171

Interprocess Communication in Concurrent Prolog

Akikazu Takeuchi Kouichi Furukawa
Research Center
Institute for New Generation Computer Technology
Mita-Kokusai Building, 21F.
4-28, Mita 1-chome, Minato-ku, Tokyo 108
Japan

Abstract

Concurrent Prolog is a logic-based concurrent programming language which was designed and implemented on DEC-10 Prolog by E. Shapiro. In this paper, we show that the parallel computation in Concurrent Prolog is expressed in terms of message passings among distributed activities and that the language can describe parallel phenomena in the same way as Actor-formalism does. Then we examine the expressive power of communication mechanism based on shared logical variables and show that the language can express both unbounded buffer and bounded buffer stream communication only by read-only annotation and shared logical variables. Finally the new feature of Concurrent Prolog is presented, which will be very useful in describing the dynamic formation and reformation of communication network.

1. Introduction

Concurrent Prolog was designed and implemented on the DEC-10 Prolog by E. Shapiro [1] for concurrent programming. As the Relational Language [2], Concurrent Prolog adopts Or-parallelism as a basis for non-deterministic processing, and And-parallelism for description for parallel processes. Shared variables are used, with some control information "variable annotation", as communication channels among concurrent processes.

In the Relational Language, there are two kinds of the variable annotation, input and output, which are used for input suspension and output suspension respectively. On the contrary, in Concurrent Prolog, there is only one annotation, read-only annotation, which is a generalized idea of input annotation and by which we can also express the output suspension when an output buffer is full. This will be explained in the section 4.

In the section 2, we review the Concurrent Prolog. In the section 3, the computation model of the language is presented and in the section 4 we examine the basic communication mechanism based on shared logical variables and derive the technique for implementing the bounded buffer communication in the language. In the section 5, we introduce the concept of the incomplete message as a new programming paradigm and explain briefly. In the section 6, we present a new feature of Concurrent Prolog which is very useful in describing the formation and reformation of the communication network.

2. Review of the Concurrent Prolog

2.1 Syntax of Concurrent Prolog

In Concurrent Prolog, a program is represented as a list of guarded clauses. The form of a guarded clause is

A :- G1,...,Gn | B1,...,Bm.   n,m >= 0.

A guarded clause must have a guard bar "|". The left hand side of guard bar is called the guard sequence and the right hand side is called the goal sequence. The guard bar can be omitted when the guard sequence is empty, that is n=0. G's and B's are both lists of literals connected by logical AND.

There are two kinds of logical AND's, which are parallel-AND and serial-AND.

serial-AND	"&"
parallel-AND	","

Their logical meaning are the same, but the way to interpret and execute is different. As it is clear from their name, goals connected by serial-AND must be executed in sequential order (left-to-right), and goals connected by parallel-AND must be executed in parallel. As for the operator precedence,

"," is lower than "&", that is,

f&g , p&q is equivalent to (f&g) , (p&q).

Current implementation of Concurrent Prolog only provides sequential-or mode. Therefore, alternative clauses are tried in the text order.

On the notation, we adopt DEC-10 Prolog-like convention, for example, a word beginning with a capital letter denotes a variable.

In Concurrent Prolog, variables can be accompanied with some special control information, "read-only" annotation, which can control the unification. Read-only annotation is denoted by "?" and can be attached to variables in the following way,

X?   where X is a variable.

The meaning of read-only annotation is that a variable occurrence annotated by "?" must not be unified with a non-variable term. The annotation can be attached to ~~each~~ any occurence of a variable, and will vanish when the variable will be instantiated to a non-variable term. Generally read-only annotation can be attached to the variables shared by concurrent processes in order to restrict the direction of data flow, where the process which annotates the shared variable can not instantiate the variable and wait for the variable to become instantiated by the other process which does not annotate it. This will be explained later again.

2.2 Reduction

In this section, the process of reduction is explained. Suppose that the goal A and the following program are given.

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

where Gi and Bi (1 =< i =< n) are a guard sequence and a goal sequence respectively.

Each clause is classified into one of the three following classes with respect to the goal A.

1. Candidate   Ai :- Gi | Bi.

   when, without instantiating variables annotated by "?" to non-variable terms, A and Ai can be unified and Gi can be solved.

2. Suspended   Aj :- Gj | Bj.

when, except for instantiating read-only variables to non-variable terms, A and Aj can be unified and Gj can be solved.

3. Failure
otherwise.

Each clause is checked in ~~a~~ text order whether it can be a candidate, and the clause that is found to be a candidate first is selected. The selected clause, say A :- Gi|Bi., is used to reduce the goal to the goal sequence Bi. Once the goal is reduced, checking of the rest of clauses will be abandoned. In this sense, the guard bar "|" acts as a cut symbol.

When the goal has no candidate and has at least one suspended clause, it will be suspended until at least one candidate will be found or it will be failed (i.e. all the clauses will be classified into the failure).

Since the instantiation of shared variables can be undo~~ed~~ne by ~~the~~ backtracking before the guard sequence is solved completely, the values of the shared variables will be hidden from other processes until the guard sequence is solved completely.

Although Concurrent Prolog adopts And-parallelism, consistency check of values of shared variables will be replaced by the restriction that the process instantiating the shared variables must be one. However, which process can instantiate a shared variable need not be specified before the execution, as long as it is guaranteed that there can be only one such process even if it is determined dynamically in a non-deterministic way.

3. The Computation Model

In this section we present the Actor-like model [3,4] of the parallel computation in Concurrent Prolog. For the simplicity, we assume that all ~~every~~ goals are solved in Or-parallel mode, that is, all the alternatives are checked in parallel.

First we define the term "event" which is a basic concept in order to formalize the computation model.

"An event is a successful unification between a goal and a head of a clause and a successful solution of the guard sequence of that clause."

Using this definition, we can specify the condition for an event to arise.

"The condition for an event associated with a goal to arise is that the goal can be unified with a head of some clause and its guard sequence can be solved successfully."

Given a goal A and a clause A' :- G1,...,Gn|B1,...,Bm, we denote the event by

A:A'.

Once a goal A is unified with the head A' of a clause

A' :- G1,...,Gn|B1,... Bm.

that is, the event A:A' happens, then A is reduced to the goal sequence B1,...,Bm which in turn begin to invoke other events, say B1:B1',...,Bn:Bn'. In this way, generally an event causes other events except the case in which a goal is unified with a clause with empty goal sequence, in this case the event causes nothing.

Let's define the causal relation among events more precisely.

"An event E, A:A', causes an event E', B:B',
if and only if B is included in the set

{ Bi | 1 =< i =< n }

where A' is a head of the clause

A' :- | B1,...,Bn. "

It is clear from the definition of an event that there can be no circular causal relation among events.

We denote the causal relation "E causes E'" by

E => E'

Generally an event causes more than one events.

$$\begin{tikzcd}
 & E1 \\
E \arrow[ur, Rightarrow] \arrow[r, Rightarrow] \arrow[dr, Rightarrow] & E2 \\
 & E3
\end{tikzcd}$$

The reflexive transitive closure of the causal relation => is denoted by ==>. By the relation ==>, an event E1 can be related to the event E2 indirectly caused by the event E1. For example, E1 => E2, E2 => E3 then E1 ==> E3 and so on.

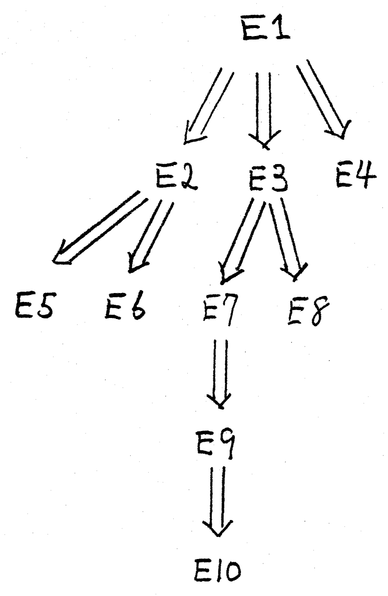
Note that the relation ==> also can be interpreted as the semi-order relation of an activation of an event. "E1 ==> E2" can be read as that an activation of an event E1 precedes an activation of an event E2.

Now we define the term "process".

"A process initiated by an event E is a chain of events
connected by the relation =>."

Given a goal A and a clause A' :- Gl,...,Gn|Bl,...,Bm., a process initiated by the event A:A' can be thought as the solution process of the goal A using that clause. From this point view, it is clear that the time when a process terminates is the time when the goal A is solved completely.

Since an event can cause more than one event, the chain of events (= process) looks like a tree (see figure).



The terminal nodes of the tree correspond to the events each of which is a unification between a goal and a clause with an empty goal sequence.

4. Interprocess communication

In Concurrent Prolog, interprocess communication is realized by variables logically shared among processes. A process can send a message to other processes by instantiating a variable shared among them to the message. Since a destructive assignment to a logical variable is not permitted, communication using one variable cannot be done more than once. However, in general, because there is no restriction about the number of the processes sharing a variable, the message to which one of the processes instantiates the shared variable will be sent to the rest of processes at the same time. Therefore broadcasting of a message has been realized without any additional mechanism.

Shared variables are created when, for example, a process forks to subprocesses.

```
p(X) :- | q(X,Y),r(Y?).
```

In the example above, the variable Y is shared between the processes, which are solution processes of the goal q and r respectively, and is used for communication between them.

However, as mentioned above, communication using one shared variable cannot be done more than once. Therefore in order to enable the successive communication among processes, there must be some mechanism to create a new logically shared variable dynamically. Most general method for this is the technique of the stream communication which is well known by the work of Clark and Gregory [2].

In the stream communication, a shared variable is instantiated to a data structure which contains a message and a new uninstantiated variable. In the Relational Language, a list was used for such structure.

```
[<message>|<variable>].
```

A variable contained in the structure is sent with a message from the sender to the receivers, becomes a new shared variable among processes and will be used for the next communication. Consequently as long as a process sends a message in this way, every time a message is sent, a new shared variable is created, so that the successive communication is established.

In general, the successive communication consists of two phases.

Phase 1	A shared variable is instantiated to a message.
Phase 2	A new shared variable is created.

In the phase 1, the action most essential to communication is performed. In the phase 2, what enables a next communication is performed. In the case of the stream communication, both phases are performed at the same time in the same process, the sender. However there is no reason for two phases to be performed in the same process and no restriction on the execution order between the phase 1 and the phase 2. If we treat the two phases separately, we will be able to find several kinds of communication style based only on logically shared variables and read-only annotation. As an example, we present in this section the bounded buffer communication based on shared logical variables, which is implemented without introducing another annotation like the Relational Language. Before that, we summarize the unbounded buffer stream communication.

[The Unbounded Buffer Communication]

In the stream communication, both phases are performed at the same time in the sender of messages by instantiating a shared variable to a pair of a message and a variable. Therefore every time a sender sends a message, it gets a new "shared" variable, so that it can send a next message as soon as it sends a message. On the contrary, a receiver can read a message only after it is received and the receiver has to wait when it tries to read a message and no message is received yet. This "wait" mechanism is implemented by making the shared variable in the receiver read-only. Because there is no mechanism for inhibiting the sender to send a message, this type of communication realizes the unbounded buffer communication. Note that the essence of unbounded buffer communication is in the fact that both phases are performed in the same process, the sender of messages.

As an example of the stream communication we show the program which describes the situation where there are two communicating processes, one of which sends an integer every time the process generates it and the other prints out an integer every time the process receives it.

```
Goal:: integers(0,N) , outstream(N?).

Program-1 ::
  integers(I,[I|N]) :-                  ....   [send]
        plus(I,1,J) | integers(J,N).
  outstream([I|N]) :-                  .... [receive]
        write(I) | outstream(N?).
```

Note that "outstream" will be suspended when the variable N is not instantiated to a non-variable term, because of the condition for read-only variables. In the example above, message sendings and receivings are processed at the unification between a goal and a head of a clause. We could write the same program in more abstract level like below.

```
Goal:: integers(0,N) , outstream(N).

Program-2 ::
  integers(I,N) :-
        send(I,N,M), plus(I,1,J) | integers(J,M).
  outstream(N) :-
        receive(I,N?,M), write(I) | outstream(M).
```

In both predicates "send" and "receive", the first argument is a message, the second argument is a current communication variable and the third argument is a next communication variable. The program "send" and "receive" are:

```
send(X,[X|M],M).
receive(X,[X|M],M).
```

The advantages in using "send" and "receive" are to hide the internal structure to which the shared variable is instantiated and to modularize programs. In fact, even if we could use another data structures, say "stream(<message>,<variable>)", instead of the list "[<message>|<variable>]", the programs which have to be changed are only "send" and "receive" (new codes are shown below) and no other programs including the user programs are kept unchanged.

```
send(I,stream(I,M),M).
receive(I,stream(I,M),M).
```

On the other hand, we could say that using "send" and "receive" is to lose the simplicity of the Program-1.

[The Bounded Buffer Communication]

In the bounded buffer communication, to send a message is suppressed when messages, the number of which is equal to the size of the buffer, are kept unread in the buffer of the receiver.

From the above analysis of communication through shared variables, we can naturally find the mechanism for this kind of communication. The key idea is the separation of the actors of two phases.

The phase 1 (instantiation) is performed by the sender at the moment it send a message and the phase 2 is performed by the receiver when and only when it reads (picks up) a message from the buffer. Therefore the sender cannot send messages more than the buffer size if the receiver did not read the messages, that is, it did not generate new shared variables.

We explain the method when the buffer size is equal to two, using the previous example.

```
Goal:: integers(0,[X,Y|Z]) , outstream([X,Y|Z]).

Program::
  integers(I,N) :-
        send(I,N?,M), plus(I,1,J) | integers(J,M).
  outstream(N) :-
        receive(I,N,M), write(I) | outstream(M).
```

Note that the second argument of "send" is annotated as read-only, while in the previous example the second argument of "receive" is annotated as such. The following is a new code for "send" and "receive" programs in the bounded buffer communication.

```
send(Msg,[Msg|NewChannel],NewChannel).
receive(Msg,[Msg|NewChannel],NewChannel) :-
          wait(Msg)|update_buff(NewChannel).
```

Here again we use the list structure for implementing the stream. "wait(X)" is a system predicate which suspends when the argument "X" is not instantiated yet, and succeeds otherwise. "update_buff(X)" is a sequential Prolog program which takes a d-list as an argument and instantiates the tail variable of it to a cons cell "[P|Q]" where both "P" and "Q" are uninstantiated variables.

```
update_buff(X) :- var(X),!,X=[P|Q].
update_buff([X|Y]) :- update_buff(Y).
```

The second argument of "receive" plays a role of a buffer consisting of slots (variables) which will be filled with messages by the sender. The buffer is updated by one slot when and only when the receiver picks up a message from the buffer, so that the length of the d-list (buffer) remains the same which corresponds to the buffer size. Although the sender shares the buffer with the receiver, it can not update the buffer and all it can do is to fill empty slots with messages if there is any such slot. When the size of buffer is equal to two, the buffer looks like:

[X,Y|Z].

For the sender, the buffer looks like one of the following.

(1) [X,Y|Z]
(2) [Y|Z]
(3) Z

where "X", "Y" and "Z" are all uninstantiated variables. (1) corresponds the case in which the buffer is empty, that is, there is two empty slots and (2) corresponds to the case in which there is one room for sending a message. (3) corresponds to the case in which the buffer is full, that is, there is no room for sending a message. Because the second argument of "send" is treated as read-only, the reduction of "send" is suspended in the case (3). The figure below shows the situation where the sender tries to send three messages, "ab", "cd" and "ef" when the buffer is empty.

```
            the receiver          the sender
            [X,Y|Z]               [X,Y|Z]

                               — send "ab" —

            [ab,Y|Z]              [Y|Z]

                               — send "cd" —

            [ab,cd|Z]             Z

                               — send "ef" is suspended
        — receive "ab" —

            [cd,P|Q]              [P|Q]

                               — send "ef" —
```

It is more convenient when we could parameterize the size of the buffer. Generally their usage are the following.

```
In sender   :: send(Msg,Channel?,NewChannel)

In receiver  At the first communication
             :: open(Channel,N),
                receive(Msg,Channel,NewChannel)

               At the subsequent communications
                receive(Msg,Channel,NewChannel)
```

"open" takes two arguments, a communication variable "Channel" and a size of a buffer "N", and it instantiates the variable "Channel" to the d-list with the first "N" arguments of it instantiated to variables. "open" is also a sequential Prolog program.

```
open(X,0) :- !.
open([X|Y],N) :- N1 is N-1,open(Y,N1).
```

The program above specifies the case in which the buffer size is more than or equal to one. Implementation of 0-Buffer communication is a little different from the above. The predicate "receive" is replaced by the following definition.

```
receive(Msg,[Msg|New],New).
```

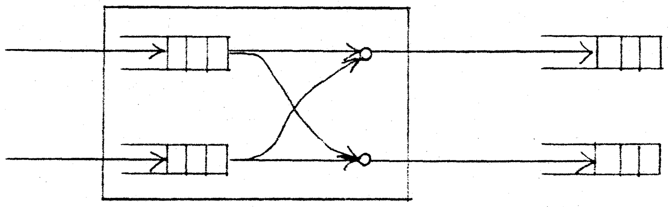
and their usage becomes:

```
In sender    :: same as above
In receiver  :: receive(Msg,Channel,NewChannel),wait(Msg)
```

The bounded buffer communication is very important when there are several processes, each of which produces or consumes data in different speed. Suppose that, in the example above, the rate of integer generation in "integers" is much greater than that of data consumption in "outstream", in such case if we use the unbounded buffer communication between two processes, the huge amount of unprocessed integers will be produced. The bounded buffer communication is a simple and efficient method to control and combine processes having different rate of data producing or consuming by controlling the production of data according to the consumption of them.

As an example of the application of this bounded buffer communication, we can define a 2 x 2 communication switch which has two input ports and two output ports. It can receive inputs from two ports and sends them to the output port which has at least one empty slot. If both ports are not available, the "switch" is suspended.

```
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In1,Ins1)&send(M,Out1,Outs1) | switch2x2(Ins1,In2,Outs1,Out2).
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In2,Ins2)&send(M,Out1,Outs1) | switch2x2(In1,Ins2,Outs1,Out2).
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In1,Ins1)&send(M,Out2,Outs2) | switch2x2(Ins1,In2,Out1,Outs2).
switch2x2(In1,In2,Out1,Out2) :-
     receive(M,In2,Ins2)&send(M,Out2,Outs2) | switch2x2(In1,Ins2,Out1,Outs2).
```



5. Incomplete Message

As in the actor formalism, Concurrent Prolog is a model of the parallel computation and provides a communication methods through shared variables. A message will be sent by instantiating the shared variables. A message which contains a variable is called an incomplete message [5]. It makes a new variable shared by the sender and the receiver of the

message, that is, it creates a new communication channel. It means that a communication channel can be made dynamically and it can be sent to other processes also.

The concept of an incomplete message is a large programming paradigm which includes the basic communication mechanism between processes, so-called pipeline processing on stream data, and yields new features of Concurrent Prolog. The close analysis of this concept is described in the paper of Shapiro and Takeuchi [5].

In this section, we review the key features of this concept according to the paper of Shapiro and Takeuchi [5].

(1) [Stream] Once a variable is instantiated, it will never be rewritten except the case where the whole goals fail. Therefore it can not be used as a communication channel in the next message passing phase. In order to enable subsequent communication, in the stream communication generally a shared variable is instantiated to a list of a message and a variable which will be used in a next communication. In this sense, the stream communication is one of the examples of incomplete messages and provides a basic communication mechanism in Concurrent Prolog.

(2) [Pipeline] In addition, incomplete messages make it possible to process partially obtained data in a pipeline style. Although pipeline processing on stream data is a new concept of programming languages, it is included naturally in the paradigm of the partially defined message. In some sense, usual message passing can be seen as a kind of pipeline processing on a sequence of commands generated incrementally.

(3) [Response] When a process sends a message which requires a response, the response can not be sent through the same shared variable, since logical variables are single-assignment. The technique of the incomplete messages is also useful in this case, in which the sender sends a message that contains an uninstantiated variable, and then examines that variable in a read-only mode, which causes it to suspend until this variable gets instantiated to the response by the recipient of the

message. However this is different from the examples above, because the process which instantiates a shared variable is the receiver of the message. In this case, once a message is sent to a process, the sender can run independently whether the receiver returns the response as long as the sender need not to refer to the response. When the sender needs the response it is forced to wait until it will be instantiated. This behavior associated to a shared variable used in a response takes an advantage in writing a monitor of shared resources and highly reduces the overhead on the resource manager because the manager will never be locked and the request will never be refused.

6. New Features of Concurrent Prolog

In this section, we explain the another feature of the Concurrent Prolog not available in other concurrent programming languages.

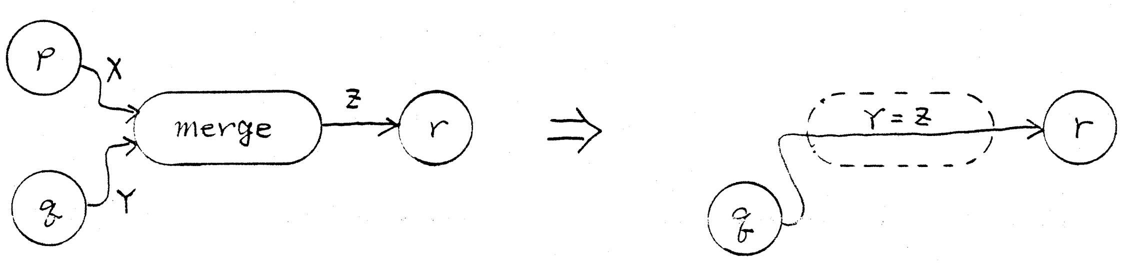
The interprocess communication based on shared variables is not new method and has been implemented generally by sharing physical memory cells. The difference between the communication by the shared variables of Concurrent Prolog and that of traditional languages is the highly abstracted level of shared object. In traditional languages, the objects shared are physical objects such as memory cells or global variables. On the contrary, in Concurrent Prolog, the objects shared are highly abstracted logical variables which can be objects of the unification operation, a very high level operation. Because of this high level abstraction, Concurrent Prolog can express very high level communication style among parallel processes in a simple way, that is, unifying two communication channels.

The well-known "merge" program is an example of this feature.

```
merge([A|X],Y,[A|Z]) :- | merge(X?,Y,Z).
merge(X,[A|Y],[A|Z]) :- | merge(X,Y?,Z).
merge([],Y,Y).
merge(X,[],X).

Goal:: p(X),q(Y),merge(X?,Y?,Z),r(Z?)
```

This program merges two input streams into one stream. The first two clauses are used for this purpose. The rest two clauses describe the situation, where one of the input stream (say "X") reached the end, and the remaining stream ("Y") is unified with the output stream ("Z"). After this unification, data on the remaining stream ("Y") are sent to the output stream ("Z") without any relay, because the input stream and the output stream are logically the same. The important point is that this change of the data flow can be performed only by the unification and that both the sender and the receiver never know the change of data flow (Figure).

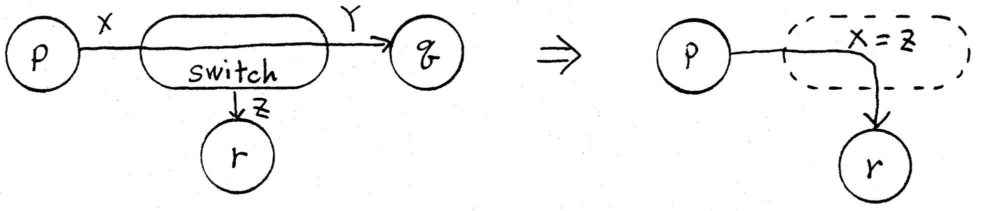


The next program shows another example.

```
switch([on|X],[],X).
switch([A|X],[A|Y],Z) :- | switch(X?,Y,Z).

Goal:: p(X), switch(X?,Y,Z), q(Y), r(Z).
```

"switch" takes three arguments. The first argument is the input stream and the second and the third are the output streams. "switch" program keeps the connection between the input stream and the second argument until it will find the "on" message in the input stream. When "switch" receives it, it changes the connection and thereafter it will pass input data to the third argument. Here again the important point is that the the data flow can be changed directly by the unification and it is hidden from both the sender and receivers (figure).



These two examples demonstrate the new feature of interprocess communication in Concurrent Prolog. Other powerful examples are presented in the paper [5].

7. Conclusion

In this paper we present the computation model of Concurrent Prolog and explain mainly the interprocess communication based on the shared logical variables. 1) From the close analysis of the stream communication, we derived the mechanism for implementing the bounded buffer communication only by the read-only annotation. 2) We have shown briefly the basic programming paradigm "incomplete messages" as a source of the powerful programming technique. 3) We have shown the new features of Concurrent Prolog programming which originate from the logical power of the unification.

8. Acknowledgement

We thank E. Shapiro for his many helpful insights and discussion. We 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.

9. References

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

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

[3] C.Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence 8 (1977).

[4] S.A.Ward, R.H. Halstead: A Syntactic Theory of Message Passing, JACM Vol.27, No.2 (1980)

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