TR-029

Mandala: A Concurrent Prolog Based Knowledge Programming Language/System

by
Koichi Furukawa,
Akikazu Takeuchi and Susumu Kunifuji

November, 1983

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Mandala : A Concurrent Prolog Based Knowledge Programming Language / System

Koichi Furukawa, Akikazu Takeuchi, Susumu Kunifuji

Institute For New Generation Computer Technology

Abstract

A knowledge programming language/system called Mandala is being developed on Concurrent Prolog. The final goal of developing Mandala is to provide a tool for knowledge processing which is capable to extract very high parallelism in its execution on the Fifth Generation Computer System(FGCS). Mandala supports both object oriented programming and data oriented programming, which are realized by the same mechanism. These two programming styles will enable us to extract parallelism in knowledge programming. On the other hand, Concurrent Prolog, the implementation language, will make it possible to run Mandala on a highly parallel computer. The main design philosophy is to introduce a multi-plane structure for distinguishing meta level concepts such as modifying and controlling the problem solving strategies from object level concepts. We defined most of the important concepts in knowledge programming, e.g. class, instance, meta class, class variable, is-a hierarchy and part-of hierarchy, in terms of Concurrent Prolog and the multi-plane structure and found the definitions clear and simple.

1. Introduction

Knowledge programming plays a very important role in FGCS as shown in Figure 1. It links between knowledge information processing applications and KL, the Kernel Language for FGCS. To extract very high parallelism in knowledge information processing, we need a suitable formalization in each level of description, that is, in problem description level, in algorithm description level and its execution level. There seems to be very few solutions that achieve such parallelism. The solution we have in mind is shown in Figure 2. Object oriented programming is considered to be a very powerful tool to express solutions to problems in terms of many processes cooperating with each other. Concurrent Prolog [Shapiro 1983, Shapiro & Takeuchi 1983, Takeuchi & Furukawa 1983] is one of the most promising candidates for the parallel algorithm description language, which is a natural extention of Prolog having the capability of expressing concurrency without losing its semantic clarity as a logic programming language.

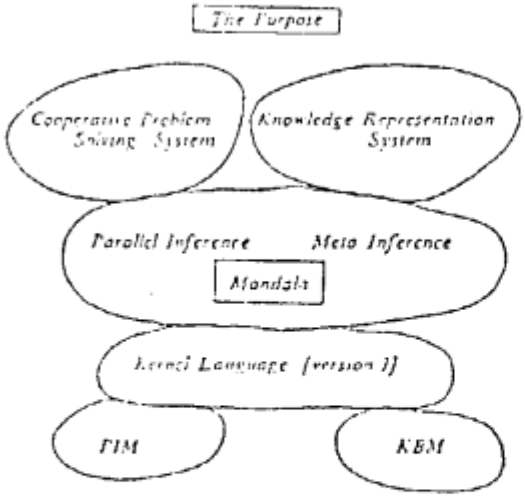

Figure 1. Mandala in FGCS

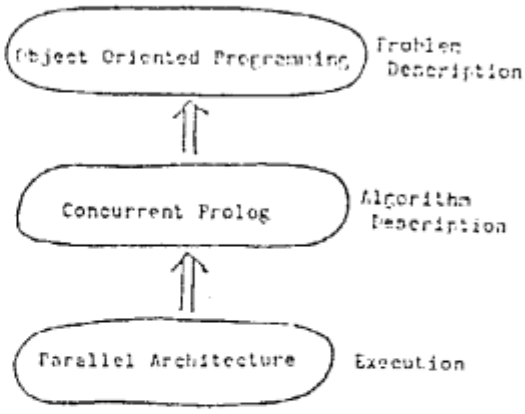

Figure 2. Knowledge information processing system

In this paper, we propose a knowledge programming language/system called Mandala as a language for problem description. It fits to Figure 2 since Mandala can provide an object oriented programming environment. In fact, Mandala provides a structuring mechanism in writing Concurrent Prolog programs as well as in building knowledge bases. It includes most of the key features of knowledge programming languages such as LOOPS [Bobrow & Stefik 1983] and also includes the basic structuring concept of Smalltalk-80

[Goldberg & Robson 1983].

In section 2, Concurrent Prolog is briefly introduced. Conceptual explanation of Mandala is given in section 3. The implementation detail is discussed in section 4 and example programs are given in section 5.

2. Review of Concurrent Prolog

Concurrent Prolog is a logic-based parallel programming language designed and implemented on the DEC-10 Prolog by E. Shapiro [Shapiro 1983]. As the Relational Language [Clark & Gregory 1981] and PARLOG [Clark & Gregory 1983], Concurrent Prolog adopts Or-parallelism as a basis for non-deterministic processing, and And-parallelism for the description of parallel processes. Shared variables are used, with some control information, "variable annotation", as communication channels among concurrent processes.

2.1 Syntax of Concurrent Prolog

The basic style of programs in Concurrent Prolog is quite similar to those in DEC-10 Prolog. In fact, the basic syntactic constructs in Concurrent Prolog have the same meanings as in DEC-10 Prolog. Therefore we neglect the explanation of the basic syntactic constructs and concentrate on the explanation of the difference between them.

[Program] In Concurrent Prolog a program is expressed as a set of guarded clauses.

[Guarded clause] A guarded clauses is a clause which always has "|" (called a commit operator) on its right side as illustrated below.

```
A :- G | B.
```

where G and B are predicates concatenated with logical AND, called a guard part and a body part, respectively. "|" is an extended concept of the cut symbol.

[AND] Logical AND is expressed in the following two ways with operationally different meanings.

sequential AND	"&"
parallel AND	","

It is evident from their names, "&" indicates goals connected by "&" to be executed sequentially while "," means goals connected by "," to be executed in parallel.

[Read-only annotation] The read-only annotation "?" is control information that can be added to occurences of variables and is written as "X?". Variables with "?" can always be unified with other variables but must not be unified with non-variable terms until they are instantiated. The annotation can be added independently to individual occurrences of variables. Normally, individual processes add, or do not add, the annotation to variables shared by them. Once a process has added the annotation to a shared variable, it cannot instantiate the variable, and has to wait until another process (without the annotation) instantiates the variable.

Variables unified with variables with the read-only annotation automatically inherit this property. The annotation is not an operator. Therefore, if X and X? appear in the same clause, they are logically the same except for the unification control information. If X is instantiated to a non-variable term with respect to X?, the annotation looses its effect.

2.2 Reduction

We will discuss how a given goal is reduced to subgoals. In Concurrent Prolog, a program consists solely of guarded clauses. If there is a clause not explicitly containing "|", it is processed as if its guard is empty, that is, it had "|" in the leftmost part of its right side.

Now assume that there is a goal A and the following are the clauses which have the same predicate name as the goal A.

```
A1 :- G1 ¦ B1.
A2 :- G2 ¦ B2.
      .
      .
      .
An :- Gn ¦ Bn.
```

Gi may be empty. These clauses are classified into the following three categories with respect to the goal A.

(1) candidate Ai :- Gi ¦ Bi.

In this case, A and Ai can be unified and Gi solved without unifying a read-only variable with a non-variable term.

(2) suspended Aj :- Gj ¦ Bj.

In this case, A and Aj can be unified and Gj can be solved except when a read-only variable is instantiated to a non-variable term.

(3) fail Ak :- Gk ¦ Bk.

Other cases.

If goal A has one or more candidate clauses, one of them is selected and the goal is reduced to Bi (assuming it is Ai :- Gi ¦ Bi). The selection mechanism evaluates each clause in parallel and selects the first candidate found. Use of this approach permits don't care non-deterministic processing. Once the goal A has been reduced, checks of alternative clauses are aborted. In this sense, the "¦" symbol functions as the cut symbol. If goal A has no candidate clause but has at least one suspended clause, it is suspended until at least one candidate can be found or a complete failure occurs.

No variable binding taking place in the course of the reduction is finalized until the computation commits to that clause and other possibilities are eliminated. Therefore, a shared variable without the read-only annotation, even if it is instantiated to a non-variable term in the course of reduction, does not allow access by other processes until "¦" is passed.

3. The structure of Mandala

The most important concept in Mandala is its multi-plane structure. The bottom plane is used to express object worlds which contain information of given problems domains and is called an object plane. The next higher plane is used to express meta knowledge about objects described in the object plane and is called a meta plane. In theory, it is possible to consider higher level plane such as a meta meta plane and so on.

In Mandala, there are two kinds of components to express the world structure of problems to be solved; one is a knowledge base (program) depicted by the symbol of a disk, and the other is an active process depicted by a circle. These two kinds of components can exist in each plane by giving different roles respectively. The typical structure is to associate an active process called knowledge base manager in the meta level to each knowledge base in the object level as shown in Figure 3. This association is called a manager-of link, which is one of the four kinds of links in Mandala.

The remaining three kinds of links are is_a, part_of and instance_of. An is_a link connects two disks in the same plane and expresses usual concept hierarchy as shown in figure 4.

An instance_of link connects a disk and a process in the same plane and expresses the fact that the process is an instance of the disk, or that it is "executing the program" in the disk.

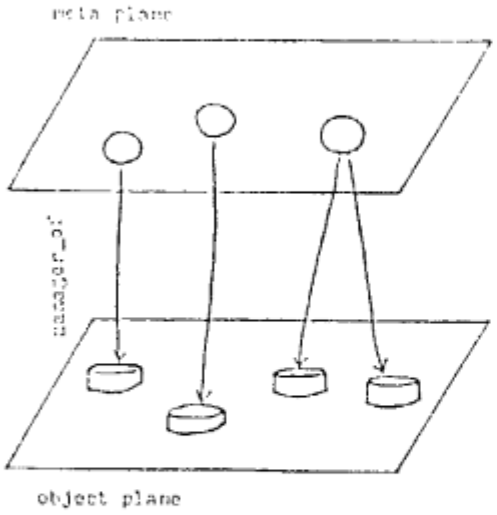

Figure 3. manager_of link

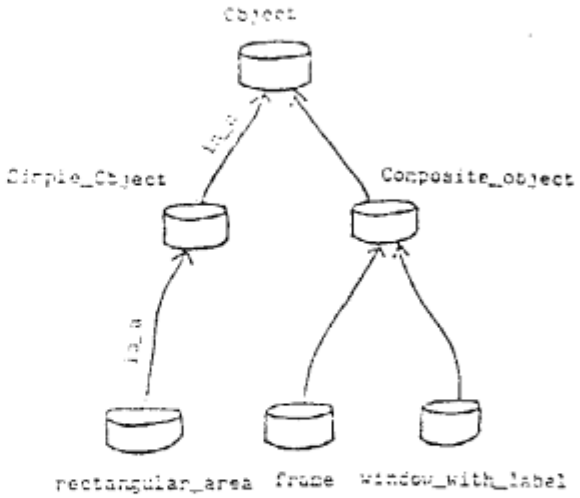

Figure 4. is_a link

A part_of link is similar to an is_a link in a sense that it connects two disks in the same plane. The difference is that there also exists a part_of link between active processes which are respectively connected to the two disks by instance_of links as shown in Figure 5.

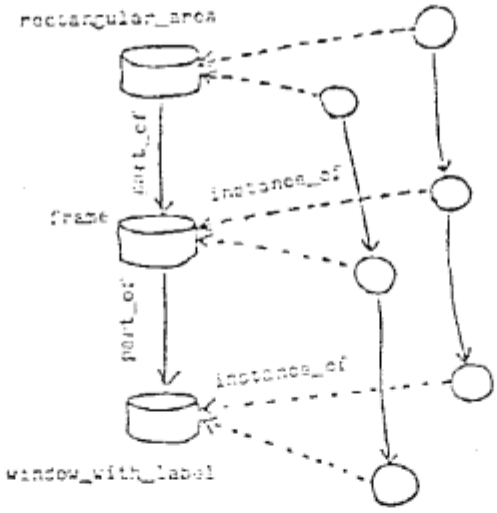

Figure 5. part_of link

Except that it uses "=>" instead of ":-" and additional annotation "+" to goals, the content of each knowledge base is a Concurrent Prolog program which describes the behavior of its instances. On the other hand, the content of each process is its status which changes according to the proceeding of computation.

The distinction between knowledge base (program) and process enables us to introduce complex links between two planes. It is possible to associate more than one disk to each knowledge base manager. Assume a knowledge base manager responsible to check the consistency of a new input to the knowledge base. It uses a different kind of knowledge called integrity constraints to check the input. Integrity constraints are themselves kinds of object knowledge which is to be expressed in the object plane. Therefore, we need two knowledge bases to be associated to the knowledge base manager. Note that if we limit only one to one correspondence for manager_of links, Mandala's structuring concept becomes equivalent to that of Smalltalk-80 [Goldberg & Robson 1983].

4. Implementation in Concurrent Prolog

As already mentioned, there are two basic components in Mandala, that is, a knowledge base (a disk) and a knowledge base manager (a circle). In the current stage, a knowledge base is implemented as a program of Concurrent Prolog and stored in the internal data base as well as another programs. On the other hand, a circle, which is an instance of some knowledge base and plays a role of a knowledge base manager if it is created on the meta plane , is implemented by a process. The goal which represents this process has the form,

```
object(Name,Input,Program).
```

Hereafter we call the process with this goal an object. An object takes three arguments. The first argument "Name" is an identifier of the object, the second argument "Input" is a channel through which the object receives a sequence of goals and the third argument "Program" is a set of clauses which specify the behavior and state of the object as Concurrent Prolog programs. When an object receives a goal through its channel, the object solves it using its own knowledge "Program" and tries next goal while it receives new goals. The Concurrent Prolog program which represents an object is shown below.

```
object(Name,[Goal|Input],Program) :-
        simulate(Name,Goal,Program,NewProgram),
        (wait(NewProgram) & object(Name,Input?,NewProgram)).
object(Name,[],Program).
```

The predicate "simulate" takes four arguments and it tries to solve the goal "Goal" given as the second argument, using local knowledge "Program" also given as the third argument. After the goal "Goal" has been solved it returns the updated program "NewProgram" to the fourth argument. In practice it only invokes "simulate" with five arguments in which the extra fifth argument is used to represent default knowledge which can also be used to solve the goal "Goal". The program of "simulate" with four arguments are shown below.

```
simulate(Name,Goal,Program,NewProgram) :-
    simulate(Name,Goal,Program,NewProgram,[]).
```

The fifth argument of "simulate" with five arguments is default knowledge, which can be used in solution of the goal "Goal" and is represented as a set of clauses. New "simulate" predicate can use both local knowledge and default knowledge when it solves a goal and returns the updated local knowledge. If the goal cannot be solved using local nd default knowledge, it enhances the default knowledge and tries to solve it again with ew default knowledge. The program of "simulate" with five arguments is shown below.

```
simulate(Name,true,W,W,_).
simulate(Name,(+Process,Q),Program,NewProgram,Default) :-
    Process, simulate(Name,Q,Program,NewProgram,Default).
simulate(Name,(A,B),W,N,D) :-
    simulate(Name,A,W,W1,D), (wait(W1)&simulate(Name,B,W1,N,D)).
simulate(Name,(A&B),W,N,D) :-
    simulate(Name,A,W,W1,D)&simulate(Name,B,W1,N,D).
simulate(Name,+Process,Program,Program,Default) :-
    Process.
simulate(Name,process_status(Program),Program,Program,_).
simulate(Name,add(C),Program,[C|Program],_).
simulate(Name,delete(C),Program,NewProgram,Default) :-
    prolog(delete(C,Program,NewProgram,Default)).
simulate(Name,(PName,Class,Chan) part_of Me,Program,Program,D) :-
    prolog(find_axiom((PName,Class,Chan) part_of Me,Program)).
simulate(Name,A,W,W,D) :-
    prolog(system_pred(A)) | prolog(A).
simulate(Name,A,W,N,D) :-
    prolog((find_method(Name,A,W,D,NewD),append(W,NewD,V),copy(V,CW))) &
    simulate_resolve(Name,A,CW,B,W(W,NewD),W1) |
    simulate(Name,B,W1,N,NewD).
```

The meaning of each clause is listed below.

1. If the goal is true, "NewProgram" is the same as "Program".
2. If the goal is a combination of the form (+Process,Q), it forks "Process" and solves the rest of goals by "simulate".
3. If the goal is a combination of the form (P,Q), it splits to two "simulate" processes.
4. If the goal is a combination of the form (P&Q), first it solves "P" and then solves "Q".
5. If the goal is +Process, then the process solves the "Process" without any layer of "simulate".
6. If the goal is "process_status(W)", it unifies the variable "W" with the local program "Program".
7. If the goal add(C), it adds the assertion "C" to the "Program" and makes "NewProgram".
8. If the goal is delete(C), it removes the assertion "C" from the "Program" and make "NewProgram".
9. If the goal is "(P,Cl,Ch) part_of Me", it tries to unify "(P,Cl,Ch) part_of Me" with an assertion in the program.
10. If the goal is a system predicate, the goal is solved by the sequential Prolog interpreter.
11. Otherwise, it tries to reduce the goal using the program and the default.

Below the programs of "simulate_resolved" and "simulate_unify" are shown.

```
simulate_resolve(Name,A,[C|Cs],B,w(W,D),W1) :-
    simulate_unify(A,C,G,B) & simulate(Name,G,W.W1.D) | true.
simulate_resolve(Name,A,[C|Cs].B.WD.W1):-
    simulate_resolve(Name,A,Cs?.B.WD.W1) | true.

simulate_unify(A,(A=>(G|B)),G.B).
simulate_unify(A,(A=>B).true,B).
simulate_unify(A,A,true,true).
```

5. Examples

A simple example which describes a counter is shown below.

```
%---------- Class   Counter ----------

    class(counter).
    counter(counter is_a 'Simple_Object').
    counter(state(0)).
    counter((clear => delete(state(X)) & add(state(0)))).
    counter((up => delete(state(X)) & X1 := X+1 & add(state(X1)))).
    counter((down => delete(state(X)) & X1 := X-1 & add(state(X1)))).
    counter((show => state(X) & write(X) & nl)).
```

Description of a meta class "Class" which is a typical knowledge manager is shown next. "Class" can create instances from a knowledge base and show listing of knowledge base and so on.

```
%---------- MetaClass   Class ----------

    metaclass('Class').
    'Class'('Class' is_a 'Simple_Object').
    'Class'(number(0)).
    'Class'((create(Name,Goals) =>
            delete(number(X)) & X1 := X+1 & add(number(X1)) &
            add(instance(Name,Goals)) & Cname instance_of Mname &
            instantiate(Cname,Name,DW) & +object(Name,[init|Goals],DW))).
    'Class'((how_many => number(X) & write(X) & nl)).
```

```
'Class'({list(db) => C instance_of M &
          program(C,Clauses),writelnl(Clauses),nl)).
'Class'((kill(Name) => delete(number(X)) & X1 := X-1 & add(number(X1)) &
                       delete(instance(Name,_)))).
'Class'((list(self) => process_status(S) & writelnl(S)&nl)).
```

Here we show three basic knowledge bases, 'Object', 'Simple_Object' and 'Composite_Object'. 'Object' is placed on the root of is_a hierarchy spawned on the object plane and 'Composite_Object' is a root node for every objects which consists of more than one object and 'Simple_Object' is a root node for other objects.

```
%---------- Class   Object ----------

     class('Object').
     'Object'('Map'(Set,[])).
     'Object'(('Map'(Set,[X|R]) => copy(Set,S)&S={X|Goals}|
            Goals,'Map'(Set,R))).
     'Object'(('Enumerate'(Set,List) => process_status(W)|get_all(W,Set,List))).

% --------- Class   Simple_Object ----------

     class('Simple_Object').
     'Simple_Object'('Simple_Object' is_a 'Object').
     'Simple_Object'(init).

%---------- Class   Composite_Object ----------

     class('Composite_Object').
     'Composite_Object'('Composite_Object' is_a 'Object').
     'Composite_Object'((send_to(Name,Msg) =>
            delete((Name,Class,[Msg|New]) part_of Me) &
            add((Name,Class,New) part_of Me))).
     'Composite_Object'((init =>
       'Enumerate'({(Name,Class,Chan)|(Name,Class,Chan) part_of _},List),
       'Map'({(Name,Class,Chan)|instantiate(Class,Name,Program) &
                                +object(Name,[init|Chan],Program)},
              List))).
```

Here we show more concrete examples. Below definitions of "rectangular_area", "frame" and "window_with_label" are shown. They are also examples of "part_of" relation, since "window_with_label" is defined using "frame" as a part and frame is also defined using "rectangular_area" as a part.

```
%---------- Class   Rectangular_Area ----------

     class(rectangular_area).
     rectangular_area(rectangular_area is_a 'Simple_Object').
     rectangular_area(state((20,5,30,10))).
     rectangular_area((clear => state(Param) & clear_primitive(Param))).

%---------- Class   Frame ----------

     class(frame).
     frame(frame is_a 'Composite_Object').
     frame((rec,rectangular_area,Chan) part_of frame).
     frame((draw => send_to(rec,state(Param)) & wait(Param) & draw_lines(Param))).
     frame((refresh => send_to(rec,clear) & draw)).
     frame((state(Param) => send_to(rec,state(Param)))).

%---------- Class   Window_with_Label ----------
```

```
class(window_with_label).
window_with_label(window_with_label is_a 'Composite_Object').
window_with_label((fr,frame,Chan) part_of window_with_label).
window_with_label(label(gazonk)).
window_with_label((change(Label) => delete(label(_)) & add(label(Label)))).
window_with_label((show => send_to(fr,refresh) & send_to(fr,state(Param)) &
        wait(Param) & label(Label) & show_label_primitive(Label,Param))).
```

Finally we show a more complex class definition than before. The new class has an assimilator as a part and manages two knowledge bases, one of which is a positive knowledge base that contains programs and the other is a negative knowledge base that contains a set of integrity constraints for the positive knowledge base. When a class receives a new data, the assimilator associated to the class tries to check if the new data invokes a contradiction and if it is a redundant information and so on. The list is given below.

```
%---------- MetaClass 'Class' with assimilator ----------

metaclass('Class').

'Class'('Class' is_a 'Composite_Object').
'Class'((assim,assimilator,Ch1) part_of 'Class').
'Class'((pinput(Assertion) =>
         Me instance_of Mname &
         P positive_kb_of Me  &  N negative_kb_of Me &
         send_to(assim,passimilate(Assertion,P,N)))).
'Class'((ninput(Assertion) =>
         Me instance_of Mname &
         P positive_kb_of Me  &  N negative_kb_of Me &
         send_to(assim,nassimilate(Assertion,P,N)))).

assimilator(assimilator is_a 'Composite_Object').
assimilator((contra,contradiction_checker,Ch2) part_of assimilator).
assimilator((redun,redundancy_checker,Ch3) part_of assimilator).
assimilator((passimilate(Assertion,P,N) =>
             program(P,Pprogram) & simulate(Pprogram,Assertion) ¦ true)).
assimilator((passimilate(Assertion,P,N) =>
             program(P,Pprogram) & program(N,Nprogram) &
             send_to(contra,contradict(Pprogram+Assertion,Nprogram))
             ¦ true)).
assimilator((passimilate(Assertion,P,N) =>
             program(P,Pprogram) &
             send_to(redun,redundant([],Pprogram+Assertion,InterPprogram))
             ¦ modify(P,InterPprogram) &
               passimilate(Assertion,P,N))).
assimilator((passimilate(Assertion,P,N) =>
             otherwise ¦ program(P,Pprogram) & modify(P,Pprogram+Assertion))).

contradiction_checker(contradiction_checker is_a 'Simple_Object').
contradiction_checker((contradict(InterPcls,[Ncl¦Ncls]) =>
                       simulate(InterPcls,Ncl) ¦ true)).
contradiction_checker((contradict(InterPcls,[_¦Ncls]) =>
                       otherwise ¦ contradict(InterPcls,Ncls))).

redundancy_checker(redundancy_checker is_a 'Simple_Object').
redundancy_checker((redundant(SeenPcls,[Pcl¦Pcls],NewPcls) =>
                    simulate(SeenPcls+Pcls,Pcl) ¦ true)).
redundancy_checker((redundant(SeenPcls,[Pcl¦Pcls],NewPcls) =>
                    otherwise ¦ redundant([Pcl¦SeenPcls],Pcls,NewPcls))).
```

6. Acknowledgement

The authors wish to thank members of ICOT working group No.2 and No.4 both for fruitful discussion and comments. The authors would also like to thank Kazuhiro Fuchi, Director of ICOT Research Center and all the other members of ICOT, both for help with this research and for providing a stimulating place in which to work.

[References]

[Shapiro 1983] E.Y.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003 (1983).

[Clark & Gregory 1981] K.L.Clark, S.Gregory: A Relational Language for Parallel Programming, Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (1981).

[Clark & Gregory 1983] K.L.Clark, S.Gregory: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, Imperial College, May (1983).

[Goldberg & Robson 1983] A.Goldberg, D.Robson: Smalltalk-80: The language and its implementation, Addison-Wesly 1983.

[Bobrow & Stefik 1983] D.G.Bobrow, M.Stefik: The LOOPS Manual, Xerox technical memo KB-VLSI-81-13, 1983.

[Shapiro & Takeuchi 1983] E.Shapiro, A.Takeuchi: Object Oriented Programming in Concurrent Prolog, New Generation Computing Vol.1, No.1 (1983)

[Takeuchi & Furukawa 1983] A.Takeuchi, K.Furukawa: Interprocess Communication in Concurrent Prolog, Proc. of Logic Programming Workshop '83 in portugal (1983).