Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

PROLOG AS A SIMULATION LANGUAGE

Heimo H. Adelsberger
Department of Computer Science
Texas A&M University, College Station, TX 7784

ABSTRACT

Prolog is a rather new language and is very different from traditional languages. Prolog is favored by the Japanese for their Fifth Generation Computer Systems. The acronym PROLOG is derived from PROgramming in LOGic and emphasizes the derivation of the language from predicate logic. Prolog can be considered as a general purpose very high level language, best suited for general symbol manipulation, intelligent and flexible database handling or problems, where some kind of search is required. Examples of application areas are computer aided design, database and "knowledge-base" management, natural language processing and rapid prototyping.

It is the purpose of this paper to demonstrate, how Prolog can be used as a tool in a simulation project.

The paper consists of two parts: a survey of the language Prolog and a description of T-Prolog, a Prolog based simulation language, using a process interaction approach.

INTRODUCTION

Most of the current work in simulation is done in general purpose high level languages like FORTRAN, BASIC or Pascal, or in simulation languages like GPSS, SIMSCRIPT or SLAM. The common characteristic of these languages is their imperative nature. One has to describe how a result is to be computed, or more precisely, how the computer should compute the desired result. This means that the programmer and analyst is forced to translate the real-world-problems into the "thinking scheme of the machine".

Prolog supports a radically opposite view of the problem: tell the computer what is to be done and what the facts and rules of the problem are. The name Prolog means PROgramming in LOGic and relates to the fact that at first, Prolog was a theorem prover. Prolog can be best described as a declarative language.

For simulation, the goal orientation of Prolog can help users to find an appropriate model. The typical search process which leads to a model is not well supported in ordinary simulation environments which are only able to run a given model. In the traditional approach most of the work in this area is done by humans: results of simulation runs are evaluated and in case of unacceptable results new input data and model parameters are provided for additional runs. This process is repeated until the desired results are reached, or otherwise the model must be redefined.

These tasks can be performed in Prolog automatically. A modeler declares his knowledge about the system, especially the description of the objects, the rules for modification of input data, model parameters and model characteristics. In such a manner the class of all possible and acceptable models is defined. It is the responsiblity of Prolog to find a model which fulfills the desired specification. (For more details, see /06/ and /08/.)

PROLOG

Prolog ('Programming in Logic') is a programming language based on symbolic logic, designed to represent and use facts about a field of knowledge. Facts are represented by a set of relations which describe the properties of objects and their interaction. A Prolog program consists of a set of rules describing these objects and relations.

Prolog is very different from traditional programming languages like FORTRAN, COBOL, BASIC, Pascal and even Ada. These languages are best described to be procedural or imperative: a program specifies explicitly the steps which must be performed to reach a solution of the problem under consideration.

Prolog can be viewed as a declarative or descriptive language. It is only necessary to describe the problem in terms of statements and rules affecting the objects in question. If the description of the problem is sufficiently precise, the problem can be solved.

The following list gives a rough idea of which kind of problems Prolog is best suited:

-- symbol manipulation
-- pattern matching
-- intelligent and flexible database handling
-- searching

Application Areas

Colmerauer's work on Prolog was designed to assist in natural language analysis and comprehension. Since then, Prolog has been chosen for applications of symbolic computation. Typical application areas are:

-- natural language processing
-- expert systems
-- database management
-- mathematical logic
-- symbolic equation solving
-- architectural design
-- software prototyping
-- artificial intelligence

History

Prolog is slightly over ten years old. Alain Colmerauer developed the first Prolog interpreter in Marseille in the early 70's. The work was based on the idea of logic programming, theoretically founded by R. Kowalski and P. Hayes. During the 70's the knowledge about Prolog was restricted to a rather small communi-

ty in academia. This situation changed with the announcement of the Japanese Fifth Generation Computer Project in 1981. Prolog (or its enhancement) was chosen as the key language of the project.

Basic Idea of Logic Programming

The problem specification is written in first-order logic and a theorem prover is used to provide a constructive proof of existence for an object meeting that specification.

```
A :- B , C , D.
```

This can be interpreted as a statement in logic, saying that A will be true if B, C and D are all true.

But it can also be interpreted as the definition of the procedure for checking the validity of A, stating that in order to execute A all procedures B, C and D should be executed.

Programming in Prolog

Computer programming in Prolog consists of:

-- declaring some facts about objects and their relationships
-- defining rules about objects and their relationships
-- asking questions about objects and their relationships

By now there exists a lot of different Prolog dialects. For the following the version given in Clocksin and Méllish's book "Programming in Prolog" (see /02/) is used.

Facts

For example, to say "Mozart composed Don Giovanni" merely affirms that a relation (composed) links two objects, designated by their names: Mozart and Don Giovanni. This could be written in Prolog in standard form:

```
composed( mozart, don-giovanni ).
```

The name of the relationship is given first, and the objects are separated by commas and are enclosed by parenthesis.

A collection of facts (and later, rules) is called a database.

Example: A database for operas

```
composed( beethoven, fidelio ).
composed( mozart,    don-giovanni ).
composed( verdi,     rigoletto ).
composed( verdi,     macbeth ).
composed( verdi,     falstaff ).
composed( rossini,   guglielmo-tell ).
composed( rossini,   il-barbiere-di-sévilla ).
composed( paesiello, il-barbiere-di-sevilla ).
```

Questions and Variables

It is possible to ask questions in Prolog. Two different types of questions can be asked: is-questions and which-questions. A typical is-question is: "Did Mozart compose Falstaff?". A typical which-question would be: "Who composed Falstaff?". In Prolog one would write:

```
?- composed( mozart, falstaff).
?- composed( X, falstaff).
```

For is-questions the answer is 'yes' or 'no'; in the above example it would be 'no'. For which-questions one has to specify one or more variables. 'X' is the variable in the above example, and the result would be

```
X = verdi.
```

If there are more solutions to a question as in "Which operas have been composed by Verdi?"

```
?- composed( verdi, X).
```

all solutions are listed:

```
X = rigoletto
X = macbeth
X = falstaff
```

When Prolog is asked a question containing a variable, Prolog searches through all its facts to find an object that the variable could stand for.

Syntax

Prolog programs consist of terms. A term is a constant, a variable or a compound term (structure). Constants are numbers or atoms. Names of atoms begin with a lower-case letter. Variables are always capitalized. A structure is written by specifying its functor ('composed' in the example above), followed by its components (also called arguments) enclosed in parenthesis, separated by commas. Lists are a special form of compound terms.

Conjunctions

Given the following database:

```
likes( mary, food).
likes( mary, wine).
likes( john, wine).
likes( john, mary).
```

In Prolog one could ask "Is there anything that John and Mary both like?" in the following form:

```
?- likes( mary, X) , likes( john, X).
```

The comma is pronounced 'and', and expresses the fact that one is interested in the conjunction of these two goals.

Rules

A rule is a general statement about objects and their relationships. Rules are used to say that a fact depends on a group of other facts. For example, to explain that a person is someones sister one could say:

```
'X is a sister of Y if
    X is female and
    X and Y have the same parents.'
```

In Prolog syntax one would write:

```
sister-of(X,Y) :-
   female(X),
   parents(X, Z1, Z2),
   parents(Y, Z1, Z2).
```

The symbol ':-' is pronounced 'if'.

Lists

A list is an ordered sequence of elements that can have any length. Lists are written in Prolog using square brackets, elements are separated by commas as in:

```
languages( [ gpss, simscript, simula, slam ] ).
```

Some other lists:

```
[]
[ [ the [ boy ] ] [ kicked [ the [ ball ] ] ] ]
```

The first list is the empty list, the second one represent the grammatical structure of the sentence 'the boy kicked the ball'.

The vertical bar is used to split a list into its head and tail:

```
?- languages( [X|Y] ).
   X = gpss
   Y = [ simscript, simula, slam ]
```

Recursion

Recursion is a powerful technique to express complex algorithms and structures in an easy way. In many cases algorithms can be expressed in two different forms: one using recursion and one using loops. A simple and good example is the computation of the factorial function. In Prolog, recursion is the normal and natural way.

The membership test for an element of a list is a simple and good demonstration of recursion in Prolog:

```
member(X, [X|Y]).
member(X, [H|Y]) :- member(X,Y).
```

This can be read as:

'The element given as the first argument is a member of the list given as the second argument, if the list starts with the element (the fact in the first line) or if the element is member of the tail (the rule in the second line).'

Possible question:

```
?- member(d, [a,b,c,d]).
yes
?- member(e, [a,b,c,d]).
no
```

It is possible to get all members of a list by asking:

```
?- member(X, [a,b,c,d]).

X = a
X = b
X = c
X = d
```

Example: Permutations

The following Prolog program can be used to compute all permutations of a given list:

```
perm( X, [H|T])  :-
   append( Y, [H|Z], X ),
   append( Y, Z, P),
   perm( P, T ).
perm( [], [] ).

append( [], X, X ).
append( [A|B], C, [A|D] ) :-
   append( B, C, D ).

?- perm( [a,b,c], X ).
X = [a,b,c]
X = [a,c,b]
X = [b,a,c]
X = [b,c,a]
X = [c,a,b]
X = [c,b,a]
```

Explanation: The predicate 'append' defines that the list given as the third argument is the joint of the list given as the first and second argument. For example it is true that:

```
append( [a,b,c], [i,j], [a,b,c,i,j] ).
```

With the first two arguments instantiated and a variable as the third argument, 'append' can be used to join two lists:

```
?- append( [1,2,3], [4,5,6], X).
X = [1,2,3,4,5,6]
```

But 'append' can also be used to split a list into all possible sublists:

```
?- append( X, Y, [1,2,3].
X = []          Y = [1,2,3]
X = [1]         Y = [2,3]
X = [1,2]       Y = [3]
X = [1,2,3]     Y = []
```

With the help of 'append' the predicate 'perm' is easy to understand:

The list X is split into two sublists Y and [H|Z]. Y and Z together form the list P which is permuted. A permutation of P is called T. The element H from the first split process together with each permutation T yields a result.

Glossary

The above Prolog 'program' to compute permutations consists of two 'procedures' ("perm" and "append"). Each procedure comprises one or more 'clauses' (2 for "perm" and 2 for "append"). A clause is terminated by a period. The procedure name is called a 'predicate'. The number of 'arguments' is called the 'arity' (2 for "perm " and 3 for "append"). Each clause has a 'head' which is also called 'procedure entry point' and may have a 'body'. A body is separated from the head by the symbol ':-'. The head defines the form of the predicates arguments. The body of a clause consists of 'goals' or 'procedure calls' which impose conditions for the head to be true. Clauses without bodies are called 'facts', clauses with bodies are called 'rules'. Prolog objects are called 'terms'. Terms are either 'variables', 'atoms' or 'compound terms'. Compound terms comprises a 'functor' (like "append") and its arguments. An atom can be considered as a functor of arity 0.

T-PROLOG

T-Prolog has been developed by I. Futo and J. Szeredi. T-Prolog can be considered as a goal oriented discrete simulation language. Compared to conventional simulation languages it can be called a very high level simulation language. The main characteristics of T-Prolog are (see /08/):

-- A process interaction view of simulation is supported.
-- A built-in backtrack mechanism permits backtracking in time in case of a deadlock or a hopeless intermediate situation arising during program execution.
-- The system can change the structure of the original simulation model automatically on the basis of logical consequences derived from sophisticated preconditions.
-- Advanced process communication mechanisms are supplied for the user.

T-Prolog is based on the M-Prolog system, the Prolog implementation developed at the Hungarian Institute for Co-ordination of Computer Techniques.

The basic ideas of T-Prolog can be best described by a small example (see /08/):

Example

Jim and Dick want to rob the Prolog-Savings-Bank. Jim climbs into the bank (which takes him 5 minutes) whereas Dick waits outside. There are different safes in the bank for each of which appropriate tools are needed. Jim choses a safe, and if they have the tools they rob the bank. The robbery has to be finished in 25 minutes. Details can be found in the following Prolog program. The question is, which safe shall be chosen for a succesful robbery.

```
(1)  jim-gets-the-money(Bank,Safe) :-
       jim-climbs-into(Bank),
       chooses(Safe),
       wait-for(Tools),
       opens(Safe,Tools),
       outputs(Safe,Bank).

(2)  dick-gets-the-money(Bank,Safe) :-
       wait(nonvar(Safe)),
       has(Tools,Safe),
       send(Tools).

(3)  chooses(wertheim).
(4)  chooses(milner).
(5)  chooses(chatwood).

(6)  has(tool-set-a,milner).
(7)  has(tool-set-b,chatwood).

(8)  jim-climbs-into(Bank) :- during(5).

(9)  opens(milner,Tools) :- during(40).
(10) opens(chatwood,Tools) :- during(10).

(11) outputs(Safe,Bank) :-
       systemtime(T),
       outstring("The bandits got "),
       outstring("the money from the "),
       output(Safe), outstring(" safe, "),
       outstring(" from the"),
       output(Bank), outstring(" bank "),
       outstring(" at time "),
       output(T).

(12) fin.

(13) problem :-
       new(dick-gets-the-money(prolog-savings,Safe),
          dick,0,25),
       new(jim-gets-the-money(prolog-savings,Safe),
          jim,0,25).
```

Explanation:

Two processes are created to solve the problem (13) via predicate 'new': The first argument of 'new' is the head of the process, the second one the name of the process. The third argument is the time when the process has to start and the forth argument is the maximum duration for the process.

Synchronization is achieved via 'during' (8,9,10), 'wait' (2) and the message passing predicates 'wait-for' and 'send' (1,2). During suspends the execution for T time units, wait until the argument of wait can be successfully completed. Processes executing a 'wait-for' goal are suspended until another process executes a 'send'.

The current internal time can be used via the predicate 'systemtime' (11).

The result of the simulation run would be:

"The bandits got the money from the chatwood safe, from the prolog-savings bank at time 15".

SUMMARY

The basic concepts of Prolog make it easy to extend the language for a specific type of application, as demonstrated for simulation by the example of T-Prolog. In addition, the goal orientation of Prolog can support the search process to find an appropriate model. The fact that Prolog is normally implemented as an interpreter has the known advantages and disadvantages: interpreters speed up code development but slow down execution, which, of course, is crucial for simulation. One way to circumvent slow execution is to regard a model developed in such a manner as a prototype, which later can be transformed automatically or by hand into a suitable language. Although not invented originally for this purpose, rapid prototyping with Prolog seems to be in general a very promising field.

REFERENCES

/01/ Clark K.L., McCabe F.G., micro-PROLOG: Programming in Logic, Prentice/Hall International, Englewood Cliffs, New Jersy, 1983.
/02/ Clocksin W.F., Mellish C.S., Programming in Prolog, Springer-Verlag Berlin Heidelberg New York, 1981.
/03/ Coelho H., Cotta J.C., Pereira L.M., How to solve it with PROLOG, Ministerio da Habitacao e Obras Publicas - Laboratorio Nacional de Engenharia Civil, 2nd edition, Lisboa, 1980.
/04/ Colmerauer A., Kanoui H., Caneghem M. van, Prolog, theoretical principles and current trends, Technology and Science of Informatics, vol. 2, no. 4 (1983), pp. 255-292.
/05/ Domolki B, Szeredi P.: PROLOG in Practice, Information Processing 83, R.E.A. Mason (ed), Elsevier Science Publishers B.V. (North-Holland), IFIP, 1983.
/06/ Futo I., Szeredi J.: T-PROLOG User Manual Version 4.2, Inst. for Coord. of Comp. Techn., Budapest, Hungary, 1983.
/07/ Kowalski R., Logic for Problem Solving, North Holland, New York, Oxford, 1979.
/08/ SZKI, T-PROLOG A Very High Level Simulation System, Budapest, 1982.