Technical University
of Cluj–Napoca

Automation Computers Applied Mathematics

Scientific Journal

Vol. 4, No. 2 – 1995
ISSN 1221 – 437X

A Non-Procedural Language

Eneia Todoran
Department of Computer Science, Technical University of Cluj-Napoca
26 Baritiu Street, 3400 Cluj-Napoca, Romania
e-mail: eneia@utcluj.ro

Abstract In this paper we generalize the *cut* primitive in PROLOG for the concurrent framework (with sequential and parallel don't know non-determinism) in NPL. For the resulted language we present some sample programs and a (briefly commented) PROLOG prototype.
Keywords: control flow, non-determinism, concurrency.

1. Introduction

In the last three years (since 1993) we have collected theoretical and experimental results concerning the semantics and the pragmatics of concurrency under the generic name NPL. The language NPL is intuitively obtained by parameterizing a process algebra (the process layer) with LISP expressions (the data layer). NPL provides operators on processes for sequential and parallel don't know non-determinism and a special process !, named *select* that - when used (its use is not mandatory) - uniquely selects the current non-deterministic alternative in a set by destroying all the others. In the sequel we shall name this primitive *relative select*. The main difference between the *relative select* and the *cut* primitive in PROLOG is that (unlike *cut*) the *relative select* does not remove the backtracking points generated by the previously executed processes of the current non-deterministic alternative.

In this paper we introduce a new version of the *select* primitive, denoted by !!, and named *absolute select*. The *absolute select* is a generalization of the PROLOG's *cut* primitive for the concurrent framework in NPL with sequential and parallel don't know non-determinism.

2. An informal introduction to the language

We briefly recall the syntax of NPL. The statements of the language are built by LISP expressions, process names and the two primitives mentioned in the introduction: ! (*relative select*) and !! (*absolute select*), combined by means of operators on processes. Process names are distinguished from LISP symbols by beginning with an uppercase letter (LISP symbols must begin with a lower case letter; in this paper we only deal with parameterless processes in NPL). There is a special process named 'Root' which has the task to create all the other processes for a certain program. NPL provides (binary) operators for sequential composition ($\cdot$), for parallel composition ($\|$) (with a lower precedence than '$\cdot$'; curly braces may be used to force the precedence in expressions), and two (n-ary) operators for non-deterministic choice (in the general sum notation): $[s_1 + ... + s_n]$(abbreviated in the sequel by $[+]$) and $< s_1 + ... + s_n >$ (abbreviated in the sequel by $< + >$).Thus, if we denote by 'e' an (arbitrary) LISP expression, and by 'X' an (arbitrary) procedure variable the syntax of NPL can be summarized as follows:

$$s ::= e \mid X \mid ! \mid !! \mid s_1 \cdot s_2 \mid s_1 \| s_2 \mid [s_1 + \ldots + s_n] \mid < s_1 + \ldots + s_n >$$

In NPL the non-determinism is a union of behaviors. The operator $[s_1 + \ldots + s_n]$introduces an ordering relation (specified by the textual order) between the non-deterministic alternatives (s_i). Operationally, this comes to a mechanism of backtracking. The operator $< s_1 + \ldots + s_n >$ specifies a parallel evaluation of the non-deterministic alternatives. In both cases the non-deterministic alternatives operate on different data. There is a mechanism of "negation by failure" in NPL. The evaluation of any LISP expression to *nil* produces the abandon of the continuation of the current non-deterministic alternative of the process. Thus, in NPL, logic is a consequence of an interpretation of process behavior. The values of truth are inherited from LISP (*nil* for *false* and everything else for *true*). The operators on processes allow for the following logic interpretation: $[+]$ = (sequential) OR, $< + >$ = (parallel) OR, $\cdot$ = (sequential) AND, $\|$ = (parallel) AND. Also, in [7] it has been shown how to extend the LISP in NPL with a logic of work with free and bound variables as a basis for a dataflow mechanism. The basic idea of the dataflow behavior is suggested by the following example. Assume that the variables x, y and z are (initially) unbounded (undefined). The following NPL program has the predictable behavior of assigning them the values $x = 1, y = 3$ and $z = 2$:

```
(setf x 1)‖(setf y (+ x z))‖(setf z 2)
```

A *while* loop can be implemented in NPL as suggested in the following example that computes (and prints) the length of a list.

```
Root = (setf l '(1 2 3)) · (setf n 0) · While · (print  n)
While = [(null l) + l.(list (setf l (cdr l))) · (setf n (+ 1 n)) · While]
```

The program is deterministic. The guards 'l' and '(null l)' can be inspected in any order, so we could have used the operator $< + >$instead of $[+]$.

The main contribution of the present paper is the presentation of two control primitives: ! named *relative select*, and !! named *absolute select*. The primitive ! uniquely selects the *current non-deterministic alternative* (hereafter we abbreviate this expression by *cna*) of an operator $[+]$or $< + >$by removing all the others. The primitive !! performs the task of ! and, additionally, it removes the non-deterministic choice points collected as a result of the previously executed processes (operators $[+]$and $< + >$)of the *cna*.

In the following example (in which e_1, e_2, ... denote LISP expressions): $< e_1 \| [e_2 + \{e_3 \| e_4\} \cdot !^1 + e_5]^2 \cdot !^2 + X \cdot !!^3 >^1$;with $X = [e_5 + e_6]^3$,$!^2$ and $!!^3$ are related to $< + >^1$and $!^1$ is related to $[+]^2$.The execution of $!^2$ determines the abandon of the evaluation of $X \cdot !!^3$ but it does not inhibit the backtracking points generated by the evaluation of $[e_2 + \{e_3 \| e_4\} \cdot !^1 + e_5]^2$. On the other hand, the execution of $!!^3$ produces the abandon of the evaluation of the other non-deterministic alternative $(e_1 \| [e_2 + \{e_3 \| e_4\} \cdot !^1 + e_5]^2 \cdot !^2)$and it also inhibits the backtracking points generated by the evaluation of X $(= [e_5 + e_6]^3)$.

In the picture below we suggest the (parallel) evaluation of the non-deterministic alternatives (of the operator $< + >$) by arrows. By double arrows we suggest the collection of choice points in each non-deterministic alternative.

<u>*Relative select*</u>

<⇒
...
⇒!⇒ (*cna*)
...
⇒>

<u>*Absolute select*</u>

<⇒
...
⇒!→ (*cna*)
...
⇒>

Remark that both the *absolute select* and the *relative select* produce the abandon of the evaluation of all but the *cna*. Additionally, the *absolute select* removes the choice points collected by the previously executed processes of the *cna*. In picture, after the execution of !!, the execution of the *cna* is suggested by a single arrow. Similar considerations take place in connection with the evaluation of the operator [+]. The main difference is that, in this case, the non-deterministic alternatives are executed in sequence. Again, the !! primitive is a strengthened version of the ! primitive, that not only (uniquely) selects the *cna* of an operator [+], but it also removes all the non-deterministic choice points (backtracking points) collected from the beginning of the evaluation of the *cna*.

For the moment we do not know whether both primitives ! and !! are indeed necessary in applications. Usually, one wants to remove all the backtracking points collected for a given process. Thus, it seems that the *absolute select* is enough for practical purposes. All our experiments sustain this affirmation. Throughout this paper, we use the both primitives, even though, in all the cases, the *relative select* can be replaced by the *absolute select* without changing the meaning of the programs. We continue with some NPL sample programs.

The processes 'SeqOnTree' and 'ParOnTree' below perform a sequential, respectively a parallel search of an element in a tree. The primitive *relative select* can be used in both cases in order to stop the search process as soon as the element was found.

```
SeqOnTree = tree1 · [(equal  e (car tree1)) · ! +
                (setf  tree1 (cdr (cdr tree1))) · SeqOnTree · ! +
                (setf  tree1 (car (cdr tree1))) · SeqOnTree]
ParOnTree = tree2 · < (equal e (car tree2)).! +
                (setf tree2 (car (cdr tree2))) · ParOnTree.! +
                (setf tree2 (cdr (cdr tree2))) · ParOnTree.! >
```

Further, we can define a process 'OnBoth', that tests whether the element is simultaneously present in both trees.

```
OnBoth = SeqOnTree ‖ ParOnTree
```

An executable process is obtained as follows.

```
Root = Init · [OnBoth · ! · (print 'yes) + (print 'no)]
Init = (setf tree1 '(1 (2 (2 nil) 15 nil) 2 (7 nil) 12 nil))‖
    (setf tree2 '(10 (70 nil 2 nil) 200 (70 nil) 120 nil))‖(setf e 2)
```

Now, we want to define a process that determines if an element that is simultaneously present in two lists. For each element of one of the two lists we must test whether it is present in the other list. We can do this as follows.

```
Root = [{(setf l1 '(1 5 7 3 8 10 2))‖(setf l2 '(3 1 80 10 7 4 45 5 12))} ·
        Common_Term · ! · (print term1) + (print 'no_common_term)]
One_Term1 = l1 · < (setf term1 (car l1)) + (setf l1 (cdr l1)) · One_Term1 >
One_Term2 = l2 · < (setf term2 (car l2)) + (setf l2 (cdr l2)) · One_Term2 >
Common_Term = < {One_Term1‖ One_Term2} · (equal term1 term2) >
```

However, the above program will print in a random order all the common elements of the two lists, because both 'One_Term1' and 'One_Term2' are (don't know) non-deterministic processes. Thus the program collects all the possible solutions of the problem. Now, we want to abandon the search as soon as the first common term was found. We try the following variant of the process 'Common_Term'.

```
Common_Term = < {One_Term1‖ One_Term2} · (equal term1 term2) · ! >
```

This process will not provide us with the desired behavior. In fact, the *relative select*, as used above, has no effect, because it only removes the non-deterministic alternatives in a sum context, and the process above has only one non-deterministic alternative. In such situations we need the *absolute select*. The desired behavior is obtained by the process below.

```
Common_Term = < {One_Term1‖ One_Term2} · (equal term1 term2) · !! >
```

We must point out here that, the definition of the language ensures that the effect of the primitives *relative select* and *absolute select* restricts to the syntactic scope of the operator [+] or < + >in which they textually appears. Thus, if we run the process 'Common_Term' in parallel (Common_Term‖All) with the following process that retrieves all the elements in a list in a (don't know) non-deterministic manner:

```
All = l · [(setf e (car l)) + (setf l (cdr l)) · All]
```

the *absolute select* in the definition of 'Common_Term' will not remove the backtracking points generated by the process 'All'.

In the sequel we will present strategies of search based on various combinations of the operators [+]and < + >.We consider the search of a all the paths between a start node and a goal node in a graph. The graph is represented as an association list indexed with the nodes of the graph. To each node it is attached the list of its neighbors.

```
Root={(setf graph '((a b c d e)(b a c h nil)(c a b nil nil)
        (d a e h nil)(e a d g nil)(h b d g nil)(g h e nil nil)))
        ‖(setf node 'a)‖(setf goal 'g)‖(setf path '(a))}.Run
Run=Search·(print path)
Search=<(equal node goal)+(not (equal node goal))·
        (setf neighbors (cdr (assoc node graph)))·Sel·
        (setf p path)·NotMember·(setf node neighbor)·
        (setf path (cons neighbor path))·Search>
NotMember=<(null p)·!+p·!·(not (equal (car p) neighbor))·
        (list (setf p (cdr p)))·NotMember>
```

The effective strategy of search is determined by the process 'Sel' that non-deterministically selects a neighbor for the current node. A PROLOG-like depth first search is obtained if we use the operator [+] in the definition of the process 'Sel'.

```
Sel=neighbors·[(setf neighbor (car neighbors))+(setf neighbors (cdr neighbors))·Sel]
```

A parallel search is obtained if we use the operator < + >.

```
Sel=neighbors·<(setf neighbor (car neighbors))+
                (setf neighbors (cdr neighbors))·Sel>
```

This time the solutions are returned in an unpredictable order.

The problem of search is in general NP-complete. The time complexity t(n) of a depth-first search - where n is the "depth" of the search tree - is exponential in n, i.e. $t(n)=O(f^n)$ where f is the "fan-out" of the search tree. A parallel search can be very fast $(t(n)=O(n))$ but it assumes the availability of a number of processors that is exponential in n, $p(n)=O(f^n)$. For $n=10$ and $f=4$ (we assume that the search tree is uniform, i.e. each node has 4 sons), either t(n) or p(n) would be of the order of $10^6(=4^{10})$. The combination of the two operators for non-deterministic choice (in the process that selects a neighbor) can distribute the computing task among the two coordinates of the algorithm (time and number of processors). For $n=10$ and $f=4$ (i.e. we assume that each node has maximum 4 neighbors) we obtain an algorithm with both t(n) and p(n) of the order of $10^3(=2^{10})$ by using the following process for neighbor selection.

```
Sel=<[(setf neighbor (car neighbors))+
      (setf neighbor (car (cdr (cdr (cdr neighbors)))))]+
      [(setf neighbor (car (cdr neighbors)))+
      (setf neighbor (car (cdr (cdr neighbors))))]>
```

In all the cases above, the search process can be interrupt immediately after the first path was found. This can be done by appropriately using the primitive !! in the definition of the process 'Run'.

```
Run=<Search·!!·(print path)>
```

3. A PROLOG prototype for NPL

In order to gain more intuition on the meaning of the NPL programs we present the following construction. Consider the class Id of identifiers with typical elements α,β, defined by: $\alpha ::= \alpha{:}i \mid i,\ i \in N$ (i is a natural number). Remark that, on the class of identifiers Id, we can define a partial ordering $\leq$ as follows: $\alpha \leq \beta$ if and only if $\beta = (\ldots(\alpha{:}i_1){:}\ldots){:}i_k$ for some $i_1,\ldots,i_k \in N$. We make a correspondence between each apparition of a primitive ! or !! and the corresponding operator $[+]$ or $<+>$ in which it textually appears. We do this by labeling each instance of an operator $[+]$ or $<+>$ with a (unique) identifier α (we obtain $[+]^\alpha$ or $<+>^\alpha$) and we label all the apparitions of a primitive ! or !! in the textual context of $[+]^\alpha$ or $<+>^\alpha$ with (the same identifier) α. The labeling of the operators $[+]$ and $<+>$ will be based on a tree-like discipline (following the syntactic structure of the expressions) as suggested in the following picture:

$$\begin{array}{ccccccc} & & & \alpha & & & \\ & \alpha{:}1 & & \ldots & & \alpha{:}k & \\ \alpha{:}1{:}1 & \ldots & \alpha{:}1{:}m & & \alpha{:}k{:}1 & \ldots & \alpha{:}k{:}n \end{array}$$

Thus, given a NPL program $<X_0, D>$ with X_0 the 'Root' process and D a list of process definitions $D \equiv\ <X_i = g_i>_{i=1,\ldots,n}$ (where X_i are recursion variables and g_i are guarded NPL statements; see [8]) we generate an infinite specification: $D^\alpha \equiv\ <X_i^\alpha = ren(\alpha,\alpha,g_i)>_{i=1,\ldots,n}$ where the function 'ren' is defined as follows:

$$\begin{aligned}
&ren(\alpha,\beta,!) = !^\alpha \\
&ren(\alpha,\beta,!!) = !!^\alpha \\
&ren(\alpha,\beta,e) = e \\
&ren(\alpha,\beta,X) = X^\beta \\
&ren(\alpha,\beta,s_1 \cdot s_2) = ren(\alpha,\beta{:}1,s_1) \cdot ren(\alpha,\beta{:}2,s_2) \\
&ren(\alpha,\beta,s_1 \| s_2) = ren(\alpha,\beta{:}1,s_1) \| ren(\alpha,\beta{:}2,s_2) \\
&ren(\alpha,\beta,[s_1 + \ldots + s_n]) = [ren(\beta,\beta{:}1,s_1) + \ldots + ren(\beta,\beta{:}n,s_n)]^\beta \\
&ren(\alpha,\beta,<s_1 + \ldots + s_n>) = \ <ren(\beta,\beta{:}1,s_1) + \ldots + ren(\beta,\beta{:}n,s_n)>^\beta
\end{aligned}$$

Thus, for $D = \ <X = [e_1 \cdot ! + Y \cdot !!] \| e_2 \cdot X,\ Y = \ <e_3 \cdot !! + X \cdot !>>$, we generate

$$\begin{aligned}
D^\alpha = \ <X^\alpha &= [e_1 \cdot !^{\alpha:1} + Y^{\alpha:1:2:1} \cdot !!^{\alpha:1}]^{\alpha:1} \| e_2 \cdot X^{\alpha:2:2}, \\
Y^\alpha &= \ <e_3 \cdot !!^\alpha + X^{\alpha:2:1} \cdot !^\alpha>^\alpha>_{\alpha \in Id}
\end{aligned}$$

Now remark that, by expanding each apparition of a recursion variable X^α or Y^α according to its definition in D^α up to an arbitrary (finite) level, we generate an expression in which each instance of an operator $<+>$ or $[+]$ is labeled with a unique identifier. Moreover, each apparition of a primitive $!^\alpha$ or $!!^\alpha$ appears in the textual context of an operator $<+>^\alpha$ or $[+]^\alpha$ (labeled with the same identifier α). Now, by definition, <u>the execution of $!^\alpha$ removes all the nondeterministic alternatives except for the current one only for the operator $<+>^\alpha$ or $[+]^\alpha$ with the same identifier α</u>. $!!^\alpha$ is a strengthened version of $!^\alpha$. Thus, <u>$!!^\alpha$ removes all the nondeterministic alternatives except for the current one for all the operators $<+>^\beta$ and $[+]^\beta$ with $\beta \geq \alpha$</u>. Remark that, due to the way we defined the function 'ren', only the processes in the textual context of the corresponding operator $[+]^{\alpha}$ or $<+>^{\alpha}$ will be labeled with identifiers $\beta \geq \alpha$. Thus, in the example above, $!!^{\alpha:1}$ will not cut the backtracking (don't know non-deterministic choice points) generated by the evaluation of $e_2 \cdot X^{\alpha:2:2}$. $!!^{\alpha:1}$ will only cut the backtracking generated by the evaluation of the expression $[e_1 \cdot !^{\alpha:1} + Y^{\alpha:1:2:1} \cdot !!^{\alpha:1}]^{\alpha:1}$.

In the sequel we present a TURBO PROLOG prototype for the executive of NPL. We take advantage of the fact that this variant of PROLOG is a typed language and it provides us with syntax for the definition of the semantic domains. We do not present here in any detail the LISP evaluator for NPL (it has been presented in [9]). We only assume given a predicate eval(E,V,S,NS) that evaluates a LISP expression E in a given state S and it returns the value V of the expression and a new state NS (the evaluation may have side effects). The domain of the LISP values can be described as follows:

```
lisp=cons(lisp,lisp);s(string);n(integer);nil;t
```

and the predicate eval has arguments of the following types: eval(lisp,lisp,lisp,lisp). An identifier will be represented as a list of integers. Thus, the PROLOG representation for α:i will be [I|Alfa] (where the PROLOG variable I corresponds to the integer value i and the PROLOG list Alfa corresponds to the identifier α). We have:

```
id=integer*
```

The following predicate defines a partial order on the set of identifiers (in the sequel we will always specify the types for the arguments of the predicates in a TURBO PROLOG - like notation).

```
/* leq(id,id) */
leq(I1,I2):-app(_,I1,I2).
/* app(id,id,id) */
app([],I,I):-!.
app([H|I1],I,[H|I2]):-app(I1,I,I2).
```

The understanding of the domain of NPL statements should raise no problem to the reader (these are syntactic structures).

```
lstmt=stmt* /* List of statements */

stmt=seq(stmt,stmt);        /*  s1·s2 */
     par(stmt,stmt);        /*  s1||s2 */
     seq_ned(lstmt);        /*  [s1+...+sn] */
     par_ned(lstmt);        /*  <s1+...+sn> */
     atom(lisp);            /*  e */
     def(string);           /*  X */
     asel;                  /*  !! */
     rsel;                  /*  ! */
     null
```

We do not present here the parser for NPL. We assume given a NPL program by the clauses of the predicate:

```
        process_definition(string,stmt)
```

For the sake of completeness we present below the assumed internal representation of the first program presented in section 2 (the program that computes the length of a list):

```
        process_definition("Root",
                seq(atom(cons(s("setf"),cons(s("l"),cons(cons(s("quote"),
                cons(cons(n(1),cons(n(2),cons(n(3),nil))),nil)),nil)))),
                seq(atom(cons(s("setf"),cons(s("n"),cons(n(0),nil)))),
                seq(def("While"),
                atom(cons(s("print"),cons(s("n"),nil)))))))
        process_definition("While",
                seq_ned([atom(cons(s("null"),cons(s("l"),nil))),
                seq(atom(s("l")),seq(atom(cons(s("list"),
                cons(cons(s("setf"),cons(s("l"),cons(cons(s("cdr"),
                cons(s("l"),nil)),nil))),nil))),
                seq(atom(cons(s("setf"),cons(s("n"),cons(cons(s("plus"),
                cons(n(1),cons(s("n"),nil))),nil)))),def("While"))))]))
```

The configurations of the executive of NPL are based on the following domain definitions:

```
        lconf = conf*
        conf = seq_ned(id,integer,lconf);par_ned(id,integer,lconf);
                ned(lisp,lproc);empty;work
```

seq_ned(id,integer,lconf) and par_ned(id,integer,lconf) are the semantic counterparts for the syntactic constructions seq_ned(lstmt) ([+])and par_ned(lstmt) (< + >)The identifiers are used for the labeling of the operators [+] and < + >as explained at the beginning of this section. The integer argument is the actual arity (number of elements in the list lconf) of each operator [+] or < + >The evaluation of the non-deterministic alternatives of an operator < + >must be performed in parallel. In the PROLOG prototype we simulate the parallelism by an arbitrary interleaving. The following two predicates are used to perform a random choice of a non-deterministic alternative in a list:

```
        /*      choose(integer,lconf,conf,lconf)        */
        choose(K,LN,N,RLN):-random(K,K0),subs(K0,LN,N,RLN).
        /*      subs(integer,lconf,conf,lconf)          */
        subs(0,[N|RLN],N,[work|RLN]):-!.
        subs(K,[N|RLN],N1,[N|NRLN]):-K1 = K-1,subs(K1,RLN,N1,NRLN).
```

'random(N,K)' is a predefined TURBO PROLOG predicate; when it is provided with an integer value N, it generates a new random integer K, such that $0 \leq K < N$. Remark that the chosen non-deterministic alternative is (temporary) replaced by the constant 'work'. After one step of evaluation, the non-deterministic alternative is put back, in the same place in the original list of non-deterministic alternatives. This task is performed by the predicate 'ins'.

```
/*      ins(conf,lconf,lconf)  */
ins(NN,[work|RN],[NN|RN]):-!.
ins(NN,[N|RN],[N|NRN]):-ins(NN,RN,NRN).
```

However, if the evaluation of the non-deterministic alternative finishes or if it fails upon the evaluation to *nil* of a LISP expression (such a non-deterministic alternative is denoted by the constant 'empty') it is simply removed from the original list by a call to the predicate 'rem'.

```
/*      rem(lconf,lconf)       */
rem([],[]):-!.
rem([work|RN],RN):-!.
rem([N|RN],[N|NRN]):-rem(RN,NRN).
```

A non-deterministic alternative consists of a state (a LISP structure in NPL) and a list of processes. The domain of processes is defined as follows:

```
proc=p(id,stmt,id,id)
lproc=proc*
```

In fact, we use lists (of processes) to model partially ordered sets with a tree structure. The physical order of the elements in a list of processes (lproc) is immaterial and the partial order is determined by the value of the first argument (id) of the functor p in the definition of the domain proc. Thus, the following list

```
[p([2,1,1],S1,I11,I12),p([1,1,1],S2,I21,I22),
 p([1,1],S3,I31,I32),p([1],S4,I41,I42)]
```

is the syntactic representation of the tree depicted below.

```
                              p([1],S4,I41,I42)
                             p([1,1],S3,I31,I32)
             p([2,1,1],S1,I11,I12)   p([1,1,1],S2,I21,I22)
```

Only the leaves of the tree (i.e. the maximal elements according with the above defined partial order) are active computing entities. The predicate leaf defined below chooses an arbitrary maximal element (leaf of the tree) in such an (ordered) set.

```
/*      leaf(lproc,proc,lproc)       */
leaf(LP,PP,NRLP):-length(LP,N),random(N,K),pick(K,LP,P,RLP),
        max(P,RLP,PP,NRLP).
/*      length(lproc,integer)     */
length([],0):-!.
length([_|T],N):-length(T,N0),N=N0+1.
```

```
/*      pick(integer,lproc,proc,lproc)        */
pick(0,[P|RLP],P,RLP):-!.
pick(K,[P|RLP],PP,[P|NRLP]):-K1=K-1,pick(K1,RLP,PP,NRLP).
/*      max(proc,lproc,proc,lproc)  */
max(P,[],P,[]):-!.
max(p(I,Si,Ai,Bi),[p(J,Sj,Aj,Bj)|RLP],P,
        [p(I,Si,Ai,Bi)|NRLP]):-
        leq(I,J),!,max(p(J,Sj,Aj,Bj),RLP,P,NRLP).
max(Pi,[Pj|RLP],P,[Pj|NRLP]):-max(Pi,RLP,P,NRLP).
```

The other arguments of the functor p are: a statement and two other identifiers used for the labeling of the operators [+]and <+>as explained at the beginning of this section. In order to distinguish between the control primitives ! (rsel(id)) and !! (asel(id)) and the ordinary atomic actions (evaluations of LISP expressions) we also use the following domain:

```
act=tau;rsel(id);asel(id)
```

The execution of a given NPL program is initiated by a call to the predicate 'os':

```
?-os(ned(nil,[p([],def("Root"),[],[])])).
```

'ned(nil,[p([],def("Root"),[],[])])' represents the initial configuration of the system. The definition of the predicate 'os' is presented below.

```
/*      os(conf)        */
os(empty):-!.
os(C):-r(C,_,C1),os(C1).
```

The predicate 'r' defines a transition relation in the style of the structured operational semantics [5], i.e. we define the transition of a composite structure in terms of the transitions of its constituents.

```
/*      r(conf,act,conf)        */
r(empty,tau,empty):-!.
r(ned(_,[]),Act,C):-!,r(empty,Act,C).
r(ned(S,LP),Act,C):-!,leaf(LP,P,RLP),rn(ned(S,[P|RLP]),Act,C).
r(seq_ned(_,0,[]),Act,C):-!,r(empty,Act,C).
r(par_ned(_,0,[]),Act,C):-!,r(empty,Act,C).
r(seq_ned(I,K,[empty|RN]),Act,C):-!,
        K1=K-1,r(seq_ned(I,K1,RN),Act,C).
r(seq_ned(I,K,[N|RN]),Act,C):-!,
        r(N,Act,NN),new_sc(I,Act,NN,RN,K,C).
r(par_ned(I,K,LN),Act,C):-choose(K,LN,N,RLN),r(N,Act,NN),
        new_pc(I,Act,NN,RLN,K,C).
```

The predicate 'rn' called in the third clause will be explained later. The last two clauses define the evaluation of an operator [+]and < + >Thus, after one transition step performed by one of the non-deterministic alternatives, the predicates 'new_sc' and respectively 'new_pc' compute the new configuration.

```
/*      new_pc(id,act,conf,lconf,integer,conf)       */
new_pc(I,_,empty,RN,K,par_ned(I,K1,LN)):-!,K1 = K-1,rem(RN,LN).
new_pc(I,rsel(I),NN,_,_,NN):-!.
new_pc(I,asel(J),NN,_,_,NN):-leq(J,I),!.
new_pc(I,_,NN,RN,K,par_ned(I,K,LN)):-ins(NN,RN,LN).
/*      new_sc(id,act,conf,lconf,integer,conf)       */
new_sc(I,rsel(I),NN,_,_,NN):-!.
new_sc(I,asel(J),NN,_,_,NN):-leq(J,I),!.
new_sc(I,_,NN,RN,K,seq_ned(I,K,[NN|RN])).
```

Remark the way that they select only one non-deterministic alternative upon the execution of one of the control primitives ! or !!.

Finally, the predicate 'rn' defines the transitions for one (single) non-deterministic alternative; it also performs the labeling of the contained processes.

```
/*      rn(conf,act,conf)     */
rn(ned(S,[p(_,null,_,_)|LP]),Act,C):-!,r(ned(S,LP),Act,C).
rn(ned(S,[p(_,rsel,A,_)|LP]),rsel(A),ned(S,LP)):-!.
rn(ned(S,[p(_,asel,A,_)|LP]),asel(A),ned(S,LP)):-!.
rn(ned(S,[p(_,atom(Lisp),_,_)|LP]),Act,C):-!,
        eval(Lisp,Val,S,S1),new_c(Val,S1,Act,LP,C).
rn(ned(S,[p(I,def(PN),A,B)|LP]),Act,C):-!,
        process_definition(PN,PDef),
        r(ned(S,[p(I,PDef,A,B)|LP]),Act,C).
rn(ned(S,[p(I,seq(S1,S2),A,B)|LP]),Act,C):-!,
        r(ned(S,[p([1|I],S1,A,[1|B]),p(I,S2,A,[2|B])|LP]),Act,C).
rn(ned(S,[p(I,par(S1,S2),A,B)|LP]),Act,C):-!,
        r(ned(S,[p([1|I],S1,A,[1|B]),p([2|I],S2,A,[2|B]),
        p(I,null,A,B)|LP]),Act,C).
rn(ned(S,[p(I,seq_ned(LS),_,B)|LP]),Act,C):-!,
        mul(B,S,I,LS,LP,N,LNed),r(seq_ned(B,N,LNed),Act,C).
rn(ned(S,[p(I,par_ned(LS),_,B)|LP]),Act,C):-
        mul(B,S,I,LS,LP,N,LNed),r(par_ned(B,N,LNed),Act,C).
```

The first three clauses should be clear. In the fourth clause, after the evaluation of a LISP expression, a decision is taken by the predicate 'new_c'. The evaluation to *nil* of the LISP expression produces the failure of the current non-deterministic alternative.

```
/*      new_c(lisp,lisp,act,lproc,conf)      */
new_c(nil,_,Act,_,C):-!,r(empty,Act,C).
new_c(_,S,Act,LP,C):-r(ned(S,LP),Act,C).
```

The fifth clause embodies procedure execution by body replacement. The sixth and the seventh clauses deal with the sequential composition ($\cdot$) and with the parallel composition ($\|$) operators in NPL. New processes are created and put in an appropriate order for further execution. Finally, the last two clauses model the creation of a new (semantic) structure seq_ned or par_ned. Entire contexts, consisting of a common state and a tree of processes are multiplied. A new instance of the current non-deterministic alternative is created for each term s_i of a composite statement $[s_1 + ... + s_n]$ or $< s_1 + ... + s_n >$ This operation is performed with the help of the predicate 'mul'.

```
/*      mul(id,lisp,id,lstmt,lproc,integer,lconf)      */
mul(_,_,_,[],_,0,[]):-!.
mul(B,ST,I,[S|RLS],LP,N1,[ned(ST,[p(I,S,B,B)|LP])|RLNed]):-
        mul(B,ST,I,RLS,LP,N,RLNed),N1=N+1.
```

4. Conclusions

The main contribution of this paper is the introduction of a new control feature in NPL which is a generalization of the *cut* primitive in PROLOG. For the resulted language we have presented some justifying programming examples and a (briefly commented) PROLOG prototype. The PROLOG prototype is a good starting point for the definition of an operational semantics for NPL in the style of the structured operational semantics of Plotkin.

References

[1] S.G. Akl. *Design and Analysis of Parallel Algorithms*. Prentice Hall, (1989).
[2] J.C.M. Baeten, C. Weijland. *Process Algebra*. Cambridge University Press, (1990).
[3] J.W. De Bakker, J.J.M.M. Rutten, (editors). *Ten Years of Concurrency Semantics*. World Scientific Publishing Co. Pte Ltd. (1992).
[4] K.L. Clark, S. Gregory. *Parallel Programming in Logic*. ACM Trans. Programming Language Systems 8(1), (1986).
[5] G.D. Plotkin. *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ, (1981).
[6] E.Y. Shapiro. *Concurrent Prolog: Collected Papers*, Vols. 1, 2, MIT Press, Cambridge, MA, (1988).
[7] E. Todoran. *Dataflow Semantics in NPL*, in Proceedings of the 1st Conference on Control and Technical Informatics, Timisoara, Romania, (1994).
[8] E. Todoran. *An Operational Semantics for NPL*, in Proceedings of ROSE '95, the 3rd Romanian Conference on Open Systems, Bucuresti, Romania, (1995).
[illegible] E. Todoran. *A Process Algebra Language*, in Proceedings of the 9th Romanian Symposium on Computer Science (ROSYCS '93), Iasi, Romania, (1993).