ACAM

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

AUTOMATION COMPUTERS APPLIED MATHEMATICS

SCIENTIFIC JOURNAL
Vol. 2. No. 2 - 1993
ISSN - 1221-437X

NPL: Design, Semantics and Programming examples

Eneia Todoran
Department of Automation and Computer Engineering,
Technical University of Cluj-Napoca, Romania

Abstract

The aim of this paper is to present the principles of the NPL programming language. NPL is structured on two distinct semantic layers: the data layer (elementary actions are specified by Lisp expressions) and the process layer which is an extension of the process algebra. NPL processes may have parameters and local data. The strength of NPL is supplied at the process level by non-determinism and concurrency. Besides some other examples the use of the process trees for data structure traversals is considered. The concept of process normalization is discussed in connection with logic and functional programming.
Key words: semantics, normalized processes, concurrency, process trees.

1.Introduction

The first version of NPL was introduced in [11]. NPL is the acronym for "**Normalized Processes parameterized with Lisp expressions**". In [11] an operational semantics for process algebra based on rewriting techniques was presented. To this end the concept of process normalization was introduced over the theory of process algebra. For normalized processes continuations are always directly available. The execution may follow a simple scheme: (1) execution of the current action, (2) rewriting of the process as its continuation, and (3) execution of the continuation process.

In [11] NPL was introduced as a machine for the study of the concurrency. That version was theory oriented (a predicative specification of process algebra). For the "‖" operator of process algebra its definition from the process algebra theory was considered (i.e. the process scheduler was considered to introduce a non-deterministic behavior). Later, we extended process definitions with parameters and local data and for the processes scheduling we chose a round-robin policy. In this way NPL becomes a (concurrent) programming language.

The strength of NPL (at the process level) is due to the combination of non-determinism and concurrency. Non-determinism is implemented by backtracking. In NPL there is a mechanism that closely resembles the "negation as failure" from Prolog (however, the NPL processes are much more general than the Prolog processes; we will make a parallel between the two languages in section 6.). Every (Lisp) expression that evaluates to NIL forces backtracking to the nearest "choice point". In this extended version the non-determinism is expressed only explicitly by the "+"operator of the process algebra. The "negation as failure" may be used in NPL to destroy processes.

In this paper we present some NPL programming examples in which the emphasize is on the process orientation of the algorithms. For the traversal of data structures NPL may be considered as a "declarative" programming language. The programmer has only to "declare" a process tree that "covers" the data structure (a graph, a tree or a list). It is not more difficult to declare a process tree in NPL than it is to declare a data structure in Pascal for instance.

The concept of process normalization over the theory of process algebra is discussed in connection with other programming paradigms (i.e. logic and functional) which allow for an operational semantics based on rewriting techniques.

In the end we will present some possible extensions of NPL.

2. Some basic design considerations

NPL is a language designed at two distinct semantic layers. For the data representation and manipulation NPL uses Lisp expressions. For processes, NPL uses the mathematical theory of process algebra. However, in NPL there may be processes defined with or without parameters and local data. For instance, to compute the length of a list we may use "iteration":

```
length_rep[(LIST)(N)] = (SETF N 0)•while
while = (EQUAL L NIL) +
     (NOT (EQUAL L NIL))•(SETF L (CDR L) N (+ N 1))•while
```

or "recursion":

```
length_rec[(LIST)(N)] = (EQUAL L NIL)•(SETF N 0) +
        (NOT (EQUAL L NIL))•length_rec[((CDR L))(N1)]•(SETF N (+ N1 1))
```

For the process algebra operators the following precedence is assumed: "•" binds stronger than "‖" which in turn binds stronger than "+". Curly braces may be used to force precedence over process algebra expressions in NPL.

A parameterless process works on the data of the process who calls it. A process with parameters creates its own data. The values of the parameters at the moment of the process call represent its start state. If the process ends it returns to the calling process its final state. At this level the design of NPL was inspired by the metaphor of the "state space" [6]. For a program that operates on N variables, the corresponding "state space" is visualized as an N-dimensional space with the N variables of the program as N Cartesian coordinates. This metaphor describes the sequence of states corresponding to a computation as a "path" through the state space, which is traversed by that computation. The metaphor allows us to view certain program transformations (as a procedure call for instance) in which program variables are replaced by others as coordinate transformations.

The parameters of a process in NPL are divided in two groups. The parameters in the first group represent the initial state of the process and are transmitted by value (we must point out here that the parameter passing is treated in NPL as a process that never fails, that is, the evaluation of a parameter to NIL will not force backtracking). If the process ends, the parameters in the second group (they must be Lisp symbols, i.e. variable names) represent its final state and are returned to the calling process. The process may also create other local data using Lisp primitives like SET or SETF.

NPL also uses a memory area for process communication. This memory area is accessible as a global area. A variable is first searched in the local area of the current process. If it is not found then the variable is searched in this global area. A process may create or alter data in this area using the commands: ASET and ASETF. The data in this area are not destroyed when a process fails (are "persistent" to failure) as it happens with the data in the local process area. Such data however may be destroyed explicitly using the command "(RESET <variable-name>)".

Let us now return to the problem of the computing of the length of a list. For a deterministic and sequential behavior a functional specification is often preferred:

```
root[()()] = (DEFUN LENGTH (L) (COND ((NULL L) 0)
                                      (T ( + 1 (LENGTH (CDR L)))))) •
     (READ LIST) • (TERPRI) • (PRINT (LENGTH LIST))
```

The lambda expressions (created with DEFUN) are always memorized in the global (and persistent) memory area, no matter which is the process which creates them.

In NPL there is a process called "root" which is the (direct or indirect) ancestor of any process. Other processes may be created either by call or by the use of the process algebra operator "‖".

An important facility offered by NPL is the process synchronization. This mechanism is mainly used for communication. This technique is inspired by the Hoare's rendez-vous. For communications there are neither queues for messages nor is the task of the programmer to keep track of process id's or addresses. In fact, in NPL processes may be created dynamically and such a book-keeping is not possible in general. Synchronization may be specified by a pair of synchronization commands. The mechanism of communication by process synchronization in NPL is presented in [11]. Using a (raw) version of NPL, in [11] we have also presented an example of two communicating processes that reverse a list at superficial level. We present the same example using facilities introduced in the actual version of NPL:

```
root[()()] = init • {send[(L)()] ‖ receive[()(RL)]} • (PRINT RL)
init = (READ L) • (LISTP L)
send[(SL)()] = sendloop • ((SYNC 7)(LIST (ASETF EL NIL)))
sendloop = (EQUAL SL NIL) + (NOT(EQUAL SL NIL)) •
      ((SYNC 7)(ASETF EL (CAR SL))) • (LIST (SETF SL (CDR SL))) • sendloop
receive[()(REVL)] = (LIST (SETF REVL NIL)) • receiveloop
receiveloop = ((SYNC-7) T) • {(EQUAL EL NIL) +
      (NOT (EQUAL EL NIL)) • (SETF REVL (CONS EL REVL)) • receiveloop}
```

3. Semantic considerations and interpretation

In this section we present a way to interpret the meaning of the NPL programs. This will be done at two distinct semantic levels: the data level and the process level.

In its first version (see [11]) NPL was using Lisp only as a "data algebra". However, except for the addition of an environment with Lambda expressions the extension of elementary actions with the entire power of Lisp (as a functional programming language) raises no conceptual difficulties. In the actual version, every atomic action in NPL is either a Lisp primitive or a (Lisp) user defined function.

At the level of processes NPL may be seen as a logic programming language. There is a natural connection between logic and control. The truth values in NPL are inherited from Lisp (NIL for FALSE and everything else for TRUE). After the elimination of the non-determinism introduced by the process scheduler the operators of process algebra allows for the following logical
interpretation:

"+"	- OR
"•"	- AND
"∥"	- (a "faster") AND

That is, the process $p_1 + p_2$ ends either by the execution of p_1 OR by the execution of p_2, while $p_1 \bullet p_2$ and $p_1 \parallel p_2$ ends by the execution of both p_1 AND p_2. So, in NPL, logic is a consequence of an interpretation of process behavior. The NPL processes allow for a (first order) predicative interpretation.

The mathematical meaning of synchronization is given by an operator of "restriction" (see [3]). We must point out here that synchronization does not alter the above presented logical interpretation of the NPL-operator "∥".

The following example is the widely known program for "comparing tree profiles". A tree profile is the list of its nodes in the order in which they are met by a traversal (for instance by an infix traversal).

```
root[()()]=(SETFTREE1 '(2 (1 NIL) 3 NIL) TREE2 '(1 NIL 2 NIL 3 NIL))•
           { { infix[(TREE1 'NAME1 'NAME2 7)()] ∥
               infix[(TREE2 'NAME2 'NAME1 -7)()] }•!•(PRINT 'YES) +
           (PRINT 'NO) }
infix[(TREE EL1 EL2 SYNCNR)()]=(NULL TREE)+(NOT(NULL TREE))•
           infix[((CAR (CDR TREE)) EL1 EL2 SYNCNR)()]•
           ((SYNC SYNCNR)(ASET EL1 (CAR TREE)))•
           (EQUAL (EVAL EL1)(EVAL EL2))•
           infix[((CDR (CDR TREE)) EL1 EL2 SYNCNR)()]
```

The assumed (Lisp) representation for the binary tree is depicted in the bottom left corner of the figure 5.2.

Here, the use of the "negation as failure" is the key idea. Two (infix) tree traversals work synchronously. The tree traversal algorithm is recursive. This is a typical parallel algorithm because a sequential algorithm should compute both profiles, and just after that it could compare them. In the parallel algorithm there are two processes that synchronize for each pair of nodes in the trees. As soon as a pair of distinct nodes is met the processes are automatically destroyed by the evaluation to NIL of the expression "(EQUAL (EVAL EL1)(EVAL EL2))". The process infix[...] is deterministic (the guards "(NULL TREE)" and

"(NOT (NULL TREE))" are disjoint). Thus, in the case of failure, the only choice point is in the process root[...] which will be able to reach a final state only by the execution of the command "(PRINT 'NO)".

In a logical interpretation the parallel execution of the two infix[...] processes is an "interleaved" conjunction. The elements of the conjunction are tests of identity for each pair of nodes in the trees. If the tree profiles are not identical, one of the elements of the conjunction is "FALSE" (NIL in NPL's logic) and thus the entire process fails.

The above presented NPL program uses another Prolog like facility, namely a special process denoted by "!" that "cuts the backtracking". We must point out here that we have been experimenting most of the facilities introduced by the actual version of NPL on a prototype implemented in the language Turbo Prolog under MS-DOS. The intented meaning of the process that cuts the backtracking is inspired from Prolog (it allows to abandon the alternate variant specified by the nearest apparition of the process algebra operator "+"; the problem is that the NPL processes are more general than the Prolog ones). However, it is the only facility not experimented in the implemented version of NPL. In the above presented program if the parallel execution of the processes that perform the tree traversals finishes, then the (Lisp) form "(PRINT 'YES)" is evaluated. However, the use of the NPL special process "!" makes the root[()()] process deterministic, that is, the form "(PRINT 'NO)" will no longer be evaluated. We must point out here that in NPL, the order of inspection of the non-deterministic alternatives of a process coincides with the order of their apparition in the textual specification of the process. That is, the process $p = q + r$ is first interpreted as q, and than it is interpreted as r.

4. Non-determinism and concurrency

"One inconvenient thing about a purely imperative language is that you have to specify far too much sequencing. For example, if you write an ordinary program to do matrix multiplication, you have to specify the exact sequence in which all of the n^3 multiplications are to be done. Actually, it doesn't matter in what order you do the multiplications so long as you add them together in the right groups. Thus the ordinary sort of imperative language imposes much too much sequencing, which makes it very difficult to rearrange if you want to make things more efficient"

C. Stratchey (1966)

NPL allows for a "procedural design" of the algorithms. It provides recursion, and the control flow structures of an imperative programming language can be specified as process algebra expressions. Thus, the "if...else" and "while" constructions (that are known to "cover" the semantics of the sequential control structures) are semantically equivalent to the processes:

$$\text{if} = \text{condition} \bullet \text{statement}_1 + (\text{NOT condition}) \bullet \text{statement}_2$$

$$\text{while} = (\text{NOT condition}) + \text{condition} \bullet \text{statement} \bullet \text{while}$$

An immediate optimization can be done by means of the special process "!":

```
        if=condition•!•satement$_1$+statement$_2$

        while=condition•!•statement•while+(NOT condition)
```

The Prolog programmer is familiar with those techniques. However, (in contrast with Prolog) NPL does not create new data for a parameterless recursive process call.

In this section, the emphasis is on the power of non-determinism and concurrency in NPL. A graph search is presented both as a non-deterministic and as a concurrent process. A graph path between a start node and a goal node is being searched. The idea is to define one or more processes that "run" towards the goal on the graph paths.

The non-deterministic solution makes use of the mechanism of backtracking in NPL.

```
root[()()]=(SETF GRAPH '((A B C D)(B A C D)(C A B D)(D A B C)))•
       (DEFUN MEMBER (E L) (COND ((NULL L) NIL)
                                 ((EQUAL E (CAR L)) T)
                                 (T (MEMBER (E (CDR L))))))•
       (SETF NODE 'A GOAL 'D PATH '(A))•{search_path+(PRINT 'THE-END)}
search_path=(EQUAL NODE GOAL)•(TERPRI)•(PRINT PATH)•NIL+
       (NOT (EQUAL NODE GOAL))•
       (SETF NEIGHBS (CDR (ASSOC NODE GRAPH)))•
       select_neighb•(NOT (MEMBER NEIGHB PATH))•
       (SETF NODE NEIGHB PATH (CONS NEIGHB PATH))•search_path
select_neighb=(NOT(NULL NEIGHBS))•{(SETF NEIGHB (CAR NEIGHBS))+
       (SETF NEIGHBS (CDR NEIGHBS))•select_neighb}
```

Let us remark that all the computations are made on the data of the process root. Initially the path contains only the start node. For the current node in the path is (non-deterministically) chosen a neighbor. To avoid the cycles the process fails if the neighbor was already inspected (i.e. is present in the partial path) and the process select_neighb is forced to provide another neighbor. This is a Prolog-like strategy. However, the algorithm uses only parameterless processes, i.e. everything is a matter of "control".

A concurrent solution may be given to the same problem in the following manner:

```
root[()()]=(ASETF ONE-PATH NIL
                  GRAPH '((A B C)(B A C D)(C A B E)(D B)(E C)))•
       { search_path[('A 'D '(A))()]+
       {ONE-PATH•(PRINT ONE-PATH)•!+(PRINT'NO-PATH)} }
path[(NODE GOAL PATH)()]=(EQUAL NODE GOAL)•
       (ASETF ONE-PATH PATH)•NIL+
       (NOT (EQUAL NODE GOAL))•
               one_way[((CDR (ASSOC NODE GRAPH)) GOAL PATH)()]
one_way[(NEIGHBS GOAL PATH)()]=(NOT NEIGHBS)+NEIGHBS•
       {path[((CAR NEIGHBS) GOAL (CONS (CAR NEIGHBS) PATH))()] ‖
       one_way[((CDR NEIGHBS) GOAL PATH)()]}
```

The process search_path[...] is deterministic because its guards are always disjoint. Thus, the first process that reaches the goal sets the global (and persistent to failure) variable ONE-PATH and destroys all the (concurrent) processes which currently search a path towards the goal. Practically, the process call search_path[...] in the process root[...] will fail as soon as a graph path is found.

This example is inspired from [5]. The intention of the author was to present the Parlog (concurrent and logic) programming language as a non-deterministic searching machine. However we must point out here that, with respect to the theory in [3] that inspired NPL the above presented program is not correct. Practically, concurrent processes run on all the possible graph paths. Some of them will cycle. In the first version of NPL we used a "depth first" strategy for the scheduling of processes. The choosing of a process was specified by the (non-deterministic) Prolog predicate:

```
choose_process([Process|_],Process).
choose_process([_|Processes],Process):-choose_process(Processes,Process).
```

and every newly created process was to be added at the beginning of the list of processes. Thus, if in the above presented program the first (searching) process cycles on a graph path the process path[...] will never end. In the present version of NPL the process scheduler is specified by the following predicate (which deterministicaly chooses the last process in the list of processes):

```
last([_|Processes],Process):-last(Processes,Process).
last([Process],Process).
```

After performing one action on the chosen process it is added at the beginning of the list of processes. This strategy (a round-robin one) ensures that it is not possible to give the control to one process for an arbitrary length of time (which is theoretically possible). In this case the above presented program behaves correctly.

5. Traversals of data structures

A wide class of applications make use of traversals and processing of data structures. There is an important property of NPL that makes it very useful for such applications. A process algebra expression may be seen as a declaration of a process tree. However, as the definition of the process algebra operator "‖" is "relaxed" in the actual version of NPL (this version does not implement the theoretical non-determinism introduced by the process scheduler) the specification of the process trees may use only the operators of Basic Process Algebra (see [1]), that is, the sequential composition and the non-deterministic choice.

In semantics, a process is usually understood as a behavior of a system. Labelled transition systems have proved to be suitable for describing the behavior (or operational semantics) of a system. A labelled transition system can be viewed as a rooted directed graph of which the edges are labelled by actions, or as a tree of which the edges are labelled by actions, which is obtained by unfolding the graph.

The traversal (with processing if needed) of a data structure may be conceived as a transformation of the structure in a process tree that "covers" it. The role of the stack that implements the recursion in the known programming languages is played here by the paths in a process tree. Let now think about a program that performs a tree traversal in Pascal. The solution is usually based on a recursive algorithm. The basic idea is that the paths of the tree may be found on the stack that implements
the recursion in the language. The stack grows and shrinks as the branches of the tree are more or less "deep". The role of the stack is played in NPL by the paths in the process tree. The advantage is that such a process tree may be simply "declared" in process algebra. Depending on the needs of the application and on the representation of the data structures different traversal strategies may be conceived.

For instance, we may double the values in a list in the global and persistent data area with the process in the following declaration:

```
root[()()]=(ASETF LIST '(1 2 7 3 ...) L LIST)•list_traversal+(PRINT LIST)
list_traversal=(ASETF (CAR L) (*2 (CAR L)) L (CDR L))•list_traversal
```

Such algorithms make use of the mechanism of "negation as failure" in NPL. Every failure of a process may be interpreted as a (return) back in time (at a previous moment and a previous state of the machine). Thus, if we want to perform some processing of the data contained in the data structure we must work in the global (and persistent to process failure) memory area in NPL.

A binary tree (infix) traversal is specified in NPL by the following process:

```
infix=(SETF TREE (CAR (CDR TREE)))•infix+
      (PRINT (CAR TREE))•NIL+
      (SETF TREE (CDR (CDR TREE)))•infix
```

In order to conceive this traversal it is enough to find a transformation of a binary tree in a corresponding process tree. The transformation for the above presented specification is suggested in the figure 5.1. The figure 5.2. contains the process tree for a sample binary tree traversal. The arrows in the process tree signify atomic actions which determine transitions between states. Atomic actions in NPL are specified by Lisp expressions. The atomic actions specified by Lisp expressions that evaluate to NIL force backtracking in NPL and are marked by slashed arrows ($\not\to$) in the figures. The atomic actions a_i ,b_i and c_i correspond to the execution of the atomic actions specified by the Lisp expressions "(SETF TREE (CAR (CDR TREE)))", "(PRINT (CAR TREE))" and "(SETF TREE (CDR (CDR TREE)))" respectively, in the state S_i. The states in figure 5.2. are met in the order S_1, S_2,...,S_8. The assumed (Lisp) representation for the binary tree is depicted in the bottom left corner of the figure 5.2.

The non-deterministic graph search in section 4. may be conceived in the same way. Thus, the problem of finding all the paths between two nodes in a graph is equivalent with the problem of finding a tree that covers the graph and whose leafs contain either detections of the goal node or detections of cycles.

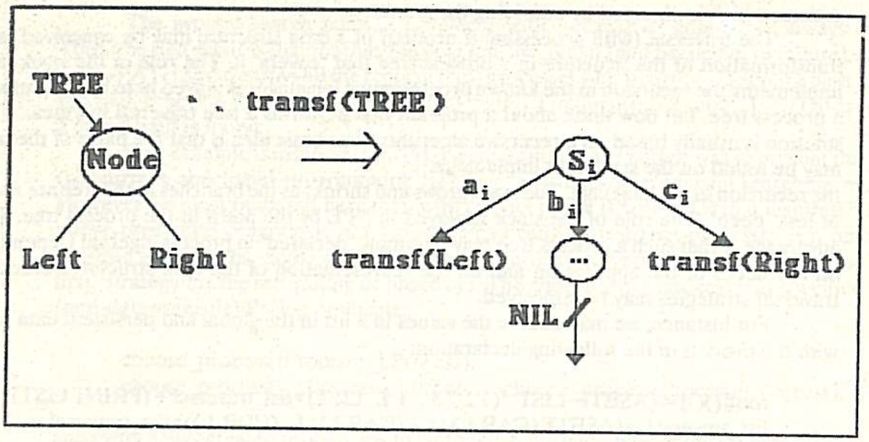
TREE
transf(TREE)
Node
Left
Right
S_i
a_i
b_i
c_i
transf(Left)
...
transf(Right)
NIL

Fig. 5.1.

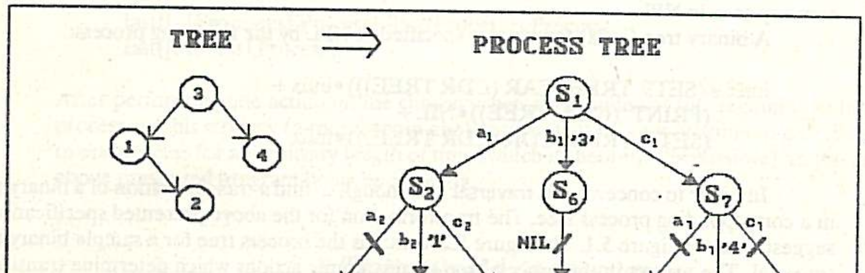
TREE
PROCESS TREE
3
1
4
2
S_1
a_1
b_1,'3'
c_1
S_2
S_6
S_7
a_2
b_2,'1'
c_2
NIL
a_1
b_1,'4'
c_1
S_3
S_4
S_8
NIL
a_4
b_4,'2'
c_4
NIL
S_5
NIL
Node
Left
Right

Fig. 5.2.

6. Comparing NPL and Prolog

NPL and Prolog implement the non-determinism in the same way (i.e. by backtracking). This operational similitude is reflected at the level of the declarative semantics. Thus, at the level of processes, the NPL programs allows for a logic interpretation as presented in section 3. With respect to this connection between the operational semantics of the two languages, the Prolog processes are of the following form:

$$\begin{aligned} p = {} & p_{11} \circ p_{12} \circ \dots + \\ & p_{21} \circ p_{22} \circ \dots + \\ & p_{n1} \circ p_{n2} \circ \dots \end{aligned}$$

The process that cuts the backtracking seems necessary only when we adopt such a logic programming strategy in the design of the algorithms. This strategy seems attractive for a wide class of applications that are suitable to be specified in the theoretical frame of the first order predicate calculus.

However, the NPL processes are more general than the Prolog processes. Even in the absence of the concurrency the NPL programs allow for arbitrary combinations of the process algebra operators "$\circ$" (sequential composition) and "+" (non-deterministic choice). Curly braces may be used in NPL to force precedence over process algebra expressions. For instance, the NPL process:

(1) $p = a \circ \{b + c\} \circ d$

where a,b,c and d are elementary actions or process calls may be expressed in Prolog only by means of a new process definition:

(2) $p = a \circ q \circ d$
$q = b + c$

If we apply the normalization procedure on the process defined in (1) we obtain a process p' with the same meaning:

(3) $p' = a \circ \{b \circ d + c \circ d\}$

Thus, we may remark that the Prolog like processes are a special case of normalized processes. The normalization procedure is based on the following two axioms of process algebra (see [11]):

A4. $(x \circ y) \circ z = x \circ (y \circ z)$
A5. $(x + y) \circ z = x \circ z + y \circ z$.

In [1] is discussed another possible axiom of process algebra, namely:

(?) $x \circ (y + z) = x \circ y + x \circ z$

If this axiom is sound, then every basic process algebra expression is equivalent with a Prolog like process (a non-deterministic choice of a set of sequences of elementary actions or process calls). Thus, p' in (3) would be equivalent with p":

(4) $p'' = a \circ b \circ d + a \circ c \circ d$

However, in [1] this axiom is not considered sound because the moment of the non-deterministic choice in the two sides of (?) is different.

Process normalization is the basis for an operational semantics for process algebra based on rewriting techniques. In this respect, NPL is a generalization of the languages whose operational semantics is based on rewriting techniques. The use of normalized processes is natural both in logic and in functional programming. Such (logic or functional) specifications are expressed as a choice between a set of recursive functional or predicative expressions.

In Prolog there is no assignment statement. Every state has a name. If we need a new value we must provide a new variable. In NPL there is an explicit assignment statement (inspired by the Lisp form SET). Moreover, a parameterless process works on the data of the calling process (no new data is created).

There is a fundamental difference between NPL and Prolog at the level of the data representation and manipulation. At this level NPL provides the strength of the functional programming, however, it lacks unification. Practically, for every possible interpretation of a Prolog predicate (that is, for every i/o pattern for the arguments of a predicate) a distinct process must be specified in NPL.

7. Possible extensions

It would be useful to add a special process call[...] that transforms a Lisp list in a NPL process and than it calls it. For instance, we could use the definition:

```
root[()()]=call[('(type'HELLO-))()]∘(PRINT 'WORLD)
type[(MESSAGE)()]=(PRINT MESSAGE)
```

to print the message "HELLO-WORLD".

This would state a connection between data and processes. In the actual implementation, both the Lisp conses and the process algebra expressions in NPL are internally represented as binary trees. We could unify the internal representation for data and processes. This could be the basis for a higher order process calculus.

NPL may inherit all the facilities supplied by Lisp, including the Common Lisp object system. Thus, NPL could be easily extended with object oriented facilities at the level of the data algebra. However, in this case the methods of the objects could be only Lisp functions. The unification of data and processes could be the basis for an integrated object oriented system, whose methods would be arbitrary NPL processes.

8. Conclusions

We have been presenting the basic principles and mechanisms of NPL. NPL seems to be a very strong (concurrent) programming language. It allows for a functional interpretation at the level of data, and for a logical interpretation at the level of processes. Its strength at the level of processes is supplied by non-determinism and concurrency. NPL may be seen as a declarative programming language for the problem of the data structures traversal.

References

[1] J.C.M. Baeten & F.W.Vaandrager. *An Algebra for Process creation*. Acta Informatica, 29 (1992), pp. 303-334.

[2] J.W. de Bakker & J.N. Kok. *Comparative Metric Semantics for Concurrent Prolog*. Theoret. Comp. Sci., 75 (1990),pp. 15-43.

[3] J.W. de Bakker & J.I. Zucker. *Processes and the Denotational Semantics of Concurrency*. Inform. and Control, 54 (1982), pp. 70-120.

[4] F. van Breugel. *Three metric domains of processes for bisimulation*. Technical report, CWI, Amsterdam, The Netherlands (1993).

[5] T. Conlon. *Programming in Parlog*. Addison Wesley Publishing Company, (1989), 250 pages.

[6] E.J.Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, (1990), 220 pages.

[7] G. Gratzer. *Universal Algebra*. D. Van Nostrand Company, Inc. (1968), 330 pages.

[8] J.J.M.M. Rutten. *Semantic correctness for a Parallel Object-Oriented Language*. SIAM J. Comput., 19 (1990), pp. 341-383.

[9] R. Sethi. *Control Flow Aspects in Semantics-Directed Compiling*. ACM 5, 4 (1983), 554-596

[10] G.L. Steele Jr. et. al. *Common Lisp*. Digital Press, (1990), 970 pages.

[11] E. Todoran. *An Operational Semantics for Process Algebra*. Automation Computers Applied Mathematics, vol. 2 no. 1, (1993), 77-90.

[12] Y. Toyama. *Term Rewriting Systems and the Church-Roser Property*. Ph. D. Thesis (1990), Tohoku University, Japan.

[13] J. Zuidweg. *Concurrent System Verification with Process Algebra*. Ph. D. Thesis (1990) Rijksuniversiteit Leiden, the Netherlands.