A modelling tool based on mathematical logic T-PROLOG

By I. FUTÓ*, J. SZEREDI* and K. SZENES**

1. Introduction

T-PROLOG is a very high level language and a simulation language at the same time. It is a tool for simulation and modelling purposes equipped with the advantageous facilities of the very high level, logic based languages used in the field of AI. As we shall show, the marriage of two different principles — simulation and logic based problem solving results in a simulation technique new features. We amplified the logic based problem solving with a new — hitherto not considered — facility: the manipulation of time-dependent problems.

The structure of the simulation models written in T-PROLOG can be made dynamic — that is it can be suitably changed during run time — as the run time modification of the model description by means of addition and deletion of assertions/reductors to the program representing the model is allowed.

The language is planned to solve problems needing the cooperation of relatively few but frequently communicating processes. The processes pass and resume control dynamically, the control transfer is determined at run time, it is not determined at the time of writing the program (e.g. SIMULA [6]). The processes are controlled by sophisticated conditions. These are not simple and/or relations but the results of logical deductions of more than one steps.

The compiler of T-PROLOG is written in PROLOG and generates PROLOG programs that always contain a scheduler which modifies appropriately the strategy of the PROLOG interpreter.

2. Preliminaries

In this section an informal introduction to PROLOG is given. The reader familiar with this language may skip it.

2.1 Program elements. A PROLOG program consists of a set of Horn clauses. Horn clauses may be introduced as a follows: Every formula of a first order predicate calculi can be written as a conjunction of clauses

$$C_1 \wedge C_2 \wedge \ldots \wedge C_H .$$

A clause is the disjunction of literals

$$C_i = L_1 \vee L_2 \vee \ldots \vee L_k.$$

A literal is an atomic formula (positive literal) or a negation of an atomic formula (negative literal). An atomic formula is an expression of the form

$$P(t_1, \ldots, t_n),$$

where P is as n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. A term is a variable symbol, a constant symbol, or an expression of the form

$$f(t_1, \ldots, t_r),$$

where f is an r-ary function symbol and $t_1, \ldots, t_r$ are terms. A clause having at most one positive literal is named Horn clause. The usual notation for Horn clauses is

$$B \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_k,$$

where both sides of the arrow can be missing.

The various Horn clauses and their notation in T-PROLOG (j stands for the number of the positive literals and k stands for the number of the negative literals)

a) $j=0, k=0$	$\square$	the empty clause,
b) $j=1, k=0$	$B.$	the assertion,
c) $j=0, k>0$	$: A_1, A_2, \ldots, A_k.$	goalsequence,
d) $j=1, k>0$	$B: A_1, A_2, \ldots, A_k.$	rule of inference or reductor.

The assertion in T-PROLOG is expressed by a literal terminated by a point. The assertion means the declaration of a simple fact.

E.g. (1) Beautiful_is (Mary).

The reductor is expressed by a literal sequence in the following way, literal_1: $\text{literal}_2, \ldots, \text{literal}_n$. The reductor serves for the declaration of the "preconditioned" fact, literal_2 and ... and literal_n are preconditions of literal_1.

E.g. (2) Will_marry(x): Clever_is(x).
(3) Will_marry(x): Beautiful_is(x).

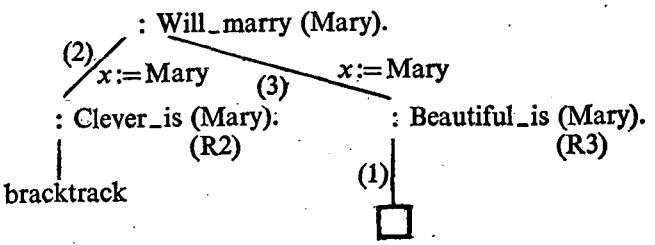
The meaning of (2) is, that every clever person will marry, while the meaning of (3) is the same, with "beautiful" instead of "clever". PROLOG goal statement is expressed by a literal or a sequence of literals preceeded by a: sign and terminated by a point.

E.g. (4): Will_marry(Mary).
(4)′: Will_be_happy(Ann), Will_marry(Ann), Is_beautiful(Ann).

The goal statement expresses a simple question (4) or a complex one — (4)′ — to be answered by the system. (Will marry Mary? or Will be happy Ann and Will marry Ann and Is beautiful Ann?).

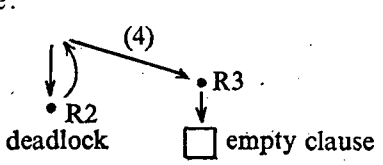
To illustrate the execution of a PROLOG program we show the diagram of the above simple example. PROLOG uses the LUSH resolution for deriving the

goal from the given set of assertions and reductors. The diagram of the program execution is the following:



2.2. The search tree of the deduction. The program execution of 2.1 can be represented by a tree, called search tree. The root of the search tree symbolizes (or is labeled by) the goal statement. When PROLOG tries to match the goal, clause (4), to the assertions/reductors it founds clauses (2) and (3) as being appropriate ones. Having executed the matching got (R2) and (R3), the remainder clauses of (2) and (3), respectively. These remainders we got leaving the first, matching literal from the clauses.

The nodes next to the root of the search tree symbolize (or are labeled by) the remainder of the clauses matching to the root. Proceeding in the proof the theorem prover founds no clauses matching to (R2) so it got into a deadlock. But for (R3) it founds the matching (1). The remainder clause of (R3) will be the next node of the search tree, in this case it is the empty clause. The nodes following those ones next to the root are labeled always by the respective remainder clauses. PROLOG traverses the search tree in a depth-first way. In the case of the example 2.1 this tree is very simple:



3. Running the processes of the simulation model

As one of the fundamental notions of simulation field is process, so we need to incorporate it in our language. Therefore our notion of process has to fulfil the following requirements:

a) The process notion has to cover the same concept people think of reading articles about simulation, co-routine programming etc.

b) The process notion has to originate somehow from the way of deduction used in T-PROLOG. This is necessary because the instructions supporting time considerations exploit the facilities resulting from the derivation method.

In section 2.2 we introduced the search tree of a PROLOG goal statement, that is the search tree of a T-PROLOG simple goal statement. Solving the problem given by a simple goal statement, the T-PROLOG system traverses the search tree determined by the simple goal and the given assertions/reductors. This traversing

is terminated when the empty clause appears on one of the branches of the tree. We define the notion of the process as this search tree traversing process.

The simulation model is described by the T-PROLOG program. The model is put into motion when the system starts the processes. The running of the simulation model means the logical deduction process of proving the compound goal statement. This is just the process notion we need for the explanation of the meaning and the effect of the T-PROLOG instructions, and these explanations will fit into the "simulational" way of thinking and also into the logical deduction context. We shall say that the processes themselves traverse the search trees, — we do so because we shall have to explain the steps of the program execution which are actually jumps from one node of the tree to an appropriate other one. (In case of the compound goal this other node won't necessarily belong to the same tree as the starting node of the "jump" does.

In a real simulation model we have obviously more than one process. For the sake of the creation of these processes compound goals are used. To the compound goal of the simulation model description, which is the T-PROLOG program itself, we attach the set of processes corresponding the simple goals comprising this compound one. At the same time to the compound goal we attach the search space of the proof of the compound goal, that is the set of the search trees corresponding to the simple goals.

The run of the processes is synchronised by a built-in scheduler. The user can change in a certain extent the synchronisation strategy of the scheduler. (See at the description of T-PROLOG instructions.) Besides the synchronisation, on a one-processor computer the scheduler helps the system to run the processes quasi-simultaneously. The processes traverse the search space but every process keeps to its own search tree. When a process passes the control to an other one it also means the respective changing of search tree. This means that the proof "goes" breadth-way too, (while going top-down, depth-first on one tree), the order of the choice of the trees depends on the order of the processes in the lists of the scheduler.

The processes communicate in three ways:

a) through the common logical variables evaluated by pattern matching,

b) through the statements expressing sending and receiving messages (so called demon mechanism),

c) by the modification of the simulation model description (they can add/delete clauses to the program and they can even create new processes and delete existing ones).

4. Scheduling the activity of the processes

4.1. Simulation system. Simulation models cannot be described in a system lacking the facilities of handling the simulation system time (in short system time).

In T-PROLOG the preconditions prescribing to a process wether it may be proceeded to the next execution step or not, can be either dependent or independent on time. This means that to every step of a process a time interval can be given in order to determine the duration of the step. If the execution of the step "doesn't take time" then this duration is considered to be zero, otherwise it is given in an integer number of time unit.

Now the meaning of system time is the following: By definition its value is zero at the start of the execution of the T-PROLOG program. If — at the current moment — the value of system time is t_0 and a process having currently the control encounters a precondition referring to time duration of value t_d and the process is able to fulfil the precondition then it proceeds to its next instruction at system time t_0+t_d. (We have only two instructions expressing the fact that a statement is true depending on a time condition. The above explanation of system time however, could be made clearer thinking only of the explicit time condition. The meaning of all the operations on system time will be explained at their description).

4.2. The scheduler. A scheduler is needed not only because only one process can be run at a time and the state of the others has to be recorded but also for system time maintenance, for synchronisation of the — of course — communicating processes and as a tool of realisation of the desired effect of the simulation instructions.

The scheduler maintains two lists — the list of the waiting, and the list of the blocked processes. The elements of the first one are the processes able to run at the current time, and the processes having start time (it can be given in the goal description) or reactivation time (a process can meet a condition prescribing to suspend — hold — its activity for a given time interval) greater than the current system time. Processes of the waiting list are ordered on the bases of their activation or reactivation time. The elements of the second list are the processes which had become blocked because of some — as yet unfulfilled — precondition instructing the process, e.g. to wait for a message from an other one, or for the fulfilment of a condition.

The scheduler works in the following way:

a) Initialisation of the program execution.

The scheduler sets the system time to the minimum of the start times given in the description of the simple goals comprising the compound goal statement of the program. To every process denoted by the simple goal statements the start time — when the process has to begin its activity — and the end time — when the process has to finish its activity — can be given. (See sect. 6.)

b) Starting the program execution.

All of the processes enlist, in the order of their start time, to the waiting list.

c) Continuation of the program execution.

If the waiting list and the blocked list is empty, it means that the deduction was successful and the program terminates.

If there is a process in the waiting list with start time or reactivation time equal to system time then the scheduler starts the first one. If this gets blocked, or has to change its position in the waiting list (having encountered a precondition of suspending effect) then the scheduler starts the next process with appropriate start or reactivation time from the waiting list, if there is any such one at all.

If there is no such a process in the waiting list, or the waiting list has become exhausted, then the scheduler turns to investigate the blocked list. If the reason for staying there dissolves for any of the processes then the scheduler removes all such processes from the blocked list, puts them into the beginning of the waiting list and returns to the start of step 3. If there is no such a process in the blocked list then the scheduler goes to step 4.

d) Incrementing the system time.

If there is a process in the waiting list whose activation time is less than or equal to its end time (the point of time when the proof of the simple goal corresponding to this process has to be finished) then the scheduler sets system time to this activation time and returns to step 3. Otherwise backtracking begins. If the waiting list is empty but the blocked list isn't, then backtracking in time begins too. (In both cases the system goes back to the last previous decision point and choses the "next" alternative. If there is no next alternative even at the first decision point, then the problem cannot be solved under the conditions given in the program.) If both of the lists are empty then we are ready, all the processes have completed their proof.

5. A simple example

The example given here is a complete T-PROLOG program. The problem to be solved: Paul and Annie want to go to a movie at six o'clock. It is half past five now. Paul can go to a movie to see a film, if there are tickets available to a film at a movie, that film is acceptable for Annie and then travels to the movie. Annie can go to a movie to see the same film as Paul, if Paul can buy tickets to the film, the film is a good one and it is acceptable for her and then she travels to the movie. There are films which are good and Paul and Annie have to travel during an interval of time of nonzero length to get to the movie.

The corresponding T-PROLOG program is:

(1) Can_go_to(Paul, movie, film):
There_is_ticket(movie, film),
Wait_for(Acceptable_for(Annie, film)),
Travels(Paul, movie).

(2) Can_go_to(Annie, movie, film):
Wait(Paul_can_buy_a_ticket(film)),
Good_film(film),
Send(Acceptable_for(Annie, film)),
Travels(Annie, movie).

(3) Paul_can_buy_a_ticket(film):
Not(*Variable*(film)).

(1)—(3) are rules of interference or reductors.

(4) There_is_a_ticket(Rex, Hair).
(5) There_is_a_ticket(Athena, Star_wars).
(6) Good film (Hair).
(7) Good film(Star_wars).

(4)—(7) are time independent assertions.

(8) Travels(Paul, Rex): *During*(45).
(9) Travels(Paul, Athena): *During*(25).
(10) Travels(Annie, Rex): *During*(20).
(11) Travels(Annie, Athena): *During*(20).

(8)—(11) are time dependent assertions.

(12): *New*(Can_go_to(Paul, movie, film). Nil, Paul)*End* 30,
New(Can_go_to(Annie, movie, film). Nil, Annie)*End* 30.

(12) is a compound goal with goals to be achieved in 30 time units. (That is from 17,30 to 18,00 in our case).

The detailed explanation of the program execution is in Sec. 7.

6. T-PROLOG instructions

We shall not deal with time independent assertions and reductors, or built-in procedures, because they are used the same way as in PROLOG [1, 7].

6.1. Compound goal statement. The compound goal statement is a sequence of simple goals separated by "," and terminated by a ".".

6.2. Simple goal statement. In T-PROLOG, as it was said, there is a corresponding process to every goal. The syntax of the simple goal statement is the following:

:*New*(goalsequence, identifier) {*Start* T_1}{*End* T_2},

where "goalsequence" is a sequence of literals separated by a ".", terminated by "Nil". "Identifier" serves as identifier of the goal and of the process corresponding to it. T_1, T_2 are points of time. The proof of the simple "goal" has to start at T_1 and has to be finished by T_2. Any or both of the two items enclosed in braces can be missing and in this case the default value of the start time and/or end time is assumed to be zero and 100 000 respectively. (4) and (4)′ of 2.1 can be given in T-PROLOG

:*New*(Will_marry(Mary). Nil,1). and
:*New*(Will_be_happy(Ann). Will_marry(Ann).
Is_beautiful(Ann). Nil,1).

6.3. Built-in procedures for synchronizing the events.

a) Time.

Hold(t)

If a process encounters this precondition in its search tree then stops the traversing for the time interval of length of t time units. The position of the process in the waiting list will be changed according to the value of the given time interval (see scheduler).

b) Logical condition.

Wait(condition), where "condition" is a literal.

If a process encounters this precondition in its search tree then if the "condition" can be proved as a simple PROLOG goal, then the proof takes place and the process continues its traversing. Otherwise, when the "condition" cannot be

proved, then the process gets blocked and will be reactivated only when the "condition" is already true. The unprovability of "condition" may occur, because of a statement is missing or if a variable has not got the required value.

Wait(condition, identifier)

Where "condition" is a literal and "identifier" is a process name or a variable.

The user has the possibility to force the T-PROLOG scheduler to pass the control to the process named "identifier" if the "condition" cannot be proved, otherwise it has the same effect as *wait*( ). If "identifier" is a variable then the backtrack strategy is changed. In the previous cases if during backtrack the control reached this node again the backtracking is continued. In this case the next process from the waiting list gets the control, and during further backtracks in the worst case all the processes of the waiting list get the control. This built-in predicate was necessitated because considering certain unit clauses as resources seezed by the processes. The order of the seezing may influence the final solution of the problem.

6.4. Built-in procedures for explicite communication.

Wait_for(message), *Send*(message)

where "message" is a term.

If a process encounters precondition *Wait_for*( ) it will be blocked, and will be reactivated only when another process reaches a *Send*( ) precondition with matching "message" value. At the same time, sending process becomes suspended, the processes waiting for this message will be inserted before the first element of the waiting list and all processes inserted will be reactivated and continue execution till a blocking point, before the sending process, which preserved its relative position in the waiting list, would be active again.

6.5. Time dependent assertions.

a) For expressing the fact that a statement is true at a given time, till a given time, after a given time, before a given time, during a time interval, the suffixes $At(T)$, $Till(T)$, $After(T)$, $Before(T)$, $From(T_1, To(T_2))$ are used.

At the moment of the match of an assertion containing one of the above mentioned suffixes these conditions are evaluated, if their value is false, the T-PROLOG begins to backtrack.

b) For changing the position in the waiting list, according to an explicite time condition, the suffix $During(T_d)$ is used.

If a process encounters an assertion suffixed with this suffix, then it passes the control to the next element of the waiting list and enlists to the waiting list to the position corresponding to system time $+T_d$.

6.6. Procedure for proving a sequence of conditions in a "really" parallel way.

Simultaneous(list of conditions)

If a process encounters this precondition — or rather — command, then it becomes blocked. The compiler assumes the list of conditions to be a "new compound goal" and generates a process to every "simple goal" of this compound goal. These processes start to execute in turn and continue to do so until all of their corresponding goals will have been proved one-by-one. However, from the point of view of the model these processes seem to execute parallely, because the reactivation time of the formerly blocked process is set to the completion time of these proofs which is the value of the system time at the blocking plus the maximum of time intervals necessary for the individual proofs.

6.7. Facilities for modifying the search strategy and the scheduler.

a) *Delete*(identifier)

If a process encounters this precondition then continues executing but the scheduler deletes the process with identifier "identifier" from its recording and the T-PROLOG compiler deletes the corresponding simple goal from the compound goal.

b) *Systemstate*($w1$, $b1$, p)

If a process encounters this precondition then the T-PROLOG compiler gives the waiting list of the scheduler in $w1$, the blocked list in $b1$, and the identifier of the running process in p.

7. Execution of the program of section 5

Initial state:

Waiting processes: 0:Paul:30. 0:Annie:30.Nil (start time:process id:end time)
Blocked processes: Nil
Active process: Paul
System time: 0

```
    (1)  | :Can_go_to(Paul, movie, film).
[A]      | :There_is_ticket(movie, film),
         |     Wait_for(Acceptable_for(Annie, film)),
         |         Travels(Paul, movie).
    (4)  |     movie:=Rex film:=Hair
         | :Wait_for(Acceptable_for(Annie, Hair)),
         |         Travels(Paul, Rex).
```

The process Paul is suspended waiting for a message.

Waiting processes: 0:Annie:30. Nil
Blocked processes: Paul. Nil
Active process: Annie
System time: 0

```
(2) | :Can_go_to(Annie, Rex, Hair).
    | :Wait (Paul_can_buy_a_ticket(Hair)),
    |     Good_film(Hair),
    |       Send(Acceptable_for(Annie, Hair)),
    |       Travels(Annie, Rex).
    :   (3) | :Paul_can_buy_a_ticket(Hair).
    :       |   film:=Hair
    :       □ :Not(Variable(Hair)).
(6) | :Good_film(Hair),
    |     Send(Acceptable_for(Annie, Hair)),
    |         Travels(Annie, Rex).
    | :Send(Acceptable_for(Annie, Hair)),
    |     Travels(Annie, Rex).
```

Process Paul waiting for the message is reactivated.

Waiting processes: 0:Paul:30. 0:Annie:30. Nil
Blocked processes: Nil
Active process: Paul
System time: 0

```
(8) | :Travels(Paul, Rex).
```

Backtrack

A backtrack occurs because of a time problem. Paul has to get to the movie at 30 (six o'clock) and now he needs for traveling 45 time units (8), thus actual system time $+45=0+45>30$, so Paul's goal can't be achieved this way.

There is another alternative but only at state [A].

```
    | :There_is_ticket(movie, film),
    |     Wait_for(Acceptable_for(Annie, film)),
(5) |         Travels(Paul, movie).
    |     movie:=Athena film:=Star_wars
    | :Wait_for(Acceptable_for(Annie, Star_wars)),
    |         Travels(Paul, Athena).
```

Process Paul is suspended.

Waiting processes: 0:Annie:30. Nil
Blocked processes: Paul. Nil
Active process: Annie
System time: 0

```
(2) | :Can_go_to(Annie, Athena, Star_wars).
    | :Wait(Paul_can_buy_a_ticket(Star wars)),
    |     Good_film(Star_wars),
    |         Send(Acceptable_for(Annie, Star_wars)),
    |             Travels(Annie, Athena).
    |         (3) | : Paul_can_buy_a_ticket(Star_wars).
    |             |     film:=Star_wars
    |             □ : Not(Variable(Star_wars)).
    | :Good_film(Star_wars),
    |         Send(Acceptable_for(Annie, Star_wars)),
    |             Travels(Annie, Athena).
    | :Send(Acceptable_for(Annie, Star_wars)),
    |         Travels(Annie, Athena).
```

Process Paul is reactivated.

Waiting processes: 0:Paul:30. 0:Annie:30. Nil
Blocked processes: Nil
Active process: Paul
System time: 0

```
(9) | : Travels(Paul, Athena).
```

Processing of Paul is abandoned for 25 time units. (He "travels".)

Waiting processes: 0: Annie: 30.25:Paul:30. Nil
Blocked processes: Nil
Active process: Annie
System time: 0

```
(10) | : Travels(Annie, Athena).
```

Processing of Annie is abandoned for 20 time units. (She "travels".)

Waiting processes: 20:Annie:30.25:Paul:30. Nil
Blocked processes: Nil
Active process: Annie
System time: 20

```
|
□
```

The goal of process Annie is achieved in 20 time units (17^{50}), she arrived at the movie Athena to see the film Star_wars.

Waiting processes: 25:Paul: 30. Nil
Blocked processes: Nil
Active process: Paul
System time: 25

```
|
□
```

The goal of process Paul is achieved in 25 time units (17^{55}), he arrived at the movie Athena to see the film Star_wars.

Waiting processes: Nil
Blocked processes: Nil
Active process: –
System time: 25

The problem is solved.

8. Conclusion

As it was shown in the preceeding sections we provide simulation technique more advanced than the previous methods from the following aspects:

a) the system takes over a part of the problem solving effort from the user who has to concentrate rather on defining the task then on solving it;

b) the system changes automatically and dynamically the model on the basis of logical consequences derived from sophisticated preconditions;

c) a built-in backtrack mechanism permits *backtracking in time* in case of a deadlock or hopeless intermediate situation (logical condition necessary for the continuation of the execution is missing or the current time conditions have become contradictory);

d) T-PROLOG provides for a process communication mechanism that is sophisticated enough owing the fact that the processes communicate through variables evaluated by pattern matching or by modification of the model description (the traditional way of communication is preserved, too: the processes are able to send /receive messages).

All the advantages enumerated above are due to this hitherto unusual approach: building the simulation method on a language founded on the principles of mathematical logic. As far as the traditional simulation facilities of T-PROLOG are concerned:

a) a dynamic simulation approach has been choosen: the processes pass the control to each other dynamically, according to the requirements of the situation and not on the basis of a rigid, preprogrammed resume — detach technique as e.g. SIMULA 67 [6],

b) transaction and event oriented simulation approach are allowed.

Abstract

A logic based simulation language is presented. The language is an extension of PROLOG towards time and process manipulation.

Comparing it to the traditional simulation languages the language has the following advantages due to its logical basis:

— it changes automatically and dynamically the simulation model on the basis of logical consequences derived from sophisticated preconditions,
— the system takes over part of the problem solving effort from the user,
— a built in backtrack mechanism permits backtracking in time in case of a deadlock or the occurence of a hopeless intermediate situation during program execution,
— a more advanced process communication mechanism is presented for the user.

The processes are synchronized by a built-in scheduler. Its strategy can be modified by the user. The realisation of the scheduler, the effect of the simulation instructions and the way of using the logical deduction ensures that:

— the interaction of the processes is dynamic, the processes pass and resume control dynamically, and

— either the event-, or the transaction oriented way of simulation programming is allowed.

* INSTITUTE FOR COORDINATION
OF COMPUTER TECHNIQUES
AKADÉMIA U. 17.
BUDAPEST, HUNGARY
H—1052

** CHEMICAL WORKS
OF RICHTER GEDEON
BUDAPEST, HUANGRY
H—1475

Bibliography

[1] FUTO, I., F. DARVAS, P. SZEREDI, Application of PROLOG to implement QA and DBM systems, *Logic and data bases*, Ed. H. Gallaire, J. Minker, Plenum Press New York, 1978, pp. 347—376.

[2] FUTO, I., J. SZEREDI, PAPLAN a very high level programming language for parallel problem solving, Preprints of Int. Conf. on Robot Control, Bratislava, 30.06—04.07, 1980, pp. 74—76.

[3] GPSS/360 User's Manual, IBM 420—03261.

[4] KIVIAT, P. J., *Simulation Using SIMSCRIPT II.*, Rand Corporation, 1968.

[5] KOWALSKI, R., Predicate logic as a programming language, DCL Memo no. 70, Edinburgh University, Edinburg, 1974.

[6] SIMULA Reference Manual, Control Data 6400/6500/6600 Computer Systems.

[7] PEREIRA, F., L. PEREIRA, D. WARREN, User's guide to DECsystem—10 PROLOG, Dept. of AI Univ. of Edinburgh, 1978.

(Received Dec. 1, 1980)