DRAFT: January 30, 1996

CS-95-24

XI: A Knowledge Representation Language Based On Cross-Classification and Inheritance

R. Gaizauskas

XI: A Knowledge Representation Language Based On Cross-Classification and Inheritance

R. Gaizauskas
Department of Computer Science
University of Sheffield
robertg@dcs.shef.ac.uk

1 Introduction

XI is a language for representing knowledge about individuals, about classes of individuals, and about inclusion relations between classes of individuals. It allows for straightforward definition of cross-classification hierarchies and for the association of arbitrary attribute-value information with classes or with individuals. This information may be in the form of simple, atom-valued attribute-value associations or in the form of Horn clause rules which specify how the value of an attribute is to be derived from other information held in the hierarchy. XI provides a simple inheritance mechanism which allows attribute values to be inherited by classes or individuals lower in the hierarchy. The name 'XI' is meant to suggest cross-classfication – X - and inheritance – I – the two primary features of the language.

The motivation for XI came from work in natural language understanding (see below) where it was discovered that traditional *property-based* logic languages such as Prolog were inadequate. In such a language it is relatively easy to discover what things have a given property (e.g. to find all green things issue the query `?- green(X).`) but very difficult, without changing representations, to discover all the properties which hold of an individual. To do this an *object-based* logic is required and it is to this end that XI has been created. Of course another option is to adopt a second, or higher order, logic. But computational problems with these languages (e.g. second-order unification is undecidable) together with their excessive formal power suggest, at least initially, the exploration of simpler approaches.

At one level XI may be viewed simply as a declarative formalism with its own syntax and a semantics based on FOPC. But an interpreter and compiler for XI exists, together with ancillary commands that allow XI models to be incorporated into Prolog programs. In this way XI may be viewed as something akin to Prolog – a language which runs but which has some claim to a declarative semantics.

Throughout, the design philosophy has been to 'keep it simple'. Classes are simply unary predicates and bare individuals are numbered constants (`e1,e2,...`). Attributes are binary predicates, the first argument identifying the class or individual of which the attribute holds and the second being the value (which may be a variable to be instantiated by the execution of a rule associated with the attribute). The language has two components, a *definitional* component and a *derivational* component. The definitional component allows a hierarchy to be defined and attributes to be associated with nodes in the hierarchy. The derivational component allows one to determine just two sorts of things: whether one node dominates another in the hierarchy and what value an attribute has at a given node. Attribute values are determined first at the given node and then are inherited by working depth-first up the hierarchy from left to right. Multiple values may be obtained by backtracking and nothing is done to prohibit these values from being contradictory – this is left to the application. The application programmer is free to implement a default inheritance scheme on top of XI.

The definition of a cross-classification hierarchy we refer to as an *ontology*. An ontology together with a mapping of nodes in the ontology onto sets of attribute-values forms a *world model*. Here is an example of the XI expressions that define a fragment of a crude world model to do with vehicles. First, the ontology (Figure 1 illustrates these hierarchical relationships graphically):

```
top(X) ==> object(X) v event(X) v property(X).

object(X) ==> vehicle(X) v country(X).
vehicle(X) ==> (car(X) v lorry(X) v motorcycle(X)) &
               (commercial(X) v private(X)).
car(X) ==> (rover(X) v toyota(X) v renault(X)) &
           (twodoor(X) v fourdoor(X)).
e1 <-- private(X) & toyota(X) & fourdoor(X).
e2 <-- country(X).
e3 <-- country(X).

property(X) ==> functional_prop(X) v relational_prop(X).
made_in <-- functional_prop(X).
colour_of <-- functional_prop(X).
```

Terms on the left of the `==>` arrow are superordinate classes and terms on the right are subclasses. Each conjunct on the right is a dimension of classification and the disjuncts within each conjunct represent mutually exclusive alternative classifications within the given dimension So, for example, a vehicle will be at most one of a car, a lorry, or a motorcycle and for whichever of these is chosen it may also be classified as either commercial or private. Terms on the left of the `<--` arrow are instances, denoted with constant terms of the form `e1, e2, ...`, and terms on the right of this arrow are the classes of which the terms on the left are instances. So, `e1` is an instance of something which is private, a Toyota, and four-door.

Here are some XI expressions associating attributes with nodes in the ontology:

```
props(vehicle(X), [presupposition(X, driver(Y,X))]).
props(rover(X), [made_in(X,e2)]).
props(toyota(X), [(made_in(X,e2) :- X <- twodoor(_)),
                  (made_in(X,e3) :- X <- fourdoor(_))]).

props(e1,[colour_of(e1,blue)]).
props(e2,[name(e2,uk)]).
props(e3,[name(e3,japan)]).

props(made_in,[instance_type(object(_)),value_type(country(_))]).
props(colour_of,[instance_type(object(_)),value]).
```

Note that an attribute value may either be directly associated with a class or an instance (as in `props(rover(X),[madein(X,uk)])` or `props(e1,[colour(e1,blue)])` which tell us, respectively, that Rovers are made in the UK and that e1 is blue) or indirectly via a rule which must be evaluated for the value to be determined (as in the case of the Toyotas which if twodoor are made in the UK and if fourdoor are made in Japan). Note also that by treating attributes themselves as individuals we can store information about attributes, such as the types of their arguments, within a XI model.

Given this world model we may now ask questions such as:

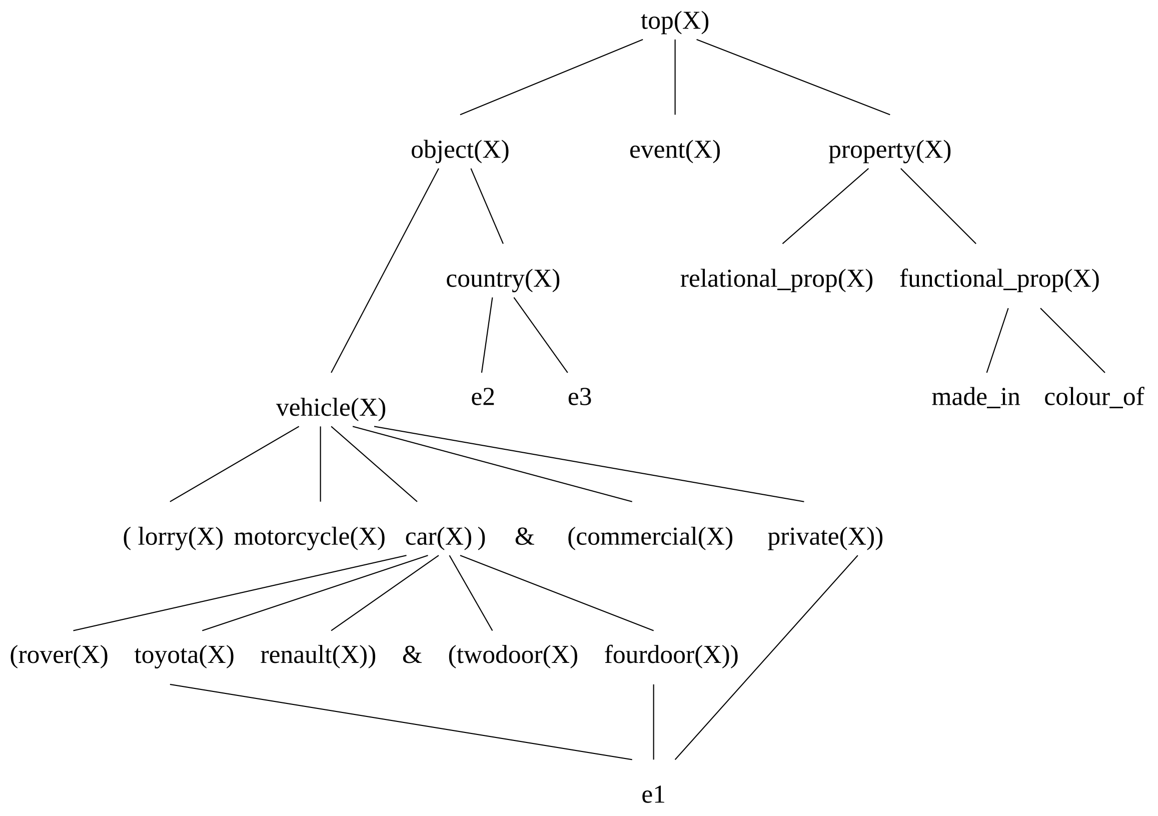
top(X)
object(X)
event(X)
property(X)
country(X)
relational_prop(X)
functional_prop(X)
vehicle(X)
e2
e3
made_in
colour_of
(lorry(X) motorcycle(X) car(X)) & (commercial(X) private(X))
(rover(X) toyota(X) renault(X)) & (twodoor(X) fourdoor(X))
e1

Figure 1: A Sample XI Ontology

```
?- e1 <- vehicle(X).                    /* Is e1 a vehicle ? */
yes

?- hasprop(e1,colour_of(e1,X)).         /* What colour is e1 ? */
X = blue.

?- hasprop(e1,made_in(e1,C)).           /* Where was e1 made ? */
C = e3.

?- hasprop(e3,name(e3,N)).
N = japan.

?- hasprop(e1,presupposition(e1,X)).    /* Are there presuppositions for e1 ? */
X = driver(Y,e1).

?- hasprop(e1,P).                       /* What attributes hold of e1 ? */
P = colour_of(e1,blue) ;
P = made_in(e1,e3) ;
P = presupposition(e1,driver(Y,e1)) ;
no
```

XI was motivated by work in natural language understanding (Gaizauskas et al. (1995), Gaizauskas et al. (1993), Cahill et al. (1992)). In particular it was designed to record the sort of world knowledge that is needed in discourse interpretation. One example of this is coreference resolution. Suppose you are trying to understand road traffic incident reports and come across the sentence "Blue Ford overturned in inner lane" and later "Vehicle now removed". To resolve "Blue Ford" and "Vehicle" requires the knowledge that Fords are vehicles. Natural language understanding requires large amounts of this kind of taxonomic knowledge. Notice that this knowledge is not present in the text but rather must be brought to bear by the language understander.

Natural langugage understanding programs can also make wide use of property inheritance. For example, you might want to record the fact that a vehicle presupposes a driver – this will help in understanding further messages in our example such as "Driver uninjured". As shown, XI allows you to associate with the vehicle node in the hierarchy a property such as `presupposition(driver(Y,X))`. An application might, when adding an instance of a vehicle to its model of the world, choose to add instances of all presupposed objects at the same time. Since we have stored the presupposition about drivers at the vehicle node it will be inherited by lower nodes. Thus, if a Toyota is added to the model, then as an instance of a subclass of vehicle it will inherit the presupposition about a driver and an instance of a driver for the Toyota may be created. Later if a new class of vehicles, say Fords, is created then the fact that they presuppose drivers need not be added again, as it will be inherited from the vehicle node. This simply illustrates the well known advantages of inheritance schemes for reducing redundancy in the knowledge base.

Finally, since attributes may themselves be treated as individuals they too may be placed in classes and have attributes attached to them, and this 'higher order' aspect of XI can be useful in NL applications. If in our traffic example we were to get another message asserting 'Red Toyota also involved' we would want to be sure that the red and blue Toyotas did not get coresolved. One way to prevent this is to create a class of attributes which are 'functional' in the sense that any such attribute cannot have multiple values (at least simultaneously) for any object of which it holds. So, a vehicle cannot be more than one `colour` simultaneously, but it can , say, be `beside` several different things at the same time. If classes of 'functional' and 'relational' attributes are distinguished, a coreference algorithm can utilise this distinction in coming to a decision as to whether or not two objects are identical. Note that in deciding that two entities are or are not the same we may need to compare everything that is known about them. This illustrates the requirement for a KR language in which it is straightforward to discover all of the attributes that hold of an individual.

The remainder of this paper is organised as follows. In section 2 the syntax of XI is formally defined. Section 3 defines *derivability* in XI and section 4 discusses the semantics of XI. Section 5 discusses various nonlogical features of XI that facilitate its incorporation into Prolog applications. Section 6 provides a few details about the compiler. Section 7 supplies concluding remarks. Source code may be found in the appendix.

2 The Definition of Pure XI

Since XI is an extension to Prolog much of its syntax bears a close ressemblance to that of Prolog.

The *alphabet of XI*, $\mathcal{A}_{XI}$ consists of seven classes of symbols:

1. functor symbols: lower case, alpha-initial alphanumerics
2. reserved functor symbols: `props`, `hasprop`
3. variables: uppercase-alpha-initial alphanumerics

4. connectives: $:-$, , ,; , $\Longrightarrow, \Rightarrow, \leftarrow, \longleftarrow, \vee, \&$

5. punctuation: (,)

The further constraint is adopted that these classes be disjoint.

A *term* is a variable, a 0-ary functor symbol (also called an *instance symbol*), or an expression of the form $f(t_1, \ldots, t_n)$ where f is a functor symbol and $t_1, \ldots, t_n$ are terms. Terms of this last sort are called *complex*. In XI classes are denoted by complex terms with a class functor and exactly one variable – we call these *class terms*. Attributes are asserted with *attribute terms* – complex terms with attribute functors and exactly two variables. If t is a term we let $\mathrm{var}(t)$ denote the set of variables occurring in t.

We now define three further classes of expressions: O-clauses (for defining an ontology), G-clauses (for defining goals or queries), and P-clauses (for associating properties with classes).

O-clauses have one of two forms:

1. $c \Longrightarrow D_1 \& \cdots \& D_n$, where c is a class term and each D_j is a disjunction of class terms of the form $d_{j,1} \vee \cdots \vee d_{j,p_j}$ and each class term $d_{j,k}$ on the right of the O-clause contains the variable occurring in the class term on the left of the O-clause – i.e. $\mathrm{var}(d_{j,k}) = \mathrm{var}(c)$;

2. $e \longleftarrow c_1 \& \cdots \& c_n$ where each c_i is a class term containing the same variable, i.e. $\mathrm{var}(c_1) = \cdots = \mathrm{var}(c_m)$ and e is an instance symbol.

$x \Longrightarrow y$ may be read 'y is an immediate subclass of x' and $x \longleftarrow y$ may be read 'x is an immediate instance of y'. Each conjunct (D_i) on the right hand side of an O-clause of type 1 may be thought of as a *classificatory dimension* and the disjoined terms within in it are intended to be mutually exclusive classes (this exclusiveness in enforced below through constraints on the sets of O-clauses that may form an ontology). The restrictions on the co-ocurrence of variables are intended to ensure that if the class terms are interpreted as predicates then when the variables in a subclass get instantiated the corresponding variables in the superclass are also instantiated in such as fashion as to guarantee that the subclass predication implies the superclass predication. E.g. suppose $financial\_event(X) \Longrightarrow buy(X) \vee sell(X)$. If $e1 \longleftarrow buy(X)$ then by unifying $e1$ with X we can determine that $e1$ is a $financial\_event$.

If $\mathbf{O}$ is a set of O-clauses we define the transitive, reflexive relation $\leq_{\mathbf{O}}$ to hold between two terms t_1 and t_2 which occur in clauses in $\mathbf{O}$ as follows. $t_1 \leq_{\mathbf{O}} t_2$ if:

1. $t_1 = t_2$; or

2. t_2 occurs on the left hand side of an O-clause of type-1 in $\mathbf{O}$ and t_1 occurs on the right hand side of the same O-clause; or

3. t_2 occurs on the right hand side of an O-clause of type-2 in $\mathbf{O}$ and t_1 occurs on the left hand side of the same O-clause; or

4. there exists a term t_3 occurring on the right hand side of an O-clause of type-1 in $\mathbf{O}$ in which t_2 occurs on the left and $t_1 \leq_{\mathbf{O}} t_3$.

G-clauses have one of the forms:

1. $c_1 \Rightarrow c_2$ where $c1$ and c_2 are class terms or variables;

2. $e \leftarrow c$ where e is an instance symbol or a variable and c is a class term or a variable;

3. $\texttt{hasprop}(e,p)$ where e is an instance symbol or a variable and p is an attribute term or a variable;

4. G_1, G_2 where G_1 and G_2 are G-clauses.

5. $G_1; G_2$ where G_1 and G_2 are G-clauses.

$x \Rightarrow y$ may be read 'y is an subclass of x' and $x \leftarrow y$ may be read 'x is an instance of y'.

P-clauses have the form $\texttt{props}(c, \mathbf{V})$ where c is either an instance symbol or a class term and $\mathbf{V}$ is a set whose members are of the form

1. $p(t_1, t_2)$ where p is a functor and t_1 is c if c is an instance term and $var(c)$ otherwise; or

2. $p(t_1, t_2) :- G$ where p is attribute term as in 1) and G is a G-clause.

A *world model* W is a pair $(\mathbf{O}, \mathbf{P})$ where

1. $\mathbf{O}$ is a set of O-clauses such that:

   (a) there exists a unique term r such that for all terms t occurring in $\mathbf{O}$ $t \leq_{\mathbf{O}} r$; and

   (b) for any O-clause $c \Longrightarrow D_1 \& \cdots \& D_n$ if t_1 and t_2 both occur in some D_i, i.e. if D_i has the form $t_1 \vee t_2 \vee \cdots \vee tn$ then there is no term t in any O-clause in $\mathbf{O}$ such that $t \leq_{\mathbf{O}} t_1$ and $t \leq_{\mathbf{O}} t_2$.

2. $\mathbf{P}$ is a set of P-clauses such that for each P-clause $\texttt{props}(c, \{V_1, \ldots, V_n\})$ in $\mathbf{P}$ c occurs in some O-clause in $\mathbf{O}$.

Some remarks about this definition of world model are in order. The conditions on the O-clauses in $\mathbf{O}$ ensure that

1. the ontology has a greatest element (the universal class);

2. no class can inherit from two classes within the same classificatory dimension; i.e. since the classes within a classificatory dimension (one of the conjuncts D_i on the right of an O-clause of the form $c \Longrightarrow D_1 \& \cdots \& D_n$) are meant to be mutually exclusive then clearly no subclass can be a subclass of two or more of them.

The O-clauses in a world model may be seen as an extensional definition of a partial ordering relation on a subset of the class terms of $\mathcal{L}_{XI}$. The conditions on $\mathbf{P}$ ensure that all the classes which have properties associated with them in P-clauses are defined somewhere in the ontology.

The *language of XI*, $\mathcal{L}_{XI}$ is the set of strings in $\mathcal{A}^*_{XI}$ which are O-, G-, or P-clauses.

3 Derivability in Pure XI

Only G-clauses may be derived in XI. There are five types of G-clauses: those asserting dominance relations between classes, those asserting dominance relations between classes and instances, those asserting that an attribute holds of an instance (possibly by inheritance), and conjunctions and disjunctions of these three basic types.

To establish that one class is a subclass of another or that an instance is an instance of some class requires recursively exploring the set of O-clauses in the world model to see if the terms in query are appropriately related; i.e. it involves seeing whether for a given set of O-clauses $\mathbf{O}$ and terms t_1 and t_2 the ordering relation $\leq_{\mathbf{O}}$ holds between them. To establish that a attribute holds of a given instance requires checking to see if the attribute is recorded in the set of attributes associated with the instance and this may in turn require determining that other G-clauses hold, if the attribute is conditional. If the attribute does not hold 'locally' then the ontology is recursively explored upwards to see if the attribute is associated with any dominating node. Note that the attribute we are seeking to establish may be a partially instantiated term. Thus, seeing that the attribute is recorded at a class or instance node requires seeing if the goal term may be unified with (the head of) any term stored in the set of attributes at the class or instance node. Conjoined goals are established by establishing each conjunct. Disjoined goals are established by establishing either disjunct.

As with Prolog, XI may be extended to permit the 'pure' underlying logic (definite clause logic in the case of Prolog) to include negated goals. As in Prolog, these are established in XI by failing to establish the corresponding unnegated goal ('negation as failure'). Thus there are no inference rules explicitly for negation and we first define a notion of derivation for pure, negation-free XI, and later extend this notion to include negated goals.

The inference rules in XI are the following:

1. $c \Longrightarrow D_{1,1} \& \cdots \& D_{1,n_1}$ (a type-1 O-clause)
   $c_1 \Longrightarrow D_{2,1} \& \cdots \& D_{2,n_2}$ (c_1 any class term in some $D_{1,j}$)
   $\vdots$
   $c_{k-1} \Longrightarrow D_{k,1} \& \cdots \& D_{k,n_k}$ (c_{k-1} any class term in some $D_{k-1,j}$)
   ―――――――――――
   $c \Rightarrow d$ (d any class term in some $D_{k,j}$)

2. $c \Rightarrow d$
   $e \longleftarrow d_1 \& \cdots \& d_m$ (a type-2 O-clause with $d = d_i$ for some $1 \leq i \leq m$)
   ―――――――――――
   $e \leftarrow c$

3. (a) $\texttt{props}(e, [\ldots, p(e,t) \ldots])$ (e an instance, p an attribute term, t any term)
   ―――――――――――
   $\texttt{hasprop}(e, p(e,t))$

   (b) $\texttt{props}(c(X), [\ldots, p(X,t), \ldots])$ ($c(X)$ a class term, p an attribute term, t any term)
   $e \leftarrow c(X)$ (e an instance term)
   ―――――――――――
   $\texttt{hasprop}(e, p(e,t))$

4. (a) $\texttt{props}(e, [\ldots, p(e,t) \text{:-}\ G, \ldots])$ (e an instance term,p an attribute term t any term, G a G-clause)
   $G\theta$ (θ a substitution for variables in G)
   ―――――――――――
   $\texttt{hasprop}(e, p(e,t)\theta)$

   (b) $\texttt{props}(c(X), [\ldots, p(X,t) \text{:-}\ G, \ldots])$ ($c(X)$ a class term, p an attribute term t any term, G a G-clause)
   $G\theta$ (θ a substitution for variables in G)
   $e \leftarrow c$ (e an instance term)
   ―――――――――――
   $\texttt{hasprop}(e, p(e,t)\theta)$

5. $$\frac{\begin{array}{l}G_1\\G_2\end{array}}{G_1, G_2} \qquad (G_1 \text{ and } G_2 \text{ G-clauses})$$

6. $$\frac{G_1}{G_1; G_2} \qquad (G_1 \text{ and } G_2 \text{ G-clauses})$$

7. $$\frac{G_2}{G_1; G_2} \qquad (G_1 \text{ and } G_2 \text{ G-clauses})$$

A *XI-derivation* of a G-clause G from a world model $\mathcal{W} = (\mathbf{O}, \mathbf{P})$ is a finite sequence of O-, P-, and G-clauses such that G is the last clause in the sequence and every other clause is either an O-clause or a P-clause in $\mathcal{W}$ or is a G-clause that follows from one or more clauses preceding it in the sequence according to one of the XI inference rules. The length of the derivation is the number of clauses in the sequence. If such a derivation exists we say G is *XI-derivable from* $\mathcal{W}$, written $\mathcal{W} \vdash_{XI} G$.

4 Semantics of Pure XI

Rather than specify a formal semantics for XI 'from the ground up' we will show how a XI world model can be translated via a translation function $\mathcal{T}$ into a first order theory which may then be given the standard first order semantics. We then show that for a given world model $\mathcal{W}$ and a goal G, if $\mathcal{W} \vdash_{XI} G$ then $\mathcal{T}(\mathcal{W}) \vdash_{FOPC} \mathcal{T}(G)$. This establishes the soundness of derivations in XI with respect to first order logic and gives an indirect semantics to XI world models. We do not appeal to any particular first order proof theory as the first order derivations we suppose are trivial: any natural deduction system such as (Smullyan, 1968) suffices.

Define a translation function $\mathcal{T}$ from O-,G-,P-clauses to sets of definite clause formulas as follows. In the first order theory we assume $:-$ is the material implication connective.

1. If C is an O-clause of the form $c \Longrightarrow D_1 \& \cdots \& D_n$, where c is a class term and each D_j is a finite disjunction of class terms of the form $d_{j,1} \vee \cdots \vee d_{j,p_j}$, then $\mathcal{T}(C)$ is the set of formulas containing
   (a) $c{:-}\ d_{j,k}$ for each (j,k), $1 \le j \le n$, $1 \le k \le p_j$ and
   (b) $\neg d_{j,1} \wedge \ldots \wedge \neg d_{j,k-1} \wedge \neg d_{j,k+1} \wedge \ldots \wedge \neg d_{j,p_j} {:-}\ d_{j,k}$ for each (j,k), $1 \le j \le n$, $1 \le k \le p_j$
2. If C is an O-clause of the form $e \longleftarrow c_1 \& \cdots \& c_n$ where each c_i is a class term and e is an instance symbol, then $\mathcal{T}(C)$ is the set of formulas $c_i\{X/e\}$ where X is the free variable in c_i, for each i, $1 \le i \le n$.
3. If C is a G-clause of the form $c_1 \Rightarrow c_2$ where $c1$ and c_2 are class terms then $\mathcal{T}(C)$ is the set of formulas $\{c_1{:-}\ c_2\}$;
4. If C is a G-clause of the form $e \leftarrow c$ where e is an instance symbol and c is a class term with free variable X then $\mathcal{T}(C)$ is the set of formulas $\{c\{X/e\}\}$;
5. If C is a G-clause of the form $\texttt{hasprop}(e, p)$ where e is an instance symbol and p is an attribute term then $\mathcal{T}(C) = \{p\}$;

6. If C is a G-clause of the form G_1, G_2 where G_1 and G_2 are G-clauses then if $\mathcal{T}(G_1) = \{G_1'\}$ and $\mathcal{T}(G_2) = \{G_2'\}$ then $\mathcal{T}(G) = \{G_2', G_2'\}$

7. If C is a G-clause of the form $G_1; G_2$ where G_1 and G_2 are G-clauses then if $\mathcal{T}(G_1) = \{G_1'\}$ and $\mathcal{T}(G_2) = \{G_2'\}$ then $\mathcal{T}(G) = \{G_2'; G_2'\}$

8. If C is a P-clause of the form $\texttt{props}(c, \{V_1, \ldots, V_n\})$ where c is a term and V_i has either the form p where p is an attribute term or p:– G where p is attribute term and G is a G-clause then $\mathcal{T}(C)$ contains exactly one formula for each V_i such that

   (a) If V_i has the form p and c is an instance symbol then the formula p is in $\mathcal{T}(C)$;

   (b) If V_i has the form p:– G and c is an instance symbol then the formula p:– G' is in $\mathcal{T}(C)$ where G' is the formula in $\mathcal{T}(G)$;

   (c) If V_i has the form p and c is a class term then the formula p:– c is in $\mathcal{T}(C)$;

   (d) If V_i has the form p:– G and c is a class term then the formula p:– c, G' is in $\mathcal{T}(C)$ where G' is the formula in $\mathcal{T}(G)$;

Suppose $\mathcal{W} = (\mathbf{O}, \mathbf{P})$ is a XI world model. Then define $\mathcal{T}(\mathcal{W})$ to be

$$\bigcup_{O \in \mathbf{O}} \mathcal{T}(O) \cup \bigcup_{P \in \mathbf{P}} \mathcal{T}(P).$$

Proposition 4.1 *Let $\mathcal{W} = (\mathbf{O}, \mathbf{P})$ be a world model and G be a G-clause. If $\mathcal{W} \vdash_{XI} G$ then $\mathcal{T}(\mathcal{W}) \vdash_{FOPC} \mathcal{T}(G)$.*

Proof We show for each of the seven inference rules in XI with premises S and conclusion G that a FOPC-derivation may be constructed of $\mathcal{T}(G)$ from $\mathcal{T}(S)$. It follows that for any XI-derivation from $\mathcal{W}$ a corresponding FOPC-derivation may be constructed from $\mathcal{T}(\mathcal{W})$.

1. G has the form $c \Rightarrow d$ and S consists of a sequence of type 1 O-clauses from $\mathbf{O}$ the first of which has c on the left and the last of which contains d somewhere on the right and each intermediate O-clause of which contains a class term on the left which occurs somewhere on the right in the preceding O-clause in the sequence. Consider the sequence $c, c_1, \ldots c_k, d$ of class terms leading from c to d. By the definition of the translation function $\mathcal{T}$ ensures that the clauses c:– c_1, c_1:– $c_2, \ldots c_{k-1}$:– c_k, c_k:– d will be in $\mathcal{T}(S)$ and these supply a FOPC-derivation of c:– d directly.

2. G has the form $e \leftarrow c$ and S consists of a G-clause $c \Rightarrow d$ followed by a single type 2 O-clause from $\mathbf{O}$ in which e occurs on the left and in which one class term on the right is d – the class term on the right of the preceding G-clause. The translation function $\mathcal{T}$ ensures that the clause c:– d is in $\mathcal{T}(S)$. In addition the clause $d\{X/e\}$ will be in $\mathcal{T}(S)$, where X is the variable in d. From these clauses it is clear there is a FOPC-derivation of $c\{X/e\}$, i.e. of $\mathcal{T}(G)$.

3. (a) G has the form $\texttt{hasprop}(e, p)$ where e is an instance symbol and p is an attribute term. S includes the P-clause $\texttt{props}(e, P)$, where the attribute term $p \in P$. $\mathcal{T}(S)$ includes p and since $\mathcal{T}(G) = p$ clearly there is a FOPC-derivation of $\mathcal{T}(G)$ from $\mathcal{T}(S)$.

   (b) G has the form $\texttt{hasprop}(e, p)$ where e is an instance symbol and p is an attribute term. S includes the P-clause $\texttt{props}(c(X), P)$, where the attribute term $p(X, t) \in P$, and the G-clause $e \leftarrow c(X)$. Thus, $\mathcal{T}(S)$ includes $p(X, t)$:– $c(X)$ and $c\{X/e\}$. From $\mathcal{T}(S)$ there is clearly a FOPC-derivation of $p(e, t) = \mathcal{T}(G)$.

4. (a) G has the form $\texttt{hasprop}(e, p(e,t)\theta)$ where e is an instance term, p is an attribute functor, t is an instance term or a variable, and θ is a substitution for $\text{var}(t)$. S includes the P-clause $\texttt{props}(e, P)$, where P is such that $p(e,t)\text{:-}\ H \in P$, and the G-clause $H\theta$. $\mathcal{T}(p(e,t)\text{:-}\ G) = p(e,t)\text{:-}\ H'$ where $H' = \mathcal{T}(H)$; further $\mathcal{T}(H\theta) = H'\theta$ (the order of application of a substitution to a G-clause relative to the application of the translation function $\mathcal{T}$ is irrelevant, keeping in mind that we mean the substitution to be applied to each member of the *set* of expressions resulting from the translation). Thus there is a FOPC-derivation of $p(e,t)\theta$, i.e. of $\mathcal{T}(G)$, from $\mathcal{T}(S)$.

   (b) G has the form $\texttt{hasprop}(e, p(e,t)\theta)$ where e is an instance term, p is an attribute functor, t is an instance term or a variable, and θ is a substitution for $\text{var}(t)$. S includes the P-clause $\texttt{props}(c(X), P)$, where P is such that $p(X,t)\text{:-}\ H \in P$, the G-clause $H\theta$, and the G-clause $e \leftarrow c(X)$. Thus, $\mathcal{T}(S)$ includes $p(X,t)\text{:-}\ c(X), H'$ where $H' = \mathcal{T}(H)$. It also includes $H'\theta$ and $c\{X/e\}$. Thus from $\mathcal{T}(S)$ there is a FOPC-derivation of $p(e,t)\theta = \mathcal{T}(G)$.

5. G has the form G_1, G_2 where G_1 and G_2 are G-clauses. S includes G_1 and G_2 and so $\mathcal{T}(S)$ contains $\mathcal{T}(G_1)$ and $\mathcal{T}(G_2)$. From them a FOPC-derivation, by conjunction introduction, of $\mathcal{T}(G_1), \mathcal{T}(G_2)$, i.e., of $\mathcal{T}(G_1, G_2)$ is immediately constructible.

6. G has the form $G_1; G_2$ where G_1 and G_2 are G-clauses and S contains G_1. So, $\mathcal{T}(S) = \mathcal{T}(G_1)$ from which $\mathcal{T}(G_1); \mathcal{T}(G_2)$, i.e. $\mathcal{T}(G_1; G_2)$, is FOPC-derivable by disjunction introduction.

7. G has the form $G_1; G_2$ where G_1 and G_2 are G-clauses and S contains G_2. So, $\mathcal{T}(S) = \mathcal{T}(G_2)$ from which $\mathcal{T}(G_1); \mathcal{T}(G_2)$, i.e. $\mathcal{T}(G_1; G_2)$, is FOPC-derivable by disjunction introduction.

∎

5 Impure XI

There are four sorts of extension to XI, as it is implemented as a KR language, which make it much more powerful but which affect the simple semantics given above. These extensions parallel similar features in Prolog. The first sort includes just the metalogical `not` operator; the second includes mechanisms for limiting search – the `xi_cut` operator which is analogous to the cut operator in Prolog and the `nodeprop` operator which limits property inheritance to the current node; the third includes mechanisms for altering the hierarchy by adding or removing instances, classes or properties and for saving or restoring 'contexts' of instances; the fourth includes mechanisms for input and output of world models.

5.1 Negation

As has already been indicated, impure XI permits a form of negation as failure. This done by introducing the meta-logical `not` operator which can be applied to G-clauses only (but hence may occur in the G-clause component of attribute clauses within P-clauses). The semantics of `not` are as in Prolog: `not` G follows from a world model $\mathcal{W}$ iff there is no derivation of G from $\mathcal{W}$.

We extend the notion of XI-derivation to that of XINF-derivation as follows. A *XINF-derivation* of a G-clause G from a world model $\mathcal{W} = (\mathbf{O}, \mathbf{P})$ is a finite sequence of O-, P-, and G-clauses such that G is the last clause in the sequence and every other clause is either an O-clause or a P-clause in $\mathcal{W}$, or is a G-clause that follows from one or more clauses preceding it in the sequence according to one of the XI inference rules, or is a G-clause of the form `not` H such that no XINF-derivation of H from $\mathcal{W}$ exists. In this case we write $\mathcal{W} \vdash_{XINF} G$.

5.2 Controlling Inheritance

When a `hasprop` G-clause is executed the ontology is searched back to the root node along every possible path until either some occurrence of the sought after attribute is found or it is determined that there is no such occurrence. Of course for complex hierarchies this can involve a large amount of computation. It is desirable, therefore, to give the application programmer some facility for reducing this search, even at the cost of failing to discover bonafide XI implications.

XI provides two such mechanisms. One is the predicate $\texttt{nodeprop}(Node, Prop)$ which returns the value of an attribute *Prop* at the given *Node* only – i.e. no inheritance is performed. If, in a given application, it is known that certain attributes will be found only at instance nodes, then expensive inheritance need not take place. This mechanism can also be used by applications to cache attributes at an instance node. That is, inheritance may be done once (or as necessary, if the world model is dynamic), the inherited attributes added to the local node, and some flag attribute set at the local node to indicate that inheritance has taken place. Subsequent access to the node can check the inheritance flag and not perform inheritance unless the flag has become unset (perhaps as the result of modifications higher up in the hierarchy).

The second mechanism XI provides for restricting search is the `xi_cut` operator, an analog of the Prolog cut operator, `!`. Like Prolog cut, `xi_cut` always succeeds, but after succeeding it prevents further search in the hierarchy for other values for the attribute in whose definition is it included. Suppose $p(X, Y){:-}\ G$ is an attribute clause occurring within $\mathbf{V}$ in a P-clause $\texttt{props}(t, \mathbf{V})$. By altering the attribute clause to

$$p(X, Y){:-}\ G, \texttt{xi\_cut}$$

further search for values for the attribute p will cease once `xi_cut` has been executed.

How is this useful ? When defining a world model it is frequently the case that an attribute is defined at one node only, or is defined at several nodes but with the knowledge that it will not be defined at any higher nodes. The world model writer can take advantage of this knowledge to insert `xi_cut` operators into the attribute definitions at this point, ensuring that no search at higher nodes, or along different branches, will take place.

5.3 Changing the World Model

5.3.1 Asserting and Retracting Instances, Classes and Attributes

Generally, XI applications will want to add instances to and remove instances from a world model. They will also want to add and remove attributes associated with instances. In some cases applications will want to dynamically alter higher levels of the ontology and associated attributes. To allow this XI has a number of built-in predicates corresponding to the `assert` and `retract` predicates of Prolog:

$\texttt{xi\_ont\_assert(}E\ \texttt{<--}\ C\texttt{)}$ Assert an instance. Adds a type-2 O-clause to the world model, so C must have the form $c_1 \& \cdots \& c_n$. C must be instantiated. If E is instantiated records that E is an instance of each c_i in C; otherwise generates a new instance id, records each instance fact as before and instantiates E with the new id.

$\texttt{xi\_ont\_assert(}C\ \texttt{==>}\ D\texttt{)}$ Assert a class. Adds a type-1 O-clause to the world model, so D must have the form $D_1 \& \cdots \& D_n$, where each D_j is a disjunction of class terms of the form $d_{j,1} \vee \cdots \vee d_{j,p_j}$. Both C and D must be instantiated.

`xi_prop_assert(`*Node*,*Attribute*`)` Assert an attribute. Adds *Attribute* to the P-clause associated with *Node* if one exists and creates a new P-clause with *Attribute* if not.

`xi_ont_retract(`E `<--` C`)` Retract an instance. Removes a type-2 O-clause of matching E `<--` C from the world model. If, following this retraction, no type-2 O-clauses mentioning E remain, any P-clauses of the form `props(`$E, \mathbf{V}$`)` are also removed.

`xi_ont_retract(`C `==>` D`)` Retract a class. Removes a type-1 O-clause of matching C `==>` D from the world model. If any class term removed in the course of the operation no longer occurs elsewhere in the ontology then any P-clauses associated with it are also removed.

`xi_prop_retract(`*Node*,*Attribute*`)` Retract an attribute. Removes an attribute matching *Attribute* from the P-clause associated with *Node*.

Using these functions it is perfectly possible for an application programmer to mangle a world model into a data structure that no longer meets the various constraints placed upon XI models, or to create, *ex nihilo* a structure that is not in fact a XI model. To attempt to enforce these constraints (e.g. no node may inherit from two higher nodes that are mutually exclusive options in the same classificatory dimension) by building them into each of the predicates described above would have been computationally prohibitive. So, *caveat* hacker. To make matters slightly better there is a predicate `xi_validate_wm(`*Nodes*`)` which fails with *Nodes* bound to an list of nodes implicated in an illegal definition of a world model. This function may be called at any time and is called by default by the compiler (next section) and hence may be used to ensure that at least an initial world model is validly formed.

5.3.2 Contexts

One feature which XI lacks with any degree of sophistication is a notion of 'context'. It would be useful to be able to restrict access to instances (and perhaps to classes) to subsets of the set of all instances held in a given world model. Ultimately, mplementing something like the Lisp notion of packages would useful. This would allow instances to be associated with a named context, and imported and exported between contexts. As with Lisp packages there would be a current context which could be switched with any of a number of alternative user-defined contexts.

A primitive version of this has been experimented with in XI: the predicates `xi_save_context/1`, `xi_restore_context/1`, and `xi_clear_context/0` allow all current instance assertions together with their associated, non-inherited attributes to be saved to a named repository, restored from a named respository, or deleted from the current world model, respectively.

5.4 Input/Output

In addition to the underlying Prolog input/output mechanisms which can be used to display and to write or load world models to or from disk files XI provides a number of useful predicates explicitly for these purposes.

The `xi_print_context/0` predicate displays all current instances and their associated, non-inherited attrributes. The `show_ontology/0` predicate displays the current ontology (but no attribute information). `write_ontology(`*File*`)` writes the current ontology to *File* from whence it may be loaded using a simple Prolog `read`.

`xi_write_uncompiled(`*File*`)` writes the current uncompiled (see next section for a discussion of the XI compiler) world model – all instances, classes and attributes – to *File*; `xi_write_compiled(`*File*`)` writes the current compiled world model to *File*; `xi_write_all(`*File*`)` writes both the compiled and uncompiled world models to *File*. Again any of these may be read back in using Prolog `read`.

6 The XI Compiler

The syntax for writing XI models is designed to make models relatively easy to write or to modify for humans. However, these representations do not lend themselves to particularly efficient processing. For this reason a 'compiler' for XI world models has been written which translates a world model defined in the notation described above into a set of Prolog clauses that permits much more efficient execution. A world model may be precompiled and saved to a disk file to be loaded at run-time; or it may be compiled at any time during execution. A world model may consist partially of compiled clauses and partially of non-compiled clauses – they may cohabit and XI can perform derivations using a mixture of the two.

The compiled representation makes its savings in three ways:

1. for each type-1 O-clause of the form $c \Longrightarrow D_1 \& \cdots \& D_n$, where c is a class term and each D_j is a disjunction of class terms of the form $d_{j,1} \vee \cdots \vee d_{j,p_j}$, `compiled_parent(c,`$\mathtt{d}_{j,k}$`)` clauses are added to the Prolog database for each disjunct on the righthandside of the O-clause. This saves repeatedly picking apart complex righthand side expressions when traversing the hierarchy at run-time;

2. for each type-2 O-clause of the form $e \longleftarrow c_1 \& \cdots \& c_n$ where each c_i is a class term and e is an instance symbol, `compiled_instance(e,`$\mathtt{c}_i$`)` clauses are added to the Prolog database for each conjunct on the righthandside of the O-clause. Again, this saves repeatedly picking apart complex righthand side expressions when traversing the hierarchy at run-time;

3. for each P-clause of the form $\mathtt{props}(t, \{A_1, \ldots, A_n\})$ where t is a node in the ontology and A_i is an attribute, `compiled_prop(t,`$\mathtt{A}_i$`)` clauses are added to the Prolog database for each attribute A_i in the P-clause. This saves repeatedly doing `member` operations to see if attributes occur in P-clauses at run-time;

The XI compiler may be invoked either by issuing the `xi_compile/0` command which compiles the current world model and then retracts any O-clauses and P-clauses or by issuing the `xi_compile(`*File*`)` command which simply calls `xi_compile/0` and then calls `xi_write_compiled(`*File*`)` to write the compiled world model to *File*.

As mentioned above the compilation process begins by default with an attempt to validate the form of the world model. The `xi_validate_wm(`*Nodes*`)` function includes three checks: that no node inherits from two mutually exclusive nodes in a common conjunct of a type-1 O-clause; that no cycles exist in the graph; that all nodes are dominated by the root. The algorithm for this is simple, if inefficient: for each node a list of all of its descendants (classes and instances) is constructed. The third check is carried out by ensuring that each node occurs in the list of descendants of the root. The second is carried out by ensuring that no node occurs in the list of its own descendants. The first is carried out by checking that for each disjunction occurring on the right of any type-1 O-clause no pair of disjuncts has any common descendants.

7 Concluding Remarks

In concluding there are several topics which should be briefly touched upon. These include the issue of default inheritance and XI and the issue of acquiring large scale XI models.

7.1 Default Inheritance and XI

The issue of default inheritance has been a fraught one in inheritance-based KR languages. One of the original motivations for inheritance-based approaches was that they seemed to offer a mechanism for copying with the 'Tweety' problem: the problem of recording that an individual bird or subclass of birds cannot fly, while all recording that in general all birds fly. In inheritance languages the solution was to record at the **bird** node that birds have the attribute of flying, then to record at the **Tweety** node that Tweety does not have this attribute. The lower level fact was intended to 'override' the default and supply the correct result. While this may be satisfactory for single inheritance hierarchies it does not suffice for multiple inheritance networks: conflicting defaults may be inherited from different ancestors (the so-called 'Nixon diamond' problem) and there is no general approach for deciding which to prefer.

XI remains resolutely agnostic on these issues and exports them all onto the user. Using the `nodeprop` predicate the user can determine whether an attribute-value originates at a given node, or is inherited. Thus, if it is desired to let local values override inherited ones, this policy may be pursued. In the case of multiple inheritance, the matter is again in the hands of the user and network designer: notions of semantic distance may be defined and used to arbitrate between conflicting inherited attributes; checks to detect and eliminate possible conflicting attribute inheritance may be implemented.

Thus, XI is advanced simply as a tool to be used in KR applications, not as a solution to outstanding theoretical problems with inheritance-based approaches. The assumption underlying XI is that there are advantages to representing knowledge in an inheritance network (chiefly, perspicuity and nonredundancy) and that a simple, robust tool to be employed in research applications needing a large amount of world knowledge would be of significant experimental value.

7.2 Acquiring World Models

How should world models be acquired ? To date all applications of XI (Gaizauskas et al. (1995), Gaizauskas et al. (1993), Cahill et al. (1992)) have involved hand-crafted world models. While it is certainly possible to construct world models by hand for limited domain applications, it seems increasingly clear that techniques for automated, or at the very least semi-automated, construction of world models will be necessary for some applications. In particular NLP applications such as information extraction from newspaper texts need very large amounts of world knowledge to be able to perform tasks such as coreference resolution.

To this end several approaches are currently being experimented with. One is to attempt to populate a XI world model using existing resources such as LDOCE and WordNet both of which contain large scale ontologies. Another is to build an interactive interface which allows an ontological engineer to interact with an NLP system which reads texts and then tries to place lexical items (after morphological analysis) into an existing XI world model. Starting with a skeleton high level model the user directs the construction of the world model allowing the system to make hypotheses about where new terms should be placed based, for example, on subcategorisation information (e.g. if the term *gronk* is encountered in the sentence *The gronk died* and *die* is already stored in the model as an event with an attribute indicating the class of its logical subject is `animate`, then *gronk* can be placed in the ontology beneath the `animate` node; later perhaps its position can be refined).

Much further work is of course needed to see if this approach is feasible. But XI provides the framework within which such experimentation, and also experimentation with coreference resolution techniques and algorithms, becomes possible.

8 Acknowledgements

Initial work on XI was carried out at the University of Sussex while the author was working on the POETIC project, a UK DTI-EPSRC sponsored project (Grant Ref:) for extracting information pertaining to road traffic incidents from police logs. The work was continued at Sussex as part of the Sussex MUC-5 System development effort, sponsored by the US Advanced Research Projects Agency, Sussex University, Racal Research and Integral Solutions Limited. During this time Roger Evans provided useful comments and helped to write the original version of the XI ontology compiler. In 1993 the author moved to the University of Sheffield and carried on work on XI there. Following the commencement in 1995 of a UK EPSRC grant to build a large scale information extraction system (Grant Ref:), the XI code was reimplemented and extended by Kevin Humphreys whom the author would like to thank in particular. Useful comments have been provided on drafts of this paper by Matt Fairtlough and Kevin Humphreys.

References

Cahill, L., Gaizauskas, R., & Evans, R. (1992). POETIC: a Fully-Implemented NL System for Understanding Traffic Reports. In *Fully Implemented Natural Language Understanding Systems: Proceedings of the Trento Workshop, March 30, 1992 (IWBS Report No. 236)*, pp. 86–99 IBM Institute for Knowledge Based Systems, Heidelberg.

Gaizauskas, R., Cahill, L., & Evans, R. (1993). Description of the Sussex System Used for MUC-5. In *Proceedings of the Fifth Message Undersanding Conference (MUC-5)*. ARPA, Morgan Kaufmann.

Gaizauskas, R., Wakao, T., Humphreys, K., Cunningham, H., & Wilks, Y. (1995). Description of the LaSIE System as Used for MUC-6. In *Proceedings of the Sixth Message Undersanding Conference (MUC-6)*. ARPA, Morgan Kaufmann. Forthcoming.

Smullyan, R. (1968). *First-Order Logic.* Springer-Verlag, Berlin.

Appendix XI: The Code

A XI software package containing the code described here is available via

`http://www.dcs.shef.ac.uk/~robertg/`

As is, the code requires Quintus Prolog but it is easily ported to SICStus Prolog and others.

```
/* ----------------------------------------------------------------------
   > File: xi.pl
   > Purpose: defines the fundamentals of the Xi KR language
   > Author: K.Humphreys, R.Gaizauskas, R.Evans
   -----------------------------------------------------------------------*/

:- op(20,xfy,v). % disjunction of categories
:- op(10,xfy,&). % conjunction of categories


% O-clauses:
:- op(40,xfx,==>). % X ==> Y: Y is an immediate subclass of X
:- op(30,xfx,<--). % X <-- Y: X is an immediate instance of Y

X ==> Y :- compiled_parent(Y,X).
X ==> Y :- !, clause(X1 ==> Y1, true), indefn(X,X1), indefn(Y,Y1).

X <-- Y :- compiled_immediate_instance(X,Y).
X <-- Y :- !, clause(X <-- Y1, true), indefn(Y,Y1).


% G-clauses:
:- op(30,xfx,=>). % X => Y: Y is a subclass of X
:- op(30,xfx,<-). % X <- Y: X is an instance of Y


X => Y :- nonvar(Y), !, parent(Y,P), (X = P;X => P).

X <- Y :- compiled_instance(X,Y). % use compiled version if available.
X <- Y :- clause(X <-- Z, true), indefn(Z1,Z), (Y = Z1;Y => Z1).


% find all properties (nodeprops returned first)
hasprop(E,P) :- var(P), !,
        findall(PP, ((nodeprop(E,PP);inheritprop(E,PP)), PP \= (_ :- _)), PPs),
        no_doubles_unify(PPs,Ps), % delete equivalent props found by
        member(P,Ps).             % different paths
% find specific property (nodeprops returned first)
hasprop(E,P) :- (nodeprop(E,P) ; inheritprop(E,P)).

% nodeprop(E,Prop) : Prop is a direct property at E (not inherited)
```

```
%
nodeprop(E,Prop) :- compiled_prop(E,Prop).
nodeprop(E,Prop) :- props(E,Ps),
        ( member(Prop,Ps) ; (member((Prop :- Goal),Ps), call(Goal)) ).

% inheritprop(E,Prop) : Prop is an inhertited property of E
%
inheritprop(E,Prop) :- E <- Node, inheritprop(E,Node,Prop). % E is an instance
inheritprop(E,Prop) :- \+(E <-- _), Node => E,              % E is a node
                       nodeprop(Node,Prop).
inheritprop(E,Node,Prop) :- Node =.. [_,E|_], nodeprop(Node,Prop).


% parent(N,P) : true if N is an immediate subclass of P
%
parent(N,P) :- compiled_parent(N,P). % use compiled version if available
parent(N,P) :- clause(P ==> N, true).
parent(N,P) :- clause(P ==> X, true), indefn(N,X).
parent(N,P) :- clause(P1 & P2 ==> X, true),
               (indefn(P,P1);indefn(P,P2)), indefn(N,X).

% child(N,C) : true if C is an immediate subclass of N
%
child(N,C) :- parent(C,N).


% indefn(Y,Z) : true if Y is a term occurring in a sentence
%               made up of terms composed with v's and &'s.
%
indefn(_,Z) :- var(Z), !, fail.
indefn(Y,Z & W) :- indefn(Y,Z);indefn(Y,W).
indefn(Y,Z v W) :- indefn(Y,Z);indefn(Y,W).
indefn(Y,Z) :- Z \= _ & _, Z \= _ v _, Y = Z, !.


% path(X,Y,Z) : Return a path Z (list of nodes) between nodes X and Y,
%               if one exists. X may occur above or below Y. Multiple
%               paths are returned on back tracking.
%
path(X,Y,Z) :- path1(X,Y,Z).
path(X,Y,Z) :- path1(Y,X,Z1), reverse(Z1,Z).

% path1 returns a path from X to Y IF Y dominates X.
path1(X,Y,_) :- (var(X);var(Y)), !, fail.
path1(X,Y,[X]) :- X = Y, !.
path1(X,Y,[X|P]) :- X <-- Z, indefn(Z1,Z), path1(Z1,Y,P).
path1(X,Y,[X|P]) :- Z ==> X, indefn(Z1,Z), path1(Z1,Y,P).
path1(X,Y,[X|P]) :- parent(X,Z), path1(Z,Y,P).
```