structure representation of facts with the linear representation, where the data base consists of a straight list of all facts that have been accumulated. Clearly, the property-structure representation is superior for honoring simple information requests like "is uRv stored (explicitly) in the data base?". However, all interesting question-answering programs must be able to perform logical deduction, i.e. to retrieve facts which are stored implicitly in the data base. It is at least not trivial to

---

(*) This is an abbreviation for $\langle R,v \rangle$. We shall often use this abbreviation of the notation for tuples.

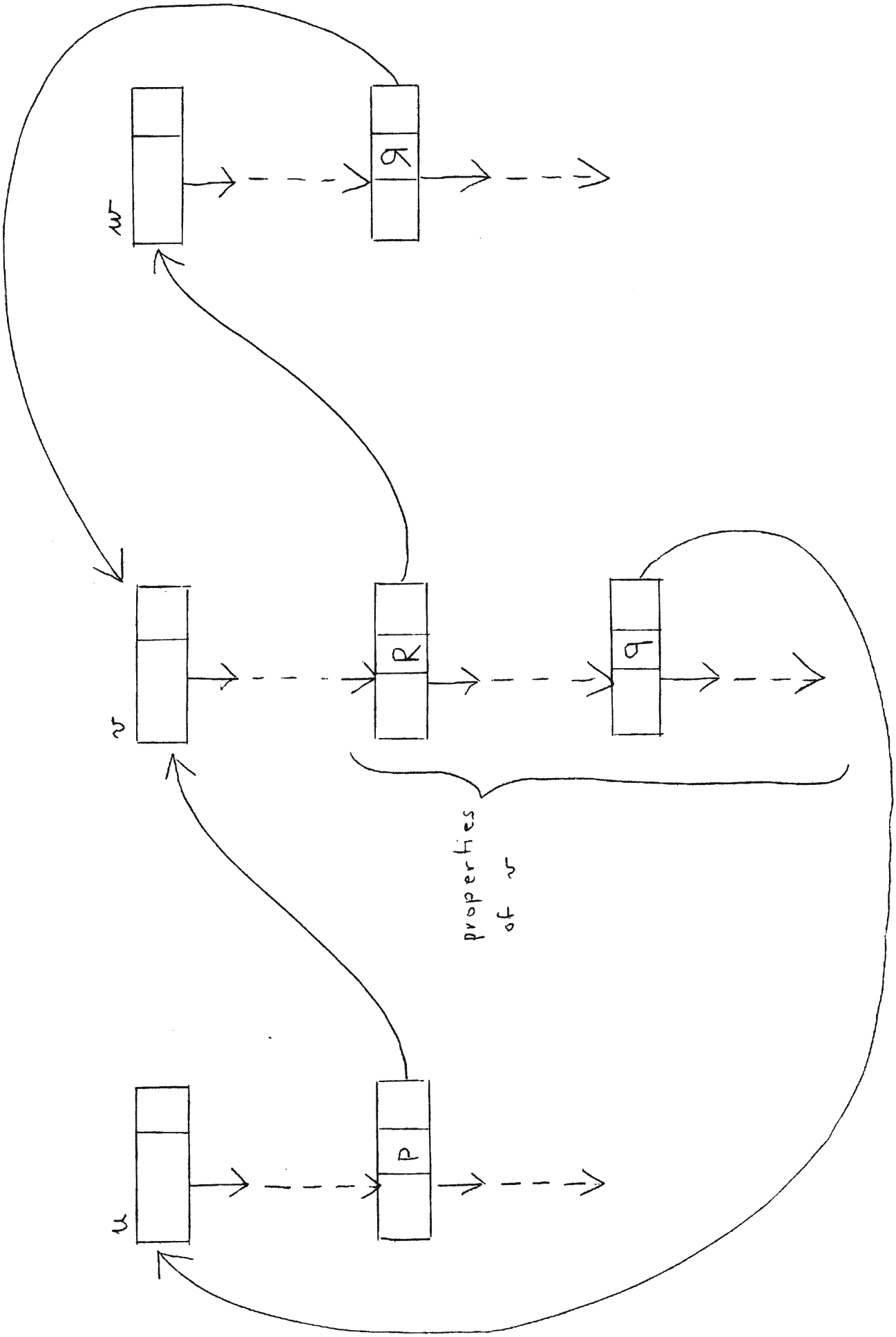

Figure 1.

modify conventional deduction procedures (e.g. resolution) so that they can make efficient use of a property-structure.

However, for certain types of inference rules, one can write deduction procedures which are particularly adapted for a property-structure data base. In particular, this is true about rules of the type

$$uPv\ ,\ vQw \vdash uRw$$

(where P, Q, and R need not be distinct). Such rules shall be called chaining rules, and shall be written

$$P \times Q \rightarrow R$$

Chaining rules can be handled by the following

Linear chaining procedure. To determine whether uRw is stored (explicitly or implicitly) in a property-structure σ, do the following:

(a) See whether $Rw \in \sigma(u)$. If so, exit with success.

(b) For all pairs $\langle P,Q \rangle$ such that $P \times Q \rightarrow R$, and for all v such that u has the property Pv, set up the sub-problem to prove that vQw is stored in σ. If this subproblem has not been attempted before, attempt it now by the linear chaining procedure. - When no new sub-problems remain, exit with failure.

We call this procedure "linear" because it only considers those properties Pv which are explicitly stored under u. A more careful procedure would also consider properties Pv that u can be proved to have. Linear chaining corresponds to linear proofs in resolution. The linear chaining procedure is complete only if certain assumptions are made about the set of chaining rules. This question will be more thoroughly treated in another context.

It is easily seen that if the number of chaining rules is finite, and the number of facts in σ is also finite, then this procedure will always terminate.

To summarize, a major advantage of the property-structure representation is that it permits fast retrieval and fast deduction, in some

simple cases. A major disadvantage is that the notation is so restric restricted: no logical connectives are allowed (except the trivial and which connects all facts in the data base), and no quantifiers or other logical operators can be used.

The notation which we shall define in the next few sections consists essentially of a few operators on the relations P,Q,... , which enable us to express connectives and quantifiers in some cases. Using these operators, each fact would be written uGv, where G has the form

$$A_1(A_2(\ldots A_j(R) \ldots))$$

Such facts can of course easily be stored in a property structure. Besides spedifying a set of interesting operators, we shall give a set of chaining rules

$$F \times G \rightarrow K$$

where F, G, and K have been formed using these operators.

2. Quantification of binary relations

We repeat that $U = \{u,v,w,\ldots\}$ is a set of objects, and $\Delta = \{P,R,\ldots\}$ is a set of binary relations on members of U.

Starting from the relations in Δ, we shall now define further relations on members of U (ground relations) and on subsets of U (set relations). The operators Id, Rev, Neg, Aa, Ae, At, Ea, Ee, and Ta on relations are defined as follows:

$$\begin{aligned}
\mathrm{Id}(R) &= R \\
\mathrm{Rev}(R) &= \lambda(x,y)\ R(y,x) \\
\mathrm{Neg}(R) &= \lambda(x,y)\ \neg R(x,y) \\
\mathrm{Aa}(R) &= \lambda(b,c)\ (\forall\, x\in b)(\forall\, y\in c)\ R(x,y) \\
\mathrm{Ae}(R) &= \lambda(b,c)\ (\exists\, y\in c)(\forall\, x\in b)\ R(x,y) \\
\mathrm{At}(R) &= \lambda(b,c)\ (\forall\, x\in b)(\exists\, y\in c)\ R(x,y) \\
\mathrm{Ea}(R) &= \lambda(b,c)\ (\exists\, x\in b)(\forall\, y\in c)\ R(x,y) \\
\mathrm{Ee}(R) &= \lambda(b,c)\ (\exists\, x\in b)(\exists\, y\in c)\ R(x,y) \\
\mathrm{Ta}(R) &= \lambda(b,c)\ (\forall\, y\in c)(\exists\, x\in b)\ R(x,y)
\end{aligned}$$

In these definitions, we have used the abbreviations

$$(\forall\, x\in b)\ P \equiv (\forall\, x)[x\in b \supset P]$$
$$(\exists\, x\in b)\ P \equiv (\exists\, x)[x\in b \wedge P]$$

We immediately obtain

$$\mathrm{Rev}(\mathrm{Neg}(R)) = \mathrm{Neg}(\mathrm{Rev}(R))$$

which can more compactly be written as

$$\mathrm{Rev} \circ \mathrm{Neg} = \mathrm{Neg} \circ \mathrm{Rev}$$

With similar notation, we obtain

$$\mathrm{Neg} \circ \mathrm{Neg} = \mathrm{Rev} \circ \mathrm{Rev} = \mathrm{Id}$$

For combinations of Rev and Neg with other operators, we obtain the following table (figure 2):

/insert figure 2 here/

Let I be the equality relation (on members of U). We obtain the following set relations:

γ	Rev $\circ \gamma$	Neg $\circ \gamma$
Aa	Aa $\circ$ Rev	Ee $\circ$ Neg
Ae	Ea $\circ$ Rev	Ta $\circ$ Neg
At	Ta $\circ$ Rev	Ea $\circ$ Neg
Ea	Ae $\circ$ Rev	At $\circ$ Neg
Ee	Ee $\circ$ Rev	Aa $\circ$ Neg
Ta	At $\circ$ Rev	Ae $\circ$ Neg

F. ure 2.

Ee(I) = λ(b,c)(b and c overlap)

Aa(Neg(I)) = λ(b,c)(b and c are disjoint)

At(I) = λ(b,c)(b is a subset of c)

Ea(Neg(I)) = λ(b,c)(b is not a subset of c)

In particular, b[Ee(I)]b means "b has some member", and b[Aa(Neg(I))]b means "b is the empty set". The relation Ea(I) may also be useful; b[Ea(I)]b means "b is a set of exactly one member".

Consider now the set Γ of all set-relations that can be constructed from Δ using these operators. Clearly, every member of Γ can be written in exactly one way as

q(d(s(R)))

where R is a member of Δ, s is either Neg or Id, d is either Rev or Id, and q is one of the six operators Aa through Ia.

A data base where members of $2^U \times \Gamma \times 2^U$ are represented in a property structure can be characterized as a set-oriented property-structure representation for binary relations. We introduce the acronym SPB for this representation.

Let us finally comment on how to visualize an SPB property structure. It should obviously be thought about as a network, where each node represents a subset of U, and each arc aGb is a member of $2^U \times \Gamma \times 2^U$. If G = q(d(s(R))), it is useful to think about the arc as directed from a to b, and labeled with d(s(R)), whereas the q is manifested by two connection codes A, E, or T which indicate how the arc is connected to the nodes. A stands for "all members", E stands for "one and the same member", and T stands for "one variable member". For example,

$$b[Ae(Rev(Neg(R)))]c$$

would be visualized as in figure 3:

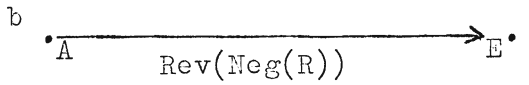

Figure 3

The same relation can be written as

$$c[Ea(Neg(R))]b$$

which is visualized as in figure 4:

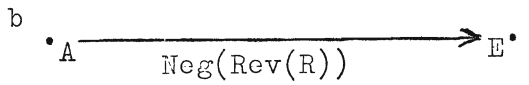

Figure 4

Thus the table for $Rev \circ \Upsilon$ above is thought about as saying "the connection codes do not move when we reverse the arrow".

The SPB property structure has the advantage of offering a simple and efficient way to store information about classes of objects in a data base. Other approaches to the same problem (such as present attempts to find good refutation procedures for higher-order logic) are certainly more general, but their maturation to practical use seems to lie quite some way in the future.

3. Ground level chaining rules

By a chaining rule, we still mean a rule of the type

$$uPv\ ,\quad vQw \vdash uRw$$

which is written compactly as

$$P \times Q \rightarrow R$$

If P, Q, and R are ground relations, then this is a ground level chaining rule. In this section, we shall study some equivalences which permit us to consider several, apparently dissimilar chaining rules as essentially the same.

We immediately notice

Proposition 3.1. If $P \times Q \rightarrow R$, then the following chaining rules also hold:

$$Rev(Q) \times Rev(P) \rightarrow Rev(R)$$

$$Neg(Rev(R)) \times P \rightarrow Neg(Rev(Q))$$

$$Q \times Neg(rev(R)) \rightarrow Neg(Rev(P))$$

Proof from the definitions of Rev and Neg.

Corollary 3.2. Iterated use of proposition 3.1 on a chaining rule $P \times Q \rightarrow R$ gives exactly six rules, viz. the four ones mentioned in proposition 3.1, plus

$$Rev(P) \times Neg(R) \rightarrow Neg(Q)$$

$$Neg(R) \times Rev(Q) \rightarrow Neg(P)$$

Proof by inspection of all possible cases.

Notice that if the Rev and Neg operators are ignored, the six forms of a chaining rule are exactly the six permutations of the three elements P, Q, and R. Let us now look for a pattern among the Rev and Neg prefixes.

The rule $P \times Q \rightarrow R$ is clearly equivalent to

$$(\forall u)(\forall v)(\forall w)\ \neg (uPv \wedge vQw \wedge wR'u)$$

where $R' = Neg(Rev(R))$. We can illustrate the latter statement as a "forbidden triangle" (figure 5):

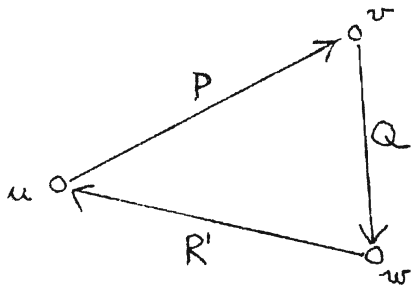

Figure 5.

Moreover, if we write this forbidden triangle as a triple

$$\langle P,Q,Neg(Rev(R))\rangle$$

then it is immediately obvious that the same triangle can be written in five other ways, e.g.

$$\langle Q,Neg(Rev(R)),P\rangle$$

or

$$\langle Rev(P),Neg(R),Rev(Q)\rangle$$

Moreover, since each triple $\langle P,Q,R'\rangle$ is equivalent to a chaining rule $P \times Q \rightarrow Neg(Rev(R'))$, these six ways of writing the forbidden triangle immediately give us the six forms of the chaining rule derived in corollary 3.2.

Since the forbidden triangle is clearly the basic thing, we shall change our habits with respect to the symbol R and talk about a forbidden triangle $\langle P,Q,R\rangle$, which is equivalent to the six chaining rules

$$P \times Q \rightarrow Neg(Rev(R))$$

$$Q \times R \rightarrow Neg(Rev(P))$$

$$R \times P \rightarrow Neg(Rev(Q))$$

$$Rev(R) \times Rev(Q) \rightarrow Neg(P)$$

$$Rev(Q) \times Rev(P) \rightarrow Neg(R)$$

$$Rev(P) \times Rev(R) \rightarrow Neg(Q)$$

Chaining rules are needed for the linear chaining procedure in section 1. However, it is more convenient to bypass the chaining rule

and to describe the procedure directly in terms of forbidden triangles:

Linear chaining procedure (new description): To determine whether uRw is stored (explicitly or implicitly) in a property structure σ, perform the following:

(a) See whether Rw (u). If so, exit with success.

(b) Otherwise, for all properties Pv that u has in σ, and for all Q such that $\langle \mathrm{Rev}(\mathrm{Neg}(R)), P, Q \rangle$ is a forbidden triangle, set up the sub-problem to prove that vQw is stored in σ. If this subproblem has not yet been attempted, attempt it by the linear chaining procedure. When no new sub-problems remain, exit with failure.

The idea in step (b) is to prove that if $u[\mathrm{Neg}(R)]w$, or equivalently, $w[\mathrm{Neg}(\mathrm{Rev}(R))]u$ were in the data base, then this would at least implicitly close a forbidden triangle there. This is indeed the case if we can prove that vQw is in the data base, which is true if $v[\mathrm{Neg}(Q)]w$ closes a forbidden triangle, etc. The process is illustrated in figure 6.

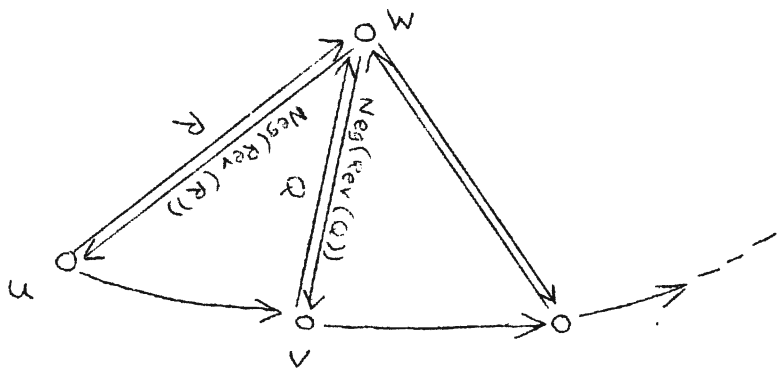

Figure 6.

The chaining procedure runs through a sequence of forbidden triangles with a common vertex w. Triangles are connected by an inversion relationship (Q to Neg(Rev(Q))) between adjacent sides of two successive triangles. Rays coming to the vertex w stand for successive questions (things to be proved); rays going out stand for things to be disproved. Arcs along the perimeter are in the data base. The procedure ends successfully when some ray coming into w is also in the data base.

Notice that w is not needed during chaining, but only in the termination criterion. Therefore, essentially the same chaining procedure can be used to answer questions of the type: "which w satisfies uRw?" (open questions, WH questions).

4. Set level chaining rules

Let $\langle P,Q,R\rangle$ be a forbidden ground level triangle. In this section, we shall derive forbidden triangles on set relations obtained from P, Q, and R.

We can immediately write down a few of them:

$\langle Aa(P),Ee(Q),At(R)\rangle$

$\langle Aa(P),Ee(Q),Ae(R)\rangle$

$\langle At(P),At(Q),Ae(R)\rangle$

$\langle At(P),Ae(Q),Ae(R)\rangle$

$\langle Ae(P),Ae(Q),Ae(R)\rangle$

The reader should verify that these are indeed forbidden. For example, if the first trinagle were not forbidden, we could have

$$b[Aa(P)]c \wedge c[Ee(Q)]d \wedge d[At(R)]b$$

i.e., all b are P'ing all c, some c is Q'ing some d, and for every d, there exists some b which it is R'ing. Consider one c1 in c and one d1 in d which are Q'ing together. d1 is R'ing some b1 in b. This b1, like all b, is P'ing with all c including c1. But this was a forbidden situation.

Now, if $\langle P,Q,R\rangle$ is a forbidden triangle, so is $\langle Q,R,P\rangle$, so we have five more set level forbidden triangles, e.g.

$\langle Aa(Q),Ee(R),At(P)\rangle$

In fact, there is no point in writing out all the set level forbidden triangles. Instead, we have the following <u>forbidden operator triangles</u>:

$\langle Aa,Ee,At\rangle$

$\langle Aa,Ee,Ae\rangle$

$\langle At,At,Ae\rangle$

$\langle At,Ae,Ae\rangle$

$\langle Ae,Ae,Ae\rangle$

plus the following rule:

/next page/

Rule of triangle composition: if $\langle P,Q,R\rangle$ is a forbidden ground level triangle, and $\langle A,B,C\rangle$ is a forbidden operator triangle, then $\langle A(P),B(Q),C(R)\rangle$ is a forbidden set level triangle.

We have now compacted our chaining rules quite a bit. Let us illustrate this with an example. Consider the ground level chaining rule

$$I \times R \rightarrow R$$

(where I is the equality relation, and R is any relation). This corresponds to the forbidden ground level triangle

$$\langle I,R,Neg(Rev(R))\rangle$$

Combining the six forms of this triangle with each of the five operator triangles, we obtain in principle 30 set level triangles, which means 180 set level chaining rules. Some of these are of course identical (basically because triangles like $\langle Ae,Ae,Ae\rangle$ do not change when they are rotated), and some are not very interesting, but we do obtain e.g.

from At,Aa,Ee :

1. $At(I) \times Aa(R) \rightarrow Neg(Rev(Ec(Neg(Rev(R))))) = Aa(R)$

   i.e. $\subset \times Aa(R) \rightarrow Aa(R)$

2. $Ee(I) \times At(R) \rightarrow Neg(Rev(Aa(Neg(Rev(R))))) = Ec(R)$

   where $Ec(I)$ is the "overlap" relation

3. $At(R) \times Aa(Neg(Rev(R))) \rightarrow Neg(Rev(Ee(I)))$

   which is the "disjoint" relation

from At,At,Ae :

4. $At(I) \times At(R) \rightarrow Neg(Rev(Ae(Neg(Rev(R))))) = At(R)$

   where $At(I)$ is the "subset" relation

5. Specializing (4) to the case R = I we obtain

   $\subset \times \subset \rightarrow \subset$

6. $At(R) \times Ae(Neg(Rev(R))) \rightarrow Neg(Rev(At(I)))$

   which is the "is not superset" relation

and quite a number of others.

If the five forbidden operator triangles are visualized with connection codes as at the end of section 2, we notice

(1) there is one "A" connection and one "T" or "E" connection at each corner of the triangle;

(2) there is at least one "E" connection in each triangle;

(3) every triangle that satisfies (1) and (2) is a forbidden one.

Observations such as these can be used in programming the linear chaining procedure for set level triangles. This will be the topic of next section.

5. The linear chaining procedure on set level

The purpose of this section is to specify in detail the algorithm for linear chaining with set level chaining rules.

We assume that expressions

$$b[q(d(s(R)))]c$$

(with q,d, and s as in section 2) are stored in a property structure so that b obtains the property

$$\langle q,d,s,r,c\rangle$$

and c obtains the property

$$\langle Rev(q),\ \neg d,\ s,\ r,\ b\rangle$$

where Rev(q) is defined to be the q' in the equality

$$Rev(q(P)) = q'(Rev(P))$$

This definition should be natural. Our notation is helpful: we have Rev(Ae) = Ea, etc. - In the above fivetuples, it is assumed that d and s are represented as boolean variables. Moreover, we assume q to be represented as a triplet $\langle q1,\ q2,\ q3\rangle$ of boolean variables according to the following conventions (figure 7):

q	q1	q2	q3
Aa	0	0	0
Ee	1	1	1
At	0	1	1
Ea	1	0	0
Ae	0	1	0
Ta	1	0	1

figure 7.

With these codes, we have

$$Rev(q1,q2,q3) = \langle q2,q1,q3\rangle$$

Moreover, if Neg(q) is defined similarly to Rev(q), we have

$$Neg(q1,q2,q3) = \langle \neg q1,\ \neg q2,\ \neg q3\rangle$$

If the components of a property are packed into one computer word, then complementation in given bit positions may be faster than

permutations of bits. We can define the Rev operation through complementation if Aa is given a double representations as (0,0,0) or (1,1,0), and Ee is given a double representation as (0,0,1) or (1,1,1). In this case,

$$Rev(q1,q2,q3) = \langle \neg q1, \neg q2, q3 \rangle$$

The "meaning" of the various bits in the representation of q, are:
q1: is there an E or T connection at this end of the arrow?
q2: is there an E or T connection at the other end of the arrow?
q3: if there is exactly one E or T: is this a T? (otherwise, conventions are arbitrary).
However, these interpretations only apply to the case of single representation of Aa and Ee.

Let us now introduce one more operation on the q, besides Rev and Neg. With given operators q and q', we shall write q + q' for that operator for which $\langle q, q', q+q' \rangle$ is a forbidden operator triangle. q+q' may have no, one, or two values, and is given in the following table (figure 8):

q = \ q' =	000 Aa	111 Ee	011 At	100 Ea	010 Ae	101 Ta
000 Aa		Ae,At		Ee		Ee
111 Ee	Ea,Ta		Aa		Aa	
011 At	Ee		Ae		Ae,At	
100 Ea		Aa		Ea,Ta		Ea,Ta
010 Ae	Ee		Ae,At		Ae,At	
101 Ta		Aa		Ea,Ta		Ea

Figure 8: Table for q + q'

As we see, we sometimes obtain both Ae and At, or both Ea and Ta

as a "sum". We shall make the + operation unambigous by defining q + q' as At viz. Ta in these cases. This convention will be explained below.

Blank squares in the table indicate that no sum exists. It is easily verified that with our encodement of the q (with the single representation for Aa and Ee), we obtain

$$\langle q1,q2,q3\rangle \;+\; \langle q1',q2',q3'\rangle \;=$$
$$\underline{\text{if}}\;\; q2 = q1' \;\; \underline{\text{then}} \;\; \underline{\text{undefined}}$$
$$\underline{\text{else}} \;\; \langle \neg q2', \neg q1, \neg (q1 \wedge q2' \;\vee\; q3 \wedge q3')\rangle$$

It follows that

$$\text{Neg}(\text{Rev}(\; \langle q1,q2,q3\rangle \;+\; \langle q1',q2',q3'\rangle \;)) \;=$$
$$\underline{\text{if}}\;\; q2 = q1' \;\; \underline{\text{then}} \;\; \underline{\text{undefined}}$$
$$\underline{\text{else}} \;\; \langle q1,\; q2',\; (q1 \wedge q2' \;\vee\; q3 \wedge q3')\rangle$$

We now have all the preliminaries for the chaining procedure. Given a question bGd, we attempt to disprove

$$d\,[\text{Neg}(\text{Rev}(G))]\,b$$

Let this relation be represented by assigning the tuple

$$\langle q1,q2,q3,d,s,R,b\rangle$$

to d, as usual. For each property

$$\langle q1',q2',q3',d',s',R',c\rangle$$

that b has in the data base, we shall clearly do the following:

Set level chaining procedure (details):

(1) Determine a suitable ground level forbidden triangle. This is essentially an "addition" of $\langle d,s,r\rangle$ and $\langle d',s',r'\rangle$. It will have to be handled by a programmed routine, or by table look-up.

Example: The ground level chaining rule

$$R \times I \rightarrow R$$

is handled by the function

$\langle d,s,r\rangle + \langle d',s',r'\rangle =$

if $r = r'$ and $d \neq d'$ and $s \neq s'$ then 0,1,I

elseif $r = I$ and $s = 1$ then $\langle \neg d', \neg s', r'\rangle$

elseif $r' = I$ and $s' = 1$ then $\langle \neg d, \neg s, r\rangle$

else undefined

We assume here that s = Neg is stored as 0, and s = Id is stored as 1. We assume here that s = Neg is stored as 0, and s = Id is stored as 1.

(2) If one or more ground level triangles exist, determine an operator triangle $\langle q1,q2,q3\rangle + \langle q1',q2',q3'\rangle$ that can be applied to it.

(3) Combine the third side of the ground level triangle and the operator triangle (this is essentially a concatenation of the bit sequences). We wish to prove that the resulting relation G' satisfies $c\,[G']\,d$. To prove this, attempt to disprove $d\,[Neg(Rev(G'))]\,c$. This is done by running through steps (1) - (3) again for each property that c has.

We can now see why + could be disambiguated in figure 8. The + operation is used to construct something that we are to prove, and it is always easier to prove a relation of the type At(P) than of the type Ae(P). Let us see what happens in the two cases! After neg-reversion, Ae has gone into At, and At into Ae. Thus our strategy tells us to disprove Ae(P) type relations, and to ignore At(P) types. This is all right, because if $\langle At,B,C\rangle$ is a forbidden triangle, so is $\langle Ae,B,C\rangle$. When we use the disambiguated table for +, we need only do one thing to account for the double sum that was removed: modify the termination criterion for the chaining procedure. It shall now go as follows:

Termination of set level chaining. The chaining procedure terminates successfully if either of the following things happen:

(a) we are told to prove cHd, and this relation is already stored in the data base (explicitly);

(b) we are told to prove $c\,[At(P)]\,d$, and $c\,[Ae(P)]\,d$ is already in the data base;

(c) we are told to prove $c[Ta(P)]d$, and $c[Ea(P)]d$ is already in the data base.

The procedure terminates with failure when there are no new sub-problems.

With the above chaining procedure, we will perform addition on the $\langle d,s,r \rangle$ and the q independently, concatenate the results, and then perform the Neg-Rev operations on the entire relation. Since the Neg and Rev operations go independently on q and s, viz. on q and d, we can perform the Neg-Rev operation immediately after addition in steps (1) and (2), and concatenate the results after Neg-Rev-ing. This is attractive because the operation Neg(Rev(q+q')) looks simpler than q+q' , when expressed in terms of bit operations. The same thing holds for the addition of $\langle d,s,r \rangle$; the example in step (1) above can be re-written as

$$\text{Neg}(\text{Rev}(\ \langle d,s,r \rangle \ + \ \langle d',s',r' \rangle\ )) \ =$$

$$\underline{\text{if}}\ \ r = r' \ \ \underline{\text{and}}\ \ d \neq d' \ \ \underline{\text{and}}\ \ s \neq s' \ \ \underline{\text{then}}\ \ \langle 0,0,I \rangle$$

$$\underline{\text{elseif}}\ \ r = I \ \ \underline{\text{and}}\ \ s = 1 \ \ \underline{\text{then}}\ \ \langle d,s,r \rangle$$

$$\underline{\text{elseif}}\ \ r' = I \ \ \underline{\text{and}}\ \ s' = 1 \ \ \underline{\text{then}}\ \ \langle d',s',r' \rangle$$

$$\underline{\text{else}}\ \ \underline{\text{undefined}}$$

(Notice that direction d is immaterial for equality). Unfortunately, it is difficult to manage completely without the "clean" (non-Neg-Rev-ed) sums, since they are needed in the termination criterion. Therefore, we either need to do a Neg-Reversion operation in each cycle of the chaining procedure, or to store both a "clean" and a "neg-reversed" copy of the relation in each property in the data base.

6. Representation of more complex logical relationships

Although the introduction of operators Aa, Ae, etc. increases the expressive power of property structure notation, it still does not approach the power of predicate calculus. In this section we shall say a few words about how more complex logical relationships can be represented.

Let us assume that the system which uses the SPB data base (usually, a question-answering system) has a fixed set Δ of ground level relations, and that only constants (for subsets of U) may be added to the system while it is running. This assumption is usually valid. It is then reasonable to distinguish between two types of logical rxpressions:

(a) lExpressions that do not contain any constant

Example 1. We may have ground relations P and Q for which uPv always implies uQv.

Example 2. On the set level, we have two rules that arise from the forbidden situation

$$b[Ee(R)]d \quad , \quad d[Aa(Neg(I))]d$$

where I is the equality relation and R is any ground level relation.

Example 3. Again on the set level, we have the general rule that $b[Ae(R)]d$ implies $b[At(R)]d$.

Expressions of this kind clearly should not have to be added to the system's data base at run time, since all such expressions can be analyzed at system design time, when the set of relations Δ is selected. For efficiency reasons, such deduction rules should be represented implicitly in the deduction procedure, e.g. as specialized sub-goal generators that cooperate with the chaining procedure. We have already carried this out for example 3. The other examples can be handled in a similar fashion. There is then no need to design a property-structure representation for such expressions.

(b) Expressions that do contain constants

Example: "If you drop an object, it will hit the ground". (With the

representation of natural-language information that we use in our project, and that was outlined in section 1, "drop", "object", "hit", and "the ground" will be taken as constants here).

It must certainly be possible to add statements such as this one to an SPB data base during a conversation. It is proposed that this is done in the following way:

1. Nodes in the SPB data base may be of two kinds: constants and variables. Constants behave as we are used to from previous sections in this paper; for variables there are some new conventions.

2. Variables have arcs coming and going, just like other nodes. However, we introduce one more connection code besides A, E, and T (cf. section 2). The new code is written D, which stands for Definition. Correspondingly, we introduce new operators Ad, Da, Dd, Dt, and Td.

3. The use of variables is outline by the following example: the rule

$$(\forall\alpha)\quad \alpha\left[At(P)\right]b \;\wedge\; \alpha\left[Aa(Q)\right]c \;\supset\; \alpha\left[Aa(R)\right]d$$

(where b, c, and d are constants) is represented in the SPB data base by a variable a, which is a node in the arcs (or "relations")

$$a\left[Dt(P)\right]b$$
$$a\left[Da(Q)\right]c$$
$$a\left[Aa(R)\right]d$$

In this case, the variable a can be interpreted as the lowest upper bound of all sets α which satisfy $\alpha\left[At(P)\right]b$ and $\alpha\left[Aa(Q)\right]c$. It follows that it is meaningful to consider the set of all the D-connected arcs that go to a variable node, but not a D-connected arc in isolation.

The idea of using variables in this way has one important advantage: it integrates well with the chaining procedure and other deduction methods in the SPB data base. Thus in a simple case, the variable a may be used as follows:

A. It is desired to prove that $e\left[Aa(R)\right]d$.

B. Through chaining, we establish the sub-goal to prove that e is

a subset of a.

C. The deduction procedure notices that a is a variable, and generates the two AND-connected subgoals: prove that e[Λa(Q)]c and e[Λt(P)]b. (These are the D-connected arcs in a, with an Λ inserted instead of the D).

A more systematic treatment of the use of variables will be given in later papers.

Acknowledgements

This research has been supported in part by the Swedish Natural Science Research Council (contract Dnr 2711-6) and by the (Swedish) Research Institute of National Defense (beställn. 719925).

References

Green 1968a C C Green, B Raphael
The use of theorem-proving techniques in question-answering systems
Paper at 1968 ACM Conference, Las Vegas

Levien 1965a R Levien, M E Maron
Relational Data File: A tool for mechanized inference execution and data retrieval
RM-4793-PR (RAND Corp., Santa Monica, Calif.)

Lindsay 1962a R K Lindsay
A program for parsing sentences and making inferences about kinship relations
Proc of Western Management Science Conference on Simulation (A Hoggatt, ed.)

Nilsson 1969a N J Nilsson
Predicate Calculus Theorem Proving
SRI, Menlo Park, Calif.

Raphael 1964a B Raphael
SIR - A computer program for semantic information retrieval
MIT Math dept., Ph D thesis, 1964

Robinson 1965a J A Robinson
A machine oriented logic based on the resolution principle
Journal of ACM, January, 1965

Slagle 1965b J R Slagle
A proposed preference strategy using sufficiency-resolution for answering questions
UCRL-14361 (Lawrence Radiation Labs, Calif.)