COSC460 Project Report
Department of Computer Science
University of Canterbury

ANALYSIS & COMPARISON
OF

KNOWLEDGE REPRESENTATION SYSTEMS

Juerg C Daellenbach

7 October 1983
ACKNOWLEDGEMENT

I would like to thank my supervisor Wolfgang Kreutzer for providing the resources and the assistance required to produce this report.
"Intelligent computer programs must have both a representation of the knowledge they involve and some mechanisms for manipulating that knowledge for certain purposes."

N. Findler

Knowledge representation is an active field of Artificial Intelligence research. The method of representation strongly influences the possible types of processing of the knowledge base. The knowledge base and the mechanisms for its manipulation determine the intelligence of a computer program. Therefore a powerful representation system is desired to enable the mechanisms to operate at full potential. Numerous approaches to representing knowledge have resulted in a wide range of knowledge representation (KR) formalisms. At one end of the scale are the simple techniques which give little or no indication of the structure of knowledge, such as LISP Property Lists. At the other extreme are semantic networks, frames, actors, predicate calculus and production systems. The distance between them is large - the designers had different ideas of how to best represent the knowledge when the systems were developed.

Many of the earlier attempts, such as the associational network model of "semantic memory" proposed by M. Quillian [Quillian 66], provided the groundwork for later KR languages. Quillian's model used associative links connecting nodes in order
to permit "human like use" of the meanings of the words, and closely reflected the organisation of a dictionary. Further improvements lead to hierarchies of concepts with attached properties [Collins & Quillian 70]. Predicate logic systems, such as PROLOG [Colmerauer et alia 73], developed independently, and were oriented towards using facts to make deductions. Production systems were first proposed in 1943 by E. Post [Post 43]. These were based on productions, a set of conditions and an action, which work on a knowledge base. Some non-network formalisms like KRL [Bobrow & Winograd 77] and FRL [Roberts & Goldstein 77] share many features with semantic network systems such as KLONE [Brachman 79].

The aim of this project is to determine the differences between the formalisms. This includes determining which features are common to all formalisms, which features are unique, the methods of representing knowledge, and some measure of the power of each formalism.

The large number of formalisms and independent research has lead to a wide range of terms for very similar items. To avoid confusion, some clarification of terminology is required. A class is a general concept which usually consists of the common features, called attributes, of a group of concepts. This is also called a generic or a prototype. An example is person, which has attributes such as name and age. An instance of a class is a single concept which is a member of the group of concepts of the class. It has the attributes of the class concepts. It is also called an individual or an instantiation of the class. An example is the person Juerg Daellenbach, who is a person with a name and an age. A subclass is a class which contains all the attributes of another class, and some more of its own. It is also called a further specification. An example is student, which is a subclass of person, and has the additional
attributes of course and stage. Classes, instances and subclasses build hierarchies, with classes at the top, and subclasses and instances below each class. Some formalisms do not distinguish between instances and subclasses; every instance is treated as a subclass and can be further specified. A relation is a mapping between several classes. Most relations are binary, which means there is a domain class and a range class. A more complex relation involves a cross-product domain. An example is the grade of a student with respect to a course. The student/course cross-product maps to the range which is the mark. An instance of a relation is an occurrence of a relation between two or more instantiations. It is called an assertion or a proposition.

A Frame [Minsky 75] is a data structure for representing a stereotyped situation. It can be thought of as a hierarchy of nodes and relations. The "top levels" represent things that are always true about the situation, and are fixed. The lower levels have many storage locations called slots that are filled by specific instances or data, subject to certain restrictions.

The term semantic network was derived from the early applications of associative networks, working with the meanings of words stored and used in the net. However the term is now used for any associative net regardless of the types of concepts stored in the KR language.

A demon is an active process with a set of requirements. It constantly checks for the satisfaction of the requirements using the values in the knowledge base, and they are satisfied, its actions are performed. The demon is independent of any other processes running "at the same time". Once a demon is started it remains active until it terminates itself. The only control over a demon is by setting the values in the knowledge base to force or prevent satisfaction of the requirements.
ANALYSIS

Due to the quantity of KR formalisms in existence, only a small cross-section of these are analysed. A procedural semantic network, a frame based system, a production system, and a predicate logic system. The four sections of this chapter contain the analysis of these four formalisms.

A small model of a university environment is used as an example, and for each formalism an example of the syntax and style of the language is provided by declaring and interrogating this knowledge base. All efforts were made to define the same model for each formalism, but in some situations this was not possible. A description of the model is provided in Appendix A.
PSN: A Procedural Semantic Network

Introduction

The Procedural Semantic Network (PSN) formalism was developed by H. Levesque [Levesque 77] of the University of Toronto to overcome some inadequacies he felt existed in pure semantic network systems. These systems limited the semantic links to a number of system-supplied links and provided an interpreter to interpret the semantics of these links. The procedural semantic network formalism allows the structure of a network to be defined at the nodes of the network, and the links to be interpreted by procedures attached to the relations of which the links are instances.

The implementation examined [Kramer & Graham 79] is an extension to a LISP system although future development includes mapping programs written in a PSN language into appropriate LISP code.

Description

A PSN consists of nodes which are known as objects, and unlabelled arcs between them. Arcs are objects which are instances of a special class called RELATION or subclasses of this class. CLASSES are objects, and RELATIONS, NUMBERS, STRINGS, LISTCLASSES and PROGRAMS are classes. The system includes some basic relations for creating subclasses of a class (IS-A), decomposing an object into its subparts (PART-OF), and binding objects to the class for which it is an instance (INSTANCE).
An object consists of a number of storage fields called structural attributes or slots. An instance of a class will inherit the structural attributes and will give them values which must be supplied at the time of creation. A slot consists of four parts: its name as supplied by the user, its TYPE: which is an object, a DEFAULT: value, and a RESTRICTION: which is a program, represented by a LISP function.

PSN provides inheritance by the IS-A and INSTANCE links. Subclasses may refine structural attribute properties of a class and supply slot values other than those of the class. This is because the structure of the class is duplicated in the subclass. A class can be a subclass of several other classes to provide more flexible inheritance capabilities, but an instantiation is only an instance of one class.

The PSN formalism requires that four programs are attached to each class and relation. The programs are used to create, destroy, test for and find instances of the class or relation. The predefined classes CLASS and RELATION have these programs already supplied, and any subclasses of these two classes will inherit the programs if no other programs are specified. The future extensions will allow these programs to be defined by the user, and to be subclasses of the predefined class PROGRAM. The Extended Procedural Semantic Network (EPSN) as described by B. Kramer [Kramer & Graham 79] defines an instance of a program as consisting of five slots: parameters, locals, prerequisites, body and "return." The prerequisites are evaluated and must be satisfied for the body to be performed, otherwise a failure occurs. If the prerequisites are true, the actions in the body are performed, which may involve using values in the parameters and locals slots. The action in the return slot is performed after successful completion of the body. If a prerequisite fails or some action in the body fails then a complaint is generated.
In EPSN each slot in the prerequisite, body and return parts of a program may have an exception class associated to it via an exception link. An exception class is an instance of the EPSN predefined class EXCEPTIONS. A complaint causes the creation of an instance of the EXCEPTION class, which implies that the exception handler is the TOADD program of this class. Since programs and exceptions are classes, they may have inheritance and instantiations as defined for any other class.

The structures of objects are defined as follows. Text after a semicolon on each line is a comment.

For subclasses:

classnamel ; the name of the class

ISA: ( class1 class2 ... ) ; its heritage
STRUCTURE: ; its slots
  ( slot1:
    TYPE: class3
    DEFAULT: value1
    RESTRICTION: program1
  slot2:
    TYPE: class4
    DEFAULT: value2
    RESTRICTION: program2
    ... more slots ... )
Instances:
  ( instance1 instance2 ... )
Subclasses:
  ( subclass1 subclass2 ... )
The last two properties are added by the system to maintain lists of instances and subclasses of each class. All classes are subclasses of the predefined CLASS.

For instances of a class:

`instancenamel`

```plaintext
INSTANCEOF: class1
VALUES: ( slot1: value1 slot2: value2 ...
```

For relations:

`relationnamel`

```plaintext
ISA: RELATION
VALUES:
( DOMAIN: class1
 RANGE: class2 )
```

For assertions:

In PSN assertions are links between two objects, and do not have names.

```plaintext
relationnamel
( domaininstancename
 rangeinstancename )
```
Example

The following example defines the university mini-world and a simple enquiry. The system prompt is a colon. Text after a semicolon on each line is a comment.

:(de setup nil ; this sets up the mini-world

(setq person ; define the class person
 (CREATE CLASS
  (list 'STRUCTURE:
   ((name: TYPE: STRING)
    (age: TYPE: NUMBER)))))

(setq department ; define the class department
 (CREATE CLASS
  (list 'STRUCTURE:
   ((name: TYPE: STRING
     DEFAULT: "Computer Science")))))

(setq courses ; define the class courses
 (CREATE CLASS
  (list 'STRUCTURE:
   ((list 'dept: 'TYPE: department
     'DEFAULT: cosc)
    (list 'code: 'TYPE: NUMBER)
    (list 'points: 'TYPE: NUMBER
     'DEFAULT: 6
     'RESTRICTION: validate)
    ; restrict values allowed
    (list 'desc: 'TYPE: STRING)))))
(setq student ; define the class student
   (CREATE CLASS
    (list 'ISA:
      person ; a student is a person
      'STRUCTURE:
      ((list 'year: 'TYPE: NUMBER
        'RESTRICTION: validate)))))

(setq enroll ; student / course relationship
   (CREATE RELATION
    (list 'VALUES:
      (list 'DOMAIN: student
        'RANGE: courses)))

(setq prereq ; course / course relationship
   (CREATE RELATION
    (list 'VALUES:
      (list 'DOMAIN: courses
        'RANGE: courses)))

(setq cosc ; instances of departments
   (CREATE department))

(setq math
   (CREATE department
    '(VALUES: (name: "Mathematics"))))

(setq psyc
   (CREATE department
    '(VALUES: (name: "Psychology"))))

(setq cosc111 ; instances of courses
   (CREATE courses
    '(VALUES: (code: 111
      "COSC 111")})
(setq cosc112
  (CREATE courses
    '(VALUES: (code: 112
                desc: "Computer Science 1")))))

(setq cosc203
  (CREATE courses
    '(VALUES: (code: 203
                desc: "Information Processing Methods")))))

(setq math241
  (CREATE courses
    (VALUES:
      (name: "Abstract Algebra"))))

(setq s1
  (CREATE student
    '(VALUES: (name: "John Brown"
               age: 18
               year: 1)))))

(setq s2
  (CREATE student
    '(VALUES: (name: "Anne Smith"
               age: 19
               year: 1)))))

(setq s3
  (CREATE student
    '(VALUES: (name: "Andrew Black"
               age: 20
               year: 2)))))
(CREATE enroll ; assert some enrollments
  (list s1 coscll1))
(CREATE enroll
  (list s1 coscll2))
(CREATE enroll
  (list s2 coscll1))
(CREATE enroll
  (list s3 cosc203))
(CREATE enroll
  (list s3 math241))

(CREATE prereq ; assert some prerequisites
  (list cosc203 coscll1))
(CREATE prereq
  (list cosc203 coscll2))

nil) ; this function returns nil

:(de validate (slotname slotvalue) ; restriction program
  (cond ((equal slotname year:) (greaterp slotvalue 0))
    ; year: must be positive
    ((equal slotname points:) (member slotvalue (list 4 6 12)))
    ; points must be 4, 6 or 12
    (t t))) ; otherwise true

:(de printenroll (acourse) ; a sample enquiry function
  (prog (people)
    (setq people ; bind people to a 'generator'
      (FETCH enroll ; which returns students doing

12
(list 'RANGE: acourse)); the course
(return
((label enrollment
  (lambda (stud)
    (cond ((null stud) nil)
      (t (cons
          (SFPSR stud 'name:)
          ; ^ this gets value from name:
          (enrollment
           (car (PSNGET people)))))
    ; this gets next student doing
    ; course from the 'generator'
  ))
  (car (PSNGET people)))))

:(setup) ; set up the mini-world
NIL

:(printenroll cosclll) ; get cosclll student names
("John Brown" "Anne Smith")

:(printenroll math241) ; get math241 student names
("Andrew Black")

The relationship between departments and the courses they
offer is not represented as a relation in this example, but as a
property of a course. This demonstrates the flexibility of this
KR language. The advantage of using the class RELATION in this
situation is that the inverse can be used to find the courses
offered by a particular department without requiring the user to
write a search function, which is required by the above
representation.
Summary

The PSN implementation examined provides features for defining, creating, altering, retrieving and removing objects. Inheritance is provided by the IS-A construct, and instantiation by the INSTANCEOF property. The predefined classes are CLASS, RELATION, PROGRAM, NUMBER, STRING, LISTCLASS and NUMBERPAIR, although for these last four only the programs to test for an instance of the class may be used.

Each class uses four attached programs to create an instance of the class, fetch all instances of the class, test whether an object is an instance of the class, and to remove an instance of the class. If a class is removed, all instances of the class and all its subclasses are also removed. Other operations are provided by predefined PSN functions. These include search function generators, functions to delete slots from a structure, merge a structure with the structures of its IS-A parents, and also functions to display and save the knowledge base. These functions are normal LISP functions and can be used in any user-written functions.

The user must use these programs and functions, as well as the standard LISP functions, to navigate through the network and use the information it contains. Therefore the user requires some knowledge of LISP in addition to knowledge about PSN. The planned extensions of EPSN include user-written programs and a better enquiry language, such as:

FOR EACH x IN <class or relation> SUCH THAT <predicate> DO

which is easier to use than the SFPSR & PSNGET constructs required in the example above.
FRL : A Frame Representation Language

Introduction

The Frame Representation Language (FRL) was developed by R. Roberts and I. Goldstein [Roberts & Goldstein 77; Goldstein & Roberts 77] of the Massachusetts Institute of Technology based on the concept of frames as suggested by Minsky [Minsky 75]. The implementation examined is an extension to LISP.

Description

A frame in FRL consists of extended Property Lists which allow properties to have comments, defaults and constraints, to inherit information from abstract forms of the same type, and to have attached procedures triggered by adding, deleting or retrieving values.

The frame concept corresponds to the concept of classes in PSN. The fields in a frame are also called slots but are more complex than the slots in PSN. Each slot has facets, and each facet contains data. The name of the facet determines to some extent how the data is to be used.

FRL provides inheritance by the AKO slot. A class can be a subclass of several other classes to provide more flexible inheritance capabilities. A subclass will inherit the slots of its superclasses but in contrast to PSN the inherited slots are not duplicated in the subclass so as to save on storage. There is no distinction between classes and instances of classes at the definition level as exists in the PSN formalism; both are
declared as frames, and can be further specified using AKO slots in other frames. Instantiations are indicated by the value of the CLASSIFICATION slot as described below. When specifying the inheritance hierarchy of a class using the AKO links, the class can be treated as either a further specialisation of some classes or an actual member of the classes. There is a special slot which distinguishes those frames containing information about a generic class of objects from those defining a single individual. This slot, the CLASSIFICATION slot can have one of two values: INDIVIDUAL for instantiations, or GENERIC for classes. This slot is recognised by two FRL predicates: (INDIVIDUAL? <frame>) and (GENERIC? <frame>). This slot is only provided for the user's convenience; it has no other effect outside these two predicates. The INSTANCE slot of a frame contains a list of all frames which are further specialisations of that frame. This is the inverse of the AKO slot. These three slots (AKO, CLASSIFICATION, INSTANCE) are the only slot names with special properties. Any other slot in the frame which the user defines, is for storing information.

Each slot in a FRL frame has up to six facets recognised by the system. These facets are $VALUE, $DEFAULT, $IF-ADDED, $IF-REMOVED, $IF-NEEDED and $REQUIRE. The $VALUE facet holds the values of a slot. The $DEFAULT facet holds the defaults for the slot if no $VALUE facet is supplied. The $IF-NEEDED facet contains a LISP form which is evaluated whenever data is added to the $VALUE facet of the slot. Similarly the $IF-REMOVED facet contains a LISP form for evaluation when data is removed. The $IF-NEEDED facet contains a LISP procedure which is invoked when data is retrieved from the slot. The $REQUIRE facet contains LISP predicates which describe allowable values for the slot. This is used to place restrictions on frames in a similar manner to the RESTRICTION: part of slots in the PSN formalism. There is
one predefined frame in FRL: THING. This frame contains the special slots and their facets, so that any frame can be easily defined using THING as a base.

The $VALUE facet of a slot holds the data to be retrieved for the slot. The user can specify if the data is to be evaluated, used as a pointer for indirection, or returned as is, when retrieving it. The only restrictions on the data are those specified by the user in the $REQUIRE facet. Each datum can have a number of comments associated with it. The user can retrieve the comments with the datum if desired.

A comment consists of a label followed by some number of messages. Three labels are recognised by FRL. The IN: label indicates that the following message is the name of the frame in which the datum is stored. This makes it possible for the user to trace the origins of any retrieved datum. This comment is added automatically by FRL when the data is first accessed. The FINHERIT: label can only have one recognisable message: CONTINUE. This tells the search function to return this data but still continue searching along the AKO link for more data. The TYPE: label is recognised by the retrieval function and indicates which of the $IF-NEEDED procedures associated with the slot are to be attempted. Recognised messages include NONE, IMMEDIATE, and DEFAULT.

The name of the frame is the first element. Facet names conventionally have a prefix "$"; labels, a suffix ":". Allowable names for an indicator are any LISP s-expression.

The structure of a frame is:

(framel
  (slotl (facetl (datuml (labell messagel

17
A frame can have any number of slots, and each slot can have any number of facets. Only six facets are used by the FRL functions. Any other facets which have been declared can be used as path names for enquiries. Comments are optional, as are multiple messages.
Example

The following example defines the university mini-world and a simple enquiry. It is used to demonstrate the format of the declarations and the structure of enquiries. The text after the semicolon on each line is a comment. The system prompt is a colon.

:(de setup nil ; this sets up the model

  (FASSERT person ; define the frame person
      (name ($REQUIRE ((stringp :VALUE))))
      (age ($REQUIRE ((greaterp :VALUE 0)))))

  (FASSERT department ; define the frame department
      (name ($DEFAULT ("Computer Science"))))

  (FASSERT cosc ; a department
      (AKO ($VALUE (department))))

  (FASSERT math
      (AKO ($VALUE (department))
         (name ($VALUE ("Mathematics"))))

  (FASSERT psyc
      (AKO ($VALUE (department))
         (name ($VALUE ("Psychology"))))

  (FASSERT courses ; define the frame courses
      (dept ($REQUIRE ((AKO? :VALUE 'department)))
         ; must be a department
         ($DEFAULT (cosc)))


19
(FASSERT student ; define the frame student
  (AKO ($VALUE (person)))
  ; a student is a person
  (year ($REQUIRE ((greaterp :VALUE 0))))
  (course ($REQUIRE ((AKO? :VALUE courses))))
  ; must be a course

(FASSERT cosclll ; a course
  (AKO ($VALUE (courses)))
  (code ($VALUE (111)))
  (desc ($VALUE ("Computing 1"))))

(FASSERT coscll2
  (AKO ($VALUE (courses)))
  (code ($VALUE (112)))
  (desc ($VALUE ("Computer Science 1"))))

(FASSERT cosc203
  (AKO ($VALUE (courses)))
  (code ($VALUE (203)))
  (desc ($VALUE ("Information Processing Methods"))
  (prereq ($VALUE (cosclll coscll2)))))

20
(FASSERT math241
  (AKO ($VALUE (courses))))
  (dept ($VALUE (math)))
  (code ($VALUE (241)))
  (points ($VALUE (4)))
  (desc ($VALUE ("Abstract Algebra"))))

(FASSERT s1 ; a student
  (AKO ($VALUE (student)))
  (name ($VALUE ("John Brown")))
  (age ($VALUE (18)))
  (year ($VALUE (1)))
  (course ($VALUE (cosc111 cosc112))))

(FASSERT s2
  (AKO ($VALUE (student)))
  (name ($VALUE ("Anne Smith")))
  (age ($VALUE (19)))
  (year ($VALUE (1)))
  (course ($VALUE (cosc111))))

(FASSERT s3
  (AKO ($VALUE (student)))
  (name ($VALUE ("Andrew Black")))
  (age ($VALUE (20)))
  (year ($VALUE (2)))
  (course ($VALUE (cosc203 math241))))

nil) ; this setup function returns nil
(:de stripname (peoplelist) ; get the names from the frames
  (cond ((nil peoplelist) nil); no more names
    (t (cons (FGET (car peoplelist) 'name '$VALUE)
        ; get the value in the name slot
      (stripname (cdr peoplelist)))))))
    ; and the rest of the names

(:de printenroll (acourse) ; a sample enquiry function
  (prog (people)
    (setq people ; bind people to a list of frames
      (FQUERY '(? (course ($VALUE (acourse)))"
        ; retrieves all frames with the
        ; value acourse in the course slot
      (return (stripname people)))))
      ; return names, not whole frames

(:de printenroll2 (acourse) ; another version using select
  (mapcar (FRINGE student 'INSTANCE)
    ; FRINGE gets the frames at the
    ; fringe of the network that are
    ; reachable from the given frame
    ; ie. it gets all instances of
    ; students that are not further
    ; specified
    (function (lambda (stud)
      (cond ((member acourse
        (FGET stud 'course '$VALUE))
          (FGET stud 'name '$VALUE))
        ; FGET gets the data using the
        ; specified path

22
(t nil)))))

:(setup) ; set up the mini-world
NIL

:(printenroll cosclll) ; get cosclll student names
("John Brown" "Anne Smith")

:(printenroll math241) ; get math241 student names
("Andrew Black")

Summary

The FRL system provides features for defining, creating, altering, retrieving and removing objects. Inheritance is provided by the AKO slot, and instantiation by the value INDIVIDUAL in the CLASSIFICATION slot. The predefined frame is THING, which has procedures for adding and removing instances of a THING attached to the INSTANCE slot. These are inherited by any frame declared to be A Kind Of (AKO) THING.

Each slot in a frame can have procedures attached to add, fetch and remove data from the slot. No indication is given of what happens when a frame with a non-empty INSTANCE slot is removed. Several search functions are provided by the system; which was originally embedded in M.I.T.'s MACLISP.

The user can use these functions to navigate through the network and use the information it contains. All standard MACLISP functions can also be used. A degree of familiarity with LISP is required for reasonably complex enquiries.
OPS : A Production System

Introduction

A production system such as OPS is a program composed entirely of conditional statements called productions. These productions operate on expressions stored in a global data base containing elements which represent facts about some mini-world model. The production consists of a set of conditions and a set of actions. The actions are executed when the data base is in a state such that the conditions of the production are all true simultaneously. The data base and the productions form the knowledge base in this formalism.

The production systems examined are OPS3RX developed by M. Rychener [Rychener 79], and OPS5 developed by C. Forgy [Forgy 81], which are part of a series of production systems to come from Carnegie-Mellon University.

Description

The condition part of a production is called its left hand side (LHS), and the action part is called its right hand side (RHS). The production system interpreter executes a production system by performing a sequence of operations called the recognise-act cycle:

1. [Match] Evaluate the LHSs of the productions to determine which productions are satisfied given the current contents of the data base.
2. [Conflict resolution] Select one production with a satisfied LHS. If no productions have satisfied LHSs, halt
the interpreter. If several productions are satisfied select the "best" one.

3. [Act] Perform the actions specified in the RHS of the selected production.

4. Go to step 1.

The recognise-act cycle is repeated until there are no productions to apply, or the loop is broken because the goal is satisfied. In each case the user can then continue enquiring.

A major difference between production systems and conventional programs is the way flow of control is managed. A conventional program uses sequential execution of statements plus a number of control constructs including function calls, loops, and conditional branching, as is the case with FRL and PSN. A production system uses LHS satisfaction and production selection. Thus each production can be thought of as a demon, waiting for a set of conditions about the data base to become true, and then attempting the perform some action if no other demon, with a higher priority, alters the data base first. The priority of a production varies depending on how recently it was last used, how many conditions it has, and even the age of the information it uses to satisfy a condition.

The global data base contains information represented by elements, consisting of an object and a collection of associated attribute-value pairs. All values are LISP symbolic atoms. There are two types of attributes: scalar and vector. Each scalar attribute can hold exactly one value. A vector attribute can hold from one to 126 values. The name of an element is always stored with the element by the OPS5 system. This allows relations to be represented in two ways. A separate element with three scalar attributes, the name of the relation, the domain element, and the range element can be used to represent an assertion, an instance of a relation. Alternatively if only one
type of relation operates on an element, the range can be stored in a vector attribute.

The format of an element is:

(elementname
  ^scalar-attribute1 value1
  ^scalar-attribute2 value2
  ^scalar-attribute3 value3
  ... more scalar-attribute value pairs ...
  ^vector-attribute1 valuev1 valuev2 valuev3 ...
)

The prefix ^ is used to distinguish attributes from values. In OPS5 there is a limit of 126 values per element. An element can only contain one vector attribute. OPS5 does not provide any mechanism for default values; the default value for any attribute is nil.

The format of a production is:

(p productionname1
  condition1
  condition2
  ... more conditions ...
  -->
  action1
  action1
  ... more actions ...
)

A condition can be a condition element or an element variable. A condition element consists of an open parenthesis, a number of terms to specify attributes and values, and a close parenthesis. An element variable is a mechanism for binding a variable to a database element. The database element is selected by a
condition element.

The format of an element variable is:

\[ \{ \text{<variable1>} \ \text{condition-element1} \} \]

A condition element is considered to match a data base element if every term in the condition element matches the corresponding part of the data base element.

The formats of a term are:

\text{value1} \quad \text{-- this must match the data base element name.}

\text{^attribute1 \ value1} \quad \text{-- the value must match the value in that attribute of the data base element.}

A value can be a constant, a variable, a predicate, or a set of values. A variable will be bound to the value of the matching data base element. A predicate specifies a condition on the values of the data base element if it is to be a match. A set of predicates is treated as a conjunction, a set of constants is a disjunction.

The formats of a value are:

\text{constant1} \quad \text{-- a constant can be a symbolic atom or a number.}

\text{<variable1>} \quad \text{-- a variable.}

\text{<<constant1 \ constant2 ...>>} \quad \text{-- any of the given constants is acceptable as a match.}

\text{operator1 \ constant1} \quad \text{-- the predicate when applied to the constant produces a value which must match that of the}
data base element.

operator1 <variable1> -- similar except the variable
{ valuel value2 ... } -- a conjunction. Any variables

must have been bound to a
value before this predicate.

a conjunction. Any variables
will be bound to the value of
the attribute of the data base
element which matches all the
other requirements of the
conjunction.

For example, the following condition element would match any
block which has some name, is red or blue, has a mass greater
than 20, and a width of 5. The variable x is bound to the value
of the attribute ~name of the knowledge base element which
matches the other conditions. Similarly the variable y will be
bound to the colour of the block.

(block ~name <x> ~colour {<<red blue>> <y>} ~mass >20 ~width 5)

An action can be any of the twelve action type provided by
OPS5. These are to make, remove and modify knowledge base
elements, actions to manipulate files, bind values to variables
and call user-written subroutines, halt to stop the interpreter's
recognise-act loop, and build to add productions to the system.

In the LISP based OPS5 the top level is the normal LISP top
level. This means that any LISP command may be issued. The top
level routine of the production system is the sequence:

1. Read a command from the user.
2. Execute the command
3. Wait for the system to reach a state where no more
   productions have satisfied LHSs, or a halt is performed.
4. Go to step 1.
The predefined OPS5 top level commands include the actions allowed on the RHS and some additional control commands. After the knowledge base is set up, with all elements created and productions defined, the majority of the commands issued by the user are commands which state the next goal, by modifying the attribute-values of an element in the data base which the productions are scanning. This is demonstrated in the example below. All the productions require the element goal to contain special values before their LHSs can be satisfied. This is the main control mechanism available to the user.

The user-written subroutines can interact with the OPS5 system by using some predefined functions to work with the knowledge base. However it is not possible to attach user-written subroutines to global data base elements which are executed when creating, removing, updating, and retrieving an element. The effects can only be achieved, for the first three operations, by the user supplying the subroutines required and the calls to execute them in any RHSs when an action of that type is desired. There is no equivalent option for retrieving.

There is also no facility to provide meaningful defaults for attributes, or to place restrictions on the type and range of values an attribute may contain. This implies that the user must ensure that only the desired type of values are put in the fields, which could be achieved in most cases by calling user-written functions in the actions which return the correct type of result, or produce error messages. Similarly the user can make the condition elements allow nil as well as some other values, because nil is the default, and then call a function in the actions to return the desired default. It would be preferable to have these features built into the language.
Example

The following example defines the university model and some sample productions. The first part creates the data base, the second part defines some productions, and the third part shows some results. The text after the semicolon on each line is a comment. The system prompt is a colon.

:(literalize department ; declare attributes of departments
  kind
  name
  status)

:(literalize courses ; declare attributes of courses
  kind
  dept
  code
  points
  desc
  status)

:(literalize student ; declare attributes of student
  kind
  name
  age
  year
  status)

:(literalize goal ; declare attributes of goal
  type
  object
  status)
:(vector-attribute prereq) ; vector declarations are global
:(vector-attribute course)

:(make department ; make the three departments
  ^kind cosc
  ^name "Computer Science")
:(make department
  ^kind math
  ^name "Mathematics")
:(make department
  ^kind psyc
  ^name "Psychology")

:(make courses ; make the four courses
  ^kind cosc111
  ^dept cosc
  ^code 111
  ^points 6
  ^desc "Computing 1")
:(make courses
  ^kind cosc112
  ^dept cosc
  ^code 112
  ^points 6
  ^desc "Computer Science 1")
:(make courses
  ^kind cosc203
  ^dept cosc
  ^code 203
  ^points 6
  ^desc "Information Processing Methods"
The first statements assign attributes to numbers. In this
example the attribute **kind** is used to store the object value of an element, which corresponds to the name of a frame in FRL or the name of an instance of a class in PSN. The goal is declared here to be a search for cosc111, and is currently active. These values were chosen because of the production below; normally more information would be specified, such as printing of student names as the ultimate goal.

```lisp
:(build p printenroll  ; prod. to find students doing course
 (goal ^status active ^type find ^object <x>)
 { <s> (student ^name <n> ^course <x> ^status nil) }
 -->
 (modify <s> ^status seen) ; mark student as "seen"
 (write (crlf) <n>)) ; write name on next line
```

```lisp
:(build p tidyup        ; reset the students status
 (goal ^status satisfied)
 { <s> (student ^status seen) }
 -->
 (modify <s> ^status nil))
```

This production will be fired if the goal state is active and the type of goal is a find, and if there is a student who's status is nil. This default value for the attribute is true initially for all three students. However once the recognise-act cycle starts, each student who is doing the course specified in the goal's object attribute will be found and have its status set to seen. This prevents the element from being selected more than once by this production. The name of each selected student is also displayed. Once all the students doing the course have been selected, the production system will halt because there are no productions to apply to the data base. At this point the goal
status should be changed to force the tidyup production to restore the data base to its previous state. The production system is started by the following command, and produces the results shown.

:(run) ; this starts things running
"John Brown" ; these are the results
"Anne Smith"

:(modify 11 ; now to tidy up the loose ends
  ^status satisfied) ; tidyup will now fire up.

:(modify 11 ; list math241 students
  ^status active
  ^object math241)

"Andrew Black" ; the results

:(modify 11 ; reset again
  ^status satisfied)
Summary

The OPS5 production system provides features for defining, creating, altering and removing elements and productions in the database. The user can define basic structures which can have instances by being duplicated, but there is no inheritance possible. The only types available are user-defined simple structures consisting of up to 126 symbolic atoms. There is no mechanism for attaching functions to attributes, or even to elements. Data is manipulated by productions, which consist of a condition part and an action part. The requirements for performing the action part are that the condition part is satisfied by the current state of the knowledge base and that all other satisfied productions have a lower priority.

The user defines productions and uses OPS5 primitives to process the information in the knowledge base. Production systems are generally oriented towards processing information with very little emphasis on how the knowledge is represented. This explains why the mechanisms for representing knowledge are not as powerful as those for knowledge representation formalisms such as FRL. An example of the differences is the methods of representing cpu utilisation in the two systems. In FRL a procedure is attached to the slot, via the $IF-NEEDED facet. In OPS5 the value must be constantly updated by declaring a number of productions which update the attribute value. These productions must fire up frequently enough to ensure that the value is "up to date", so that the state of the knowledge base is as accurate as possible.
PROLOG : Predicate Logic

Introduction

Prolog is a programming language based on Horn clause predicate logic [Kowalski 74]. The language was developed by A. Colmerauer's research group in Marseilles [Colmerauer et alia 73]. A Prolog program consists of a set of rules and a data base of assertions. The rules are used to interrogate and modify the data base. The University of New South Wales implementation is examined.

Description

The basic units in Prolog's representation of knowledge are objects and relationships between objects. These relationships are stated as "facts" about objects. An instance of a relationship, which is declared as a fact, is called an assertion. Rules can be built up by specifying a set of conditions required for a relation to be true. These conditions are relations which may also require further rules to be used. Evaluating conditions may produce side effects by using system functions such as assignment, input and output, and changing the data base. These changes can be further assertions of facts or can retract facts stated earlier.

The properties of an object are represented by the relationships the objects participate in. There are no restrictions on what type of object can participate in assertions. The default for an object is that the object does not participate in the relationship. This default can not be
altered. In this representation all objects are at the same hierarchical level. Features such as further specification and classes which can be instantiated must be incorporated into the model by the user as they are not provided by Prolog.

Both the rules and the assertions have the same declaration form, called clauses. A prolog program comprises a sequence of clauses. The clauses are implications of the form:

```
relationname (term1, term2, ... termN).
```

or

```
relationname (term1, ... termN) :- condition1, ... conditionK.
```

The first format asserts that the relation between the terms is true. The second format specifies that the relation is true if all the conditions are true, or in different terminology: to show the relation is true, first show that condition1 is true, then show that condition2 is true, and so on until all the conditions are true, or a condition is false. If any conditions are false, the condition is not shown to be true by that rule. However, another rule in the knowledge base may be used to try to show that the relation is true. A term is a constant, a variable or a function with a list of argument terms. A constant is like a LISP atom. A variable is used in pattern matching and is bound to any terms which would result in the relation matching an assertion or the relation part of rule in the data base.

The control structures in Prolog are recursion, unification pattern matching, and backtracking. Recursion results from a relation with one set of argument terms being a condition in a rule for the same relation with different argument terms. In these cases an assertion of the rule should also be given to provide a base for the recursion.

Unification pattern matching is used for binding variables with the terms of a relation. Unification generates the most general set of terms for which the condition is true. Where
there are multiple values, backtracking is used should one of the values result in the failure of another condition. Another value is bound to the variable until the query is successfully answered or all the values have been tried. Should the latter occur, this condition will itself fail, so that an earlier pattern match will be broken.

Backtracking can be prevented within a clause using the "cut" operator. This specifies that failures in the conditions after the cut may not result in backtracking of conditions evaluated before the cut. Should this be attempted, the clause fails and the system backtracks to the clause requiring the rule with the cut to be evaluated. There is also a method of generating multiple solutions. The condition fail causes the clause to fail at that point, no matter what pattern matches are tried. The fail condition should be preceded by a condition which outputs what would normally be a successful answer. The failure causes backtracking until another answer is printed, at which point the fail fails again. After all the answers are found, the initial goal will fail because all paths failed. The fail condition can also be used in conjunction with the cut operator to prevent backtracking within a small set of clauses that are known to be leading to dead ends.
Example

The following example defines the university model and demonstrates a simple enquiry. The system prompt is a colon. Text after the semicolon on each line is a comment.

:department(cosc). ; these objects are departments
:department(math).
:department(psyc).

:name(cosc,"Computer Science"). ; the name of the object cosc is
:name(math,"Mathematics"). ; "Computer Science"
:name(psyc,"Psychology").

:student(sl). ; these objects are students
:student(s2).
:student(s3).

:name(sl,"John Brown"). ; the names of the students
:name(s2,"Anne Smith").
:name(s3,"Andrew Black").

:courses(cosclll). ; these objects are courses
:courses(coscll2).
:courses(cosc203).
:courses(math241).

:dept(cosclll,cosc). ; the cosc department offers the
:dept(coscll2,cosc). ; course cosclll
:dept(cosc203,cosc).
:dept(math241,math).
:code(cosclll,111). ; the code for cosclll is 111
:code(coscll2,112).
:code(cosc203,203).
:code(math241,241).

:points(cosclll,6). ; cosclll is worth 6 points
:points(coscll2,6).
:points(cosc203,6).
:points(math241,4).

:desc(cosclll,"Computing 1"). ; the description of cosclll
:desc(coscll2,"Computer Science 1").
:desc(cosc203,"Information Processing Methods").
:desc(math241,"Abstract Algebra").

:prereq(cosc203,cosclll). ; the prerequisites for cosc203
:prereq(cosc203,coscll2).

:age(s1,18). ; age age of the students
:age(s2,19).
:age(s3,20).

:year(s1,1). ; the year at university of the students
:year(s2,1).
:year(s3,2).

:enroll(s1,cosclll). ; students enrolled in courses
:enroll(s1,coscll2).
:enroll(s2,cosclll).
:enroll(s3,cosc203).
:enroll(s3,math241).
:printenroll(C) :- enroll(S,C), name(S,N), write(N), nl, fail.
   ; this rule states how to print
   ; the names of the students
   ; enrolled in course C.

: :- printenroll(cosclll)
   ; this enquiry will result in the
   ; names of the cosclll students
   ; being displayed.

"John Brown"
"Anne Smith"
fail

: :- printenroll(math241)

"Andrew Black"
fail
Summary

The Prolog predicate logic language provides facilities to state and retract assertions and rules used to represent objects and relationships. Terms can be of three types (constants, variables, functions). There are no mechanisms provided to build structures to represent concepts; concepts are represented by relationships between objects. There is no inheritance or instantiation for objects, as there are no system defined functions for building hierarchies. Functions can only be attached to relations, and then only if the assertion is changed to a rule with the condition part being the desired attached function which should always evaluate to true.

The user asserts facts and rules which are used to answer queries. The data base is searched for assertions and rules which unify with the conditions of the enquiry. The rules may require further conditions to be evaluated. There are mechanisms for controlling program execution: the order of declaration of the rules and assertions, the cut operator to prevent backtracking, and the fail condition to force it.
CONCLUSION

With the PSN formalism it is possible to build semantic networks with programs attached to classes representing concepts. These programs are treated as classes, not as LISP functions. There are facilities for hierarchical organisation using IS-A and structure composition using PART-OF. There is a distinction between classes and instantiations. Constraints and defaults can be specified for slots, but there is no mechanism for attaching programs to slots. The alternative, making a relation between the classes of object and the class PROGRAM with the name the slot would have had, will be possible in EPSN. This would require a search function generator to be created for every reference to a value of the slot, which is now the result of the program instance in the assertion.

PSN distinguishes between structural properties and assertional properties, which are instances of relations. These assertions are the arcs in the network, and are restricted to a domain and a range for any one arc. Therefore it is not possible to represent a three-way relationship without using an intermediate object.

The programs are used to provide four basic operations (create, retrieve & remove instances; test for instanceship) and are invoked only when that particular operation is performed. There is no mechanism for creating demons of any sort; once started a program runs to completion before returning. There are no built-in facilities for concurrent processing of any sort, so that the four programs and any user-written functions are executed serially. Default programs are supplied for each class,
as are restrictions for each slot, the default here being T (LISP's true).

With the FRL formalism it is possible to build knowledge bases with concept representations similar to those of the PSN formalism. In FRL the user can attach functions to slots, while in PSN programs were attached at the class level. The functions in FRL are for working with the data in the slots, which means that the same effect as produced by programs in the PSN system can be simulated by attaching the functions to the INSTANCE slot of a frame, which will cause the appropriate function to be invoked whenever an instance of the class is referenced in some way. In this respect the FRL system is more powerful than the PSN formalism. Attaching functions at the slot level enables slots to hold information that is changing without requiring the value to be constantly updated by the user or the system. The $IF-NEEDED procedure can calculate the value when the slot is referenced. The calculations are only required when the data value is required, not at some frequency which is independent of the accesses to the data. An example of this is cpu utilisation, which is constantly fluctuating.

FRL treats relations between objects in a different manner than PSN. In PSN the relations and assertions were of a separate type, and were not built into the domain objects. In FRL there is no such distinction between classes and relations; relations are implied by the values in certain user-defined slots. It is these slots and the AKO slot that determine the network structure. The AKO link specifies the inheritance hierarchy. The entire frame is inherited, including default values, constraints on values, and any attached procedures. Using PSN terminology, the 'relation' slot values are indirect pointers to the 'range' of the relation, and the domain is the frame which contains this slot. The differences can be seen in the
mini-world example. The prerequisites for cosc203 are given as the value for the prereq slot, while in the PSN formalism the prerequisites were separate from the structure of the courses. In both systems it is difficult to have relationships between three or more objects, without requiring some separation of the parts of the relationship, such as turning the relation into an object with the relevant slots or relations defined. An alternative in FRL is to use the messages of a datum as indirect pointers to the various other objects participating in a relationship. However, this is not the purpose of comments.

In FRL there is no mechanism for creating demons. The functions which can be attached to the slots are evaluated when the slot is referenced, and there is no method of specifying that several expressions are to be evaluated concurrently.

The top level of FRL is the LISP top level. FRL provides some functions for saving frames and the knowledge base. It also provides commands for displaying frames and functions in a frame.

The simple university example for OPS5 demonstrates that there is no hierarchy mechanism, and no inheritance. The structures available are similar to the array structure in the programming language Pascal. The type of the array element values in OPS5 are symbolic atoms of LISP. There are no restriction mechanisms to force a subset such as strings or positive integers. However there are methods of specifying the format of these arrays, as can be seen in the example for student, department and courses, which provides a form of user type definitions.

The attribute values can be used to build a network structure. These consist of a set of elements containing references to other elements to build a set of nodes (elements) and arcs (the attributes and their values). This corresponds to the implementational level of semantic networks as described by R. Brachman [Brachman 79]. FRL would be considered to be at the
epistemological level of semantic networks, where the primitives are concepts, with inheritance and structural relations.

It is not possible to attach procedures to elements in the knowledge base, and to simulate it in some situations requires writing special productions for each situation where this effect is desired. For some of the procedures this is not even possible. OPS5 does provide functions to create, edit and remove elements. There is no way of retrieving data without using the productions. The standard LISP (MACLISP or FRANZ LISP) functions can be used in user-written subroutines, and some predefines OPS5 functions are also available.

The production system repeatedly evaluates the LHSs of the productions, selects a production with a satisfied LHS, and then performs the actions of the RHS. The evaluation and actions are straightforward. The conflict resolution strategy is the most complex mechanism of the production system. The production rules can be thought of as n-eyed demons, with one exception. Only one RHS will be performed at any time. When the actions are completed the LHSs are re-evaluated and the next RHS must be selected. Bad selection can lead to trivial loops, or ignoring the "newer" knowledge base elements. OPS5 has two strategies for conflict resolution which bias towards the more specific productions and those using the newer data.

The choice of strategies can be altered whenever the system is in the top level. Tracing and file functions are also provided at the top level, as are functions for displaying elements and productions.

Predicate logic systems such as Prolog are similar to production systems in that the information is stored in a global data base, and conditional statements act on the data base. In Prolog the information is represented by objects and relationships, called assertions. The conditional statements are
rules which require some conditions to be shown to be facts before determining whether the rule is applicable. If it is, the relation it defines is also a fact. Conflict resolution is not required in Prolog as it is in production systems because the search for applicable rules is sequential, stopping at the first rule or assertion which does not fail. Only if backtracking is required does the system continue searching for a new rule to use.

As for production systems, predicate logic systems place more emphasis on processing information than on how the knowledge is represented. In Prolog the declaration language is higher level than that of OPS5. The user does not have to specify where information should be stored, only the name of the relation, the name of the object and the other objects and values involved in the relationship. If the relationships are considered as slots, and the domain object and range objects are specified by name, then a collection of clauses can be viewed as a structure similar to an FRL frame. The conditions can be used as attached functions for when retrieving values, but not for creating, altering or removing concepts from the knowledge base.

Prolog would correspond to the logical level of semantic networks [Brachman 79], where the primitives are propositions, predicates and logical operators.

All the formalisms examined could represent the model of the university, although to varying degrees of success. Those which could not fully specify the restrictions of types for certain values also provided the easiest to use retrieval mechanisms to answer the enquiries.

Production systems are unique in the ability to provide demons which are constantly scanning the data base for certain requirements. This can be simulated to some extent by using the
pattern matching features of Prolog and the correct ordering of assertions and rules to force the best rule to be applied before trying any others. However the order is fixed, so special conditions are required which fail if there are better rules to be used at that point. The PSN system does not provide any methods to achieve this, as is the case with FRL.

The procedural semantic network formalism distinguishes between properties of objects and relationships between objects. There is no distinction in FRL or OPS5, and Prolog does not provide the same features for working with relationships.

Only FRL and PSN provide for full hierarchies and inheritance, further specification and instance of classes. This provides powerful features for representing knowledge, and thus in these languages it is possible to define the university model completely. This is not possible with the other two languages, especially the default and restriction of values.

Prolog provides the highest level language of the four examined. The method of representing objects and relations is left up to the system, with the user only specifying the assertion and rules. The other languages required descriptions of structures, slots and except for OPS5, retrieval required special searches.

As stated at the start of this report, the method of representing knowledge affects the possible processing of that knowledge. The overall power of FRL and similar languages (KRL, KL-ONE) when specifying the network structures, attaching procedures to slots, and inheritance capabilities indicate that there are differences in the formalisms in terms of what can be represented, how it can be processed, and how it advances research into how knowledge is represented.
REFERENCES


Kowalski R.A. "Predicate Logic as Programming Language", 49


50
APPENDIX A

"John Brown"  "Computing 1"  "Psychology"
  name  desc  code  name
  enroll  cosc111  dept
    enroll  cosc111
      points  dept
        111

"Anne Smith"
  name
  19-age-sl
  year
  enroll

"Computer Science 1"
  desc
  cosc
  dept
  112
  code
  points
  (cosc112)
  enroll
  year
  1

"Information Processing Methods"
  desc
  "Computer Science"
  cosc
  dept
  cosc203
  desc
  prerequisites
  points
  cosc
  prerequisites
  points
  prerequisite

"Andrew Black"
  name
  20-age-s3
  year
  enroll

"Abstract Algebra"
  desc
  "Mathematics"
  name
  enroll
  math241
  code
  points
  math241
  241
  4