New Generation Computing, 1 (1983) 25-48
OHMSHA, LTD. and Springer-Verlag

Object Oriented Programming in Concurrent Prolog*

Ehud SHAPIRO
Department of Applied Mathematics,
Weizmann Institute of Science,
Rehovot 76100, Israel
Akikazu TAKEUCHI
ICOT Research Center
Institute for New Generation Computer Technology,
Mita-Kokusai Bldg. 21F, 4-28 Mita 1-chome, Minato-ku, Tokyo 108

Abstract It is shown that the basic operations of object-oriented programming languages — creating an object, sending and receiving messages, modifying an object's state, and forming class-superclass hierarchies — can be implemented naturally in Concurrent Prolog. In addition, a new object-oriented programming paradigm, called incomplete messages, is presented. This paradigm subsumes stream communication, and greatly simplifies the complexity of programs defining communication networks and protocols for managing shared resources. Several interesting programs are presented, including a multiple-window manager. All programs have been developed and tested using the Concurrent Prolog interpreter described in.[1)]

§ 1 Introduction

Concurrent Prolog[1)] introduces an operational semantics of parallel execution to logic programs, thus allowing them to express concurrent computations. Concurrent Prolog can specify process creation, termination, communication, synchronization, and indeterminacy. This paper focuses on the object-oriented aspects of Concurrent Prolog. It is shown that the language lends itself naturally to the programming idioms and techniques of Actors[2)] and Smalltalk.[3)]

The paper is structured as follows. Section 2 reviews Concurrent Prolog.

* Part of this research was carried out while Ehud Shapiro was visiting ICOT, the Institute for New Generation Computer Technology. Ehud Shapiro is a recipient of the Sir Charles Clore Fellowship.

Section 3 surveys the elements of object-oriented logic programming. Section 4 studies in detail a non-trivial Concurrent Prolog program : a multiple-window system. The system is operational on the DECSYSTEM-20 and VAX-11 for a VT100 terminal. Section 5 compares traditional object-oriented programming to object-oriented logic programming, and identifies two important programming techniques not easily available in the former : incomplete messages, and constraint propagation.

§ 2 Concurrent Prolog

An examination of the abstract computation model of logic programs suggests that they are readily amenable to parallel execution. A computation of a logic program amounts to the construction of a proof to a goal statement from the axioms in the program. The search space for a proof can be described by an And-Or tree, where an And-node corresponds to a conjunctive goal, and an Or-node corresponds to the different way to reduce a unit goal, using axioms in the program.

An abstract logic program interpreter searching an And-Or tree is assumed to make the **correct** non-deterministic choices at the Or-nodes and can traverse the And-nodes in an arbitrary order. The sequential Prolog interpreter, on the other hand, traverses And-Or trees in depth-first, left-to-right order : conjunctive goals are reduced from left to right, and if there are several alternative ways to reduce a unit goal, they are tried one by one, using backtracking.

One may attempt to search the And-Or tree in parallel, and two forms of parallelism are possible : Or-parallelism and And-parallelism. In Or-parallel execution several alternatives to reduce a unit goal are tried in parallel. In And-parallel execution the goals in a conjunction are reduced in parallel. Since goals in a conjunction may have logical variables in common, the processes attempting to prove each of the conjuncts are not independent and may interfere with each other by instantiating shared variables to incompatible (non-unifiable) solutions. Because of this dependency, one needs some means to coordinate the computations of And-parallel processes.

However, concurrent programming is more than attempting to parallelize the execution of code that can run sequentially : it must have the ability to respond in real-time to multiple events that occur concurrently. The emphasis on the declarative reading of logic programs in the past might have suggested that this formalism will be of no use to real-time application, such as the implementation of an operating system. Nevertheless, the contrary is suggested in the following.

Logic programming was founded on the dual reading of definite clauses. A definite clause

$$A :- B1, B2, \ldots, Bn. \qquad n \geq 0.$$

reads declaratively : A is true if $B1$ and $B2$ and ... and Bn are true. Kowalski's seminal paper[4] suggested a second reading to definite clauses, the procedural,

or problem reduction reading: to execute the procedure call *A*, perform the procedure calls *B1* and *B2* and ... and *Bn*, or: to solve problem *A*, solve the subproblems *B1* and *B2* and ... and *Bn*. In the procedural reading, a unit goal is analogous to a procedure call, both in the way it is used and in the way it is implemented.

Concurrent Prolog[1] and its predecessors, the Relational Language of Clark and Gregory[5] and the language of van Emden and de Lucena,[6] employ a third reading of logic programs: the behavioral reading.

In the behavioral reading, a unit goal is analogous to a process, a conjunctive goal is analogous to a system of processes, and variables shared between goals function similarly to communication channels. A definite clause is read behaviorally: a process *A* can replace itself by the system of processes that contain *B1* and *B2* and ... and *Bn*. A process terminates by replacing itself with the empty system.

In the procedural reading, unification provides a mechanism for parameter passing, variable assignment, and data access and construction. In the behavioral reading it also provides a mechanism for message sending and receiving and an easy way of specifying the different actions to be taken upon the receipt of different messages.

In the behavioral reading, the actions a process can take are specified by the definite clauses in the program: all a process can do is to reduce itself to other processes. In the course of this reduction, variables shared with other processes may get instantiated via the unification of the process with the head of the reducing clause, thus achieving the effect of process communication.

To support process synchronization, Concurrent Prolog introduces a new syntactic construct, called read-only variables. Variables in a process can be annotated as read-only. A process suspends if every reduction of it requires the instantiation of a read-only variable.

Another construct in Concurrent Prolog borrowed from the Relational Language, the guarded-clause, is similar to Dijkstra's guarded-command in its effect. Together with the read-only annotations, guarded-clauses can specify a wide-range of indeterminate process behaviors.

The subset of Concurrent Prolog described and used in this paper was implemented in Prolog-10 on DECSYSTEM-20 and is described in[1], that paper also includes a full listing of the interpreter. With minor modifications, that interpreter can run in Pereira's CProlog on the VAX, under Unix and VMS. A listing of a Concurrent Prolog interpreter written in Waterloo Prolog for IBM / VM computers is available from the first author upon request.

The rest of this section provides a more detailed description of this subset of Concurrent Prolog.

2.1 Syntax

A Concurrent Prolog program is a finite set of guarded-clauses. A

guarded-clause is a universally quantified logical axiom of the form

$$A :- G1, G2, \ldots, Gm \mid B1, B2, \ldots Bn. \qquad m,n \geq 0.$$

where the G's and the B's are atomic formulas, also called unit goals. A is called the clause's head, the G's are called its guard, and the B's its body. When the guard is empty the commit operator " | " may be omitted. Clauses may contain variables marked read-only, such as $X?$. The Prolog-10 syntactic conventions are followed : constants begin with a lower-case letter, and variables with an upper-case letter. The special binary term $[X \mid Y]$ is used to denote the list whose head (car) is X and tail (cdr) is Y. The constant [ ] denotes an empty list.

2.2 Semantics

Concerning the declarative semantics of a guarded clause, the commit operator reads like a conjunction : A is implied by the G's and the B's. The read-only annotations can be ignored in the declarative reading.

Procedurally, a guarded-clause functions similarly to an alternative in a guarded-command. To reduce a process A using a clause $A1 :- G \mid B$, unify A with $A1$, and, if successful, recursively reduce G to the empty system, and, if successful, commit to that clause, and, if successful, reduce A to B.

The reduction of a process may suspend or fail during almost any of these steps. The unification of the process against the head of the clause suspends if it requires the instantiation of variables occurring as read-only in A. It fails if A and $A1$ are not unifiable. The computation of the guard system G suspends if any of the processes in it suspends, and fails if any of them fails.

The commitment operation is the most delicate, and grasping it fully is not required for the understanding of the example programs in this paper. It suffices to say that partial results computed by the first two steps of the reduction — unifying the process against the head of the clause and solving the guard — are not accessible to other processes in A's system prior to the commitment, and that after commitment all the Or-parallel attempts to reduce A using other clauses are abandoned.

The reduction of all processes in a system can be attempted in parallel, as can the search for a clause to reduce a process. Two restrictions prevent an all-out parallelism. Regarding Or-parallelism, only the guards are executed in parallel. Once a guard system terminates, the computations of other Or-parallel guards are aborted. Regarding And-parallelism, read-only annotations can enforce rather severe constraints on the order and pace in which processes can be reduced, as the example programs below show.

This completes the description of the subset of Concurrent Prolog used in this paper. One additional construct — otherwise — is introduced in Section 3. Our Concurrent Prolog implementation supports also the use of the system predicates of the underlying sequential Prolog, including arithmetic and external I / O.

It is worth mentioning that our Concurrent Prolog interpreter is more of

a toy than a real implementation, since it is about 100 to 200 times slower than the underlying sequential Prolog implementation. It runs at approximately the same speed, 130 reductions per CPU second (LIPS), on a DECSYSTEM 2060 running Prolog-10 and on an IBM 4341 running Waterloo Prolog.

§ 3 Object Oriented Programming in Concurrent Prolog

Concurrent Prolog is capable of expressing modern programming concepts, including object-oriented programming. The concept of objects in Concurrent Prolog has close resemblance to that of Actor systems,[2] in that a computation is performed via the cooperation of distributed objects. First a general scheme for object-oriented programming in Concurrent Prolog is presented. It is then explained how objects can be created, and how they can cooperate in computation. A Concurrent Prolog programming technique, called filters, is then introduced, which achieves the effect of hierarchical definition of objects and property inheritance, a useful tool in other object-oriented programming languages. In addition to the usual object oriented features, Concurrent Prolog can provide new features that originate from the logical power of unification. One of them is computation by incomplete messages, and the other is implicit activation of objects, which is similar to a constraint network.[7,8,9]

3.1 Objects

Our view of objects is based on Hewitt's Actor model of parallel computation.

An object can be thought of as an active process that receives messages and performs action on its internal state according to the received message. During the computation, an object can send messages to other objects.

The general properties of objects are as follows:

(1) An object is a process that can have internal states. It becomes active when it receives a message.

(2) The internal state of an object can be operated upon from the outside only by sending it a message, which specifies the operation to be performed.

(3) An object can exchange messages with other objects during its computation.

(4) Any number of object-instances can be generated from a definition of an object.

3.2 Realization of Objects

The following shows how Concurrent Prolog realizes objects.

[1] **A (perpetual) object is a process that calls itself recursively and holds its internal state in unshared arguments**

The state of an object corresponds to the arguments of a process. Its

internal state corresponds to arguments not shared by other processes. An object acts by reducing itself to other objects. A perpetual object survives by reducing itself to itself. A perpetual object changes its state by calling itself recursively with different arguments.

[2] **Objects communicate with each other by instantiating shared variables**
Since parallel processes are realized by And-parallelism, they can be linked by shared variables. These variables are used as communication channels among objects. Message passing is performed by instantiating a shared variable to a message. Because a shared variable can be referred to by multiple processes, a message can be sent to multiple objects at once. Successive communication is possible by the stream communication technique, that is, by instantiating a channel variable to the binary term $\langle message \rangle . X$ (Prolog's counterpart of Lisp's dotted pair, usually written as $[\langle message \rangle \mid X]$) where $\langle message \rangle$ is a message to be sent and X is a new variable to be used in the next communication.

[3] **An object becomes active when it receives a message; otherwise it is suspended.**
The synchronization mechanism forces a process to suspend when it tries to instantiate read-only variables to non-variable terms. Since objects peek into their input stream in read-only mode, they are suspended if the next message is not available yet.

[4] **An object-instance is created by process reduction**
An object instance B is created when an object A is reduced via a clause $A :- \ldots B \ldots$

[5] **Response to a message**
When an object sends a message which requires a response, the response can not be sent through the same shared variable, since logical variables are single-assignment. There are two techniques for sending a response to a message. One is to prepare another shared stream variable, in which the communication flows in the opposite direction. The other uses a technique called incomplete messages, which is explained more fully in Section 5.1. In this technique 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. For example, a message *show* which asks a target object about its internal state is replaced by the message *show*(*State*), where the variable *State* is used as a communication channel that carries the response from the target object back to the sender. The sender will get the response in this variable sometime in the future, when the message is received and processed. So the sender must wait until the response variable is instantiated if it needs to refer to the response.

3.3 The *counter* Example

A simple example of how to describe an object is shown below.

counter([*clear* | *S*], *State*) :−

```
        counter(S ?, 0).
counter([up | S], State) :-
        plus(State, 1, NewState), counter(S ?, NewState).
counter([down | S], State) :-
        plus(NewState, 1, State), counter(S ?, NewState).
counter([show (State) | S], State) :-
        counter(S ?, State).
counter([ ], State).          % for temination.
```

The object *counter* has two arguments. One is an input stream, and the other is its internal state. When receiving a *clear* message, it resets the state to 0. When receiving *up* and *down* messages, it increments or decrements its state by 1, respectively. The process *plus*(X, Y, Z) suspends until at least two of its arguments are instantiated, then instantiates the third so that they satisfy the constraint $X+Y=Z$. If this constraint cannot be satisfied, *plus* fails. The implementation of *plus* is described in Section 5.2.

When receiving a *show*(X) message, *counter* unifies the variable X with the internal state *State*. The last clause terminates *counter* process, upon encountering the end of the input stream. Note that the stream variable is used recursively. In every reduction it is instantiated to a pair : the message and a new variable, to be used in the next communication.

3.4 Object-instance Creation

Object-instance creation is accomplished by parallel-And. A new instance of an object *counter* may be created by executing the following code :

```
terminal(X), use_counter(X ?, C1), counter(C1 ?, 0)
```

where *terminal* is an object that generates the stream of commands produced by a user at a terminal, *use_counter* is an object receiving commands from the terminal and passing the commands to the object *counter*, except for the *show* command, which, in addition to passing it to *counter*, causes the object *use_counter* to wait for the response from the object *counter* and then to output it to the screen. Note that the first argument of *use_counter* and the second argument of the object *counter* are treated as read only variables, because they are used only as input.

```
use_counter([show(Val) | Input], [show(Val) | Command]) :-
        use_counter(Input ?, Command), wait_write(Val).
use_counter([X | Input], [X | Command]) :-
        dif(X, show(Y)) | use_counter(Input ?, Command).
wait_write(X) :- wait(X) | write(X).
```

wait(X) is a Concurrent Prolog system predicate which is suspended if X is not instantiated, and succeeds otherwise. *dif*(X,Y) is a system predicate that succeeds if and when it can determine that X and Y are different (i. e. not unifiable). Note that the stream variable is used as an object-instance name. Message passing is performed against the stream variable and not against the target object itself,

because there are no global names in Concurrent Prolog, and the only information about an object that is accessible from outside are the communication channels to it.

As described before, a new object-instance is created using a definition in the program. However, object-instances created from the same definition are different and can be distinguished by the names of their communication channels. The example below demonstrates this. The object *use_many_counters* is similar to *use_counter*. It receives a command stream from the terminal. When it receives a *create(Name)* message, it creates a new object *counter* and saves its name and a communication channel to it in its internal state. Other messages must be the form *(Name, Command)*, where *Name* specifies the name of an object *counter* to which the message *Command* should be sent.

```
use_many_counters([create(Name) | Input], List_of_counters) :-
        couner(Com ?, 0),
        use_many_counters(Input ?, [(Name, Com) | List_of_counters]).
use_many_counters([(Name, show(Val)) | Input], List_of_counters) :-
        send(List_of_counters, Name, show(Val), NewList) |
        use_many_counters(Input ?, NewList), wait_write(Val).
use_many_counters([X | Input], List_of_counters) :-
        dif(X, create(Y)), dif(X, show(Y)),
        send(List_of_counters, Name, X, NewList) |
        use_many_counters(Input ?, NewList).

send ([(Name, [Message | Y])|List], Name, Message, [(Name, Y)|List]).
send ([C|List], Name, Message, [C|L1]) :- send(List, Name, Message, L1).
```

The object *use_many_counters* has two arguments. One is an input stream from the terminal, and the other is a list of *(Name, Channel)* where *Name* is an identifier of the object *counter* given by the *create* command and *Channel* is the communication channel to that object. The object *send* takes four arguments. The first argument is the same as the second argument of *use_many_counters*. The second and the third arguments are an identifier of the object *counter* and

```
use_many_counters(In ?, [])
        |        In=[create(c1) | In1]
use_many_counters(In1 ?, [(C1, X1)]), counter(X1?, 0)
        |        In1=[create (c2) | In2]
use_many_counters(In2 ?,[(c2, X2), (c1, X1)]), counter(X2 ?, 0)
        |        In2=[create(c3) | In3]
use_many_counters(In3 ?,[(c3, X3), (c2, X2), (c1, X1)]), counter(X3 ?, 0)
```

Fig. 1 Multiple object-instantiation

a message to be sent, respectively. The fourth argument is the updated list of counters. Figure 1 shows a situation in which three *create* commands were processed. Note that since there are no global variables in Concurrent Prolog, an object must keep channel variables associated with other objects in order to send messages to them.

The multi-window system described in Section 4 uses a similar technique to create processes and to associate with them windows and communication channels.

3.5 Default Programming, Filters, and Object Hierarchies

Some object-oriented languages, such as Smalltalk,[3] associate a hierarchy with objects. This hierarchy supports a very convenient form of default programming. Methods for responding to a message can be associated with an object high in the hierarchy, and an object-instance, receiving a message which it does not know how to respond to, can default to an object higher in the hierarchy to respond to the message. In addition to increasing the brevity of programs, such a mechanism also increases their modularity, since a code associated with a class of objects may occur only in the definition of the class, rather than with the definitions of its subclasses. This mechanism also encourages the programmer to identify useful abstractions, so it can be used.

Concurrent Prolog does not have special hard-wired mechanisms to support object hierarchies. However, a certain programming technique, called filters, together with a new Concurrent Prolog construct, *otherwise*, achieves a very similar effect. The resulting programs exhibit a behavior of an Actor-like cooperative group of objects.

Consider the following hierarchy of objects: a rectangular-area; a window-frame, which is a rectangular-area with four border-lines; a window-with-label, which is a window-frame with a label at the bottom of the window. These can be defined by a class-superclass hierarchy (see Fig. 2).

In a language that supports such hierarchies directly, the functionality of a rectangular-area is inherited by the window-frame, and the functionality of a window-frame is inherited by a window with a label. Operationally, an object

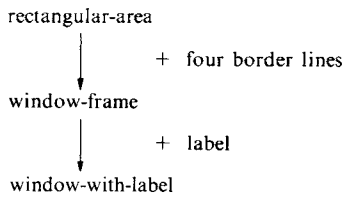


Fig. 2 Class-superclass hierarchy

that receives a message checks whether it knows how to respond to it. If it does not, then it defaults to its parent in the hierarchy to respond to it. In this sense every object in the hierarchy functions like a filter on a stream of message, and this is precisely how object hierarchies are implemented in Concurrent Prolog.

Every object in a hierarchy must have at least one designated input stream and one designated output stream, except the topmost object, which may have an input stream only. The hierarchical structure of the objects is reflected by the structure of the communication network that they form. An object A lower in the hierarchy has its output stream connected to the input stream of an object B next up in the hierarchy. If A receives a message that it cannot respond to, it simply defaults to B by passing to it the message.

The following Concurrent Prolog implementation of the window hierarchy demonstrates this technique. First a rectangular-area is defined.

```
rectangular_area([clear | M], Parameters) :-
    clear_primitive(Parameters) |
    rectangular_area(M ?, Parameters).
rectangular_area([ask(Parameters) | M], Parameters) :-
    rectangular_area(M ?, Parameters).
```

Parameters is a data structure consisting of four parameters (*Xpos Ypos, Width, Height*), where *Xpos* and *Ypos* are the coordinates of the upper-left corner of the area, and *Width* and *Height* are the size of the area. *clear_primitive* is a system defined primitive predicate which clears the screen area specified in its arguments.

From this *rectangular_area* object, a window-frame can be defined. The *frame* object can be viewed as a filter on the input stream of a *rectangular_area*. It filters two types of message, on which it knows how to respond : draw and refresh.

```
create_frame(M, Parameters) : -
    rectangular_area(M ?, Parameters),
    frame(M?, M1).
frame([draw|M], [ask(Parameters) | M1]) : -
    draw_lines(Parameters) |
    frame(M ?, M1).
frame([refresh | M], [clear | M1]) : -
    frame([draw | M], M1).
frame([X|M], [X|M1]) :-
    dif(X, draw), dif(X, refresh) |
    frame(M ?, M1).
```

The first clause specifies the initialization procedure, which creates a *rectangular_area* object by passing the parameters and an original *frame* object with the communication channel to the *rectangular_area*. The rest of the clauses specify the method for interpreting each message. On receiving a *draw* message, it asks the *rectangular_area* about the dimensional parameters and then draws four

border lines. On receiving a *reflesh* message, it sends two messages, *clear* and *draw*, to the *rectangular_area* and self respectively. On receiving other messages, it only passes them to the *rectangular_area*.

To support default programming, a new construct is introduced to Concurrent Prolog, called *otherwise*. An *otherwise* goal that occurs in a guard succeeds if and when all other parallel-Or guards fail. Given the other clauses for *frame*, the last clause is equivalent to :

```
frame([X | M], [X|M1]) :-
    otherwise | frame(M ?, M1).
```

It is not difficult to see that if all clauses for an object have empty guards, then *otherwise* can be implemented via a preprocessor that expands it to an appropriate sequence of calls to *dif*(_, _). If the guards are not empty, then *otherwise* can be implemented via a negation-as-failure primitive. In this sense *otherwise* does not increase the expressive power of Concurrent Prolog more than the addition of negation as failure does. However, an efficient implementation of *otherwise* requires a modification to the Concurrent Prolog interpreter.

Now, a window frame with a label is defined.

```
create_window_with_label(M, Label, Parameters) :-
        create_frame(M1 ?, Parameters),
        window_with_label(M ?, Label, M1).
window_with_label([change(Label) | M], OldLabel, M1) :-
        window_with_label(M ?, Label, M1).
window_with_label([show | M], Label, [ask(Parameters) | M1]) :-
        show_label_primitive(Label, Parameters) |
        window_with_label(M ?, Label, M1).
window_with_label([refresh|M], Label, [refresh | M1]) :-
        window_with_label([show | M], Label, M1).
        window_with_label([X | M], Label, [X | M1]) :-
        otherwise |
        window_with_label(M ?, Label, M1).
```

The first clause defines the initialization procedure which creates the object *frame* with the parameters and a *window_with_label* with the communication channel to the object *frame*. The rest of the clauses define the methods to interpret messages. On receiving a *change* message, it changes the label. On receiving a *show* message, it asks the object *frame* about its parameters and displays the label in the appropriate position in the window, using the predefined predicate *show_label_primitive*. On receiving a *refresh* message, it sends two messages, *refresh* and *show*, to the *frame* and self respectively. On receiving other messages, it only passes them to the *frame*.

In the class-superclass hierarchy, a message which can not be processed by an object is passed to its superclass. In Concurrent Prolog such a hierarchy is simulated by a network of objects connected via communication channels, through which unprocessable messages are sent. A system like Flavor[10] and

Smalltalk-80[3] can permit objects to access instance variables of their superclass. However, in Concurrent Prolog, since a superclass of an object is also an object, such direct access to states of other objects is not possible. Instead of this, an object has to send a message asking about states to the object that plays the role of its superclass.

Table 1 shows the relation between objects and acceptable messages.

Table 1 Objects and acceptable messages

Objects		Messages	Process
rectangular_area	1	*clear*	clear the area
	2	*ask(X)*	instantiate *X* to parameters
frame	3	*draw*	draw four lines
	4	*refresh*	=1+3
		clear	send to *rectangular_area*
		ask(X)	send to *rectangular_area*
window_with_label	5	*change(Label)*	change the label
	6	*show*	display the label
	7	*refresh*	send to *frame*
		draw	send to *frame*
		clear	send to *frame*
		ask(X)	send to *frame*

In the case of an object *window_with_label*, there are two kinds of methods. One is an own method, and the other is a so-called generic method. Generic methods are invoked by sending messages to objects which play the role of a superclass.

In this cooperating objects approach, there is no difference between the class-superclass hierarchy and the part-whole relation. In other words, the role of an object in a group of cooperating objects is not determined from a structural description (such as superclass declaration and part declaration) but from a behavioral description in the form of a communication network. From this point of view, a *rectangular_area* can be seen both as a superclass of a *frame* and as a part of a *frame*. The essential point, however, is the behavioral role of the *rectangular_area* against the *frame*, and in this sense this approach is close to the Actor formalism.

More explanation is needed about the modularity of this approach. First, the internal states of an object can never be operated upon directly from a user of the object. All a user can do is to send a message that specifies the operation to be performed. Thus, the encapsulation of internal states is established. Second, there is no way to access directly component objects of an object from the outside, except by using incomplete messages, as explained in Section 5.1. Thus, the encapsulation of component objects is also established. By these properties of objects, it is possible to construct complicated objects from simpler

objects in a modular way.

§ 4 Multi-window System

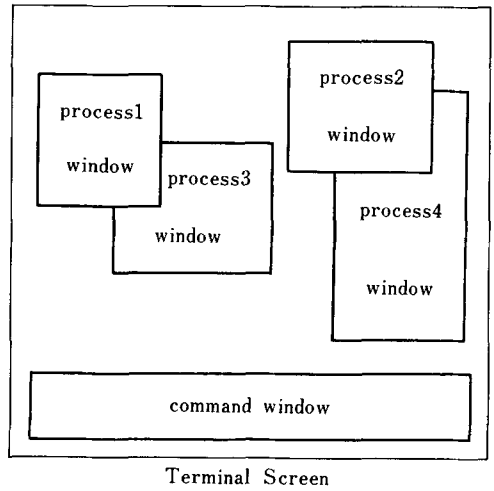
The following is a powerful and expressive example of a Concurrent Prolog implementation of a multi-windows system. The system is architectured after the MUF (Multi User Forks) program and the SM (Session Manager) program of the Yale Tools programming environment.[11] The Tools environment supports only multi-tasking but not multi-windows. Consequently, it does not support concurrent processes output to the screen, whereas this program does. On the other hand, Tools is a real, usable system, while this is still a toy.

The system can create processes dynamically, make them run concurrently and associate each process with a window that can display input and output of the process in the specified position on the screen. Terminal input is managed by the window manager, which can switch the connection between the terminal and a specified process. A user of this system can create several processes dynamically, make them run concurrently, and see concurrently the input-output behavior of each process in the window associated with it.

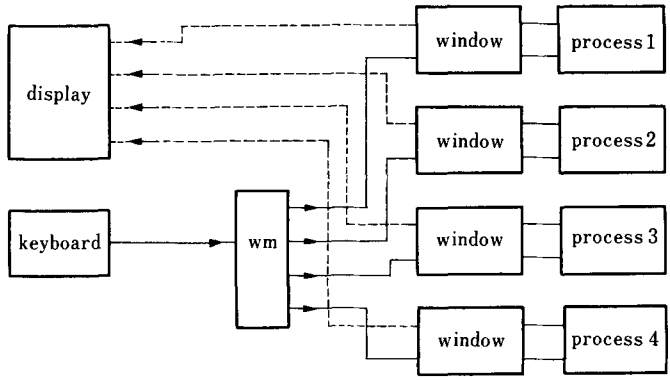
A window consists of a rectangular area, four border lines and a label field like the *window_with_label* defined above (see Fig. 3). It also has a text string as its internal states. A window has two modes. In normal mode it can fill the region with its text string, append a new string and display it by scrolling up if the window is full. In the other mode, called session manager, or sm mode, all the history of the input and the output since the window was created is displayed according to the commands the window receives, by scrolling up and down. In both modes, a window can also move to somewhere else on the screen, change its size, and so on. First the window object will be shown. General form of the window : :

window((*Input, PI, PO*), *State, normal*) in normal mode
window((*Input, PI, PO*), *State, sm*) in sm mode

The first argument of *window* is three communication channels : *Input* is input from the window manager, and *PI* and *PO* are output to the associated process and input from the process, respectively (see Fig. 4). The second argument, *State,* must be a data structure which represents the window's internal state : geographical parameters, contents (text string), and label. In the current implementation, it is represented as ((*X0, Y0, W, H*), *Y, Contents, Label*) where *X0* and *Y0* are the coordinate of the upper-left corner of the window, *W* and *H* are width and height of the window respectively and *Y* points the current cursor position. The form of Contents is (*Tof, Top, Tail, Last*), where *Tof, Top* and *Tail* point the top of file (all the string), the head of the text currently displayed in the screen and end of text respectively. *Last* is used when entering the sm mode to save the current *Top,* which will be restored upon exit from sm mode. Text stings are represented as a bi-directionally linked list of lines, which can be constructed easily by unification with no occur-check. The varia-



(a) A sample screen of multi-window system



(b) Total view of multi-window system

Fig. 3 Multi-window system

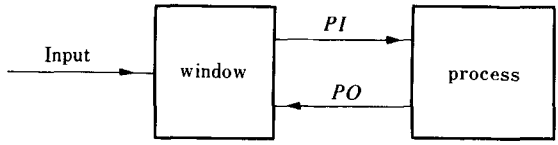


Fig. 4 A window object

ble *Label* is the label of the window. The fourth argument of *window* indicates the current mode of the window and must be sm or normal.
The window in both modes:

```
window( ([erase | In], PI, PO), State, Mode) :-
        erase_window(State) |
        window( (In ?, PI, PO ), State, Mode).
window( ([move(X, Y) | In], PI, PO), State, Mode) :-
        erase_window(State), set_parameters(xy, (X, Y), State, State1 ) |
        window( ([show | In], PI PO), State1, Mode).
window( ([grow(W, H) | In], PI, PO), State, Mode) :-
        erase_window(State), set_parameters(wh, (W, H), Stfoe, State1) |
        window( ([show | In], PI, PO), State, Mode).
window( ([show | In ], PI, PO), State, Mode) :-
        show(State) |
        window( (In ?, PI, PO), State, Mode).
```

On receiving an *erase* message, the window erases the area specified in the *State* parameter, including border lines and label field. On receiving a *move*(X, Y) message, it erases the current area and appears in the new position, the xy-coordinate of the upper-left corner, which is specified by X and Y. *set_parameters*(*xy*, (X, Y), *State*, *State1*) is a primitive method which changes the xy parameter (xy-coordinates of a window) of *State* to X and Y and returns new window parameters *State1*. On receiving a *grow*(W, H), it erases the current area and appears again in the same position with a new size specified by the new width W and the new height H. It can become wider, narrower, longer or smaller according to W and H. On receiving a *show* message, it only redisplays itself in the same position with the same size.

The code of window specific to normal mode ::

```
window( ([sm | In], PI, PO), State, normal) :-
        enter_sm(State) |
                window( (In ?, PI, PO), State, sm).
window( ([X | In], [X | PI], PO), State, normal) :-
        fill_input(X, State, State1) |
                window( (In ?, PI, PO), State1, normal).
window( (In, PI, [X | PO]), State, normal) :-
        fill_output(X, State, State1) |
                window((In, PI, PO ?), State1, normal).
```

On receiving an *sm* message, the window enters the sm mode. On receiving other messages, which must be messages to the associated process, it appends the messages to the current contents, displays it and passes it to the associated process. On receiving messages from the associated process, it also appends the messages to the current contents and displays it.

The code the window executes only in sm mode is :

```
window( ([up | In], PI, PO), State, sm) :-
```

```
        show_up(State, State1) | window( (In?, PI, PO), State1, sm).
window( ([down | In], PI, PO), State, sm) :-
        show_down(State, State1) | window( (In?, PI, PO), State1, sm).
window( ([exit | In], PI, PO), State, sm) :-
        exit_sm(State, State1) |
        window( ([show | In], PI, PO), State1, normal).
window( ([X | In], PI, PO), State, sm) :-
        window( (In?, PI, PO), State, sm).
```

On receiving *up* and *down* messages, the window scrolls up and down the screen respectively. On receiving an *exit* message, it exits from the sm mode and returns to the normal mode.

The window manager, wm for short, has two arguments.

wm(Input, ListOfChannels)

The first argument is the input command stream to the window manager, and the second argument is a list of pairs of a window label and an output channel to the associated process.

The window manager can accept three kinds of messages.

```
wm([create(Label, Process, (PI, PO), (X0, Y0, W, H)) | Input], Processes) :-
        window( ([show | In], PI, PO?),
                ((X0, Y0, W, H), Y0, (C, C, C, C), Label), normal),
        Process,
        wm(Input?, [(Label, In) | Processes]).
wm([resume(Label) | Input], Processes) :-
        find_process(Label, Processes, PI, Processes1)|
        distribute([show | Input], PI, Input, PI1),
        wm(Input?, [(Label, PI1 | Processes1]).
wm([close | Input], [(Label,[]) | Processes]) :-
        wm(Input?, [(Label,_) | Processes1]).
wm([], Processes) :- close_input(Processes).
```

On receiving *create (Label, Process, (PI, PO), (X0, Y0, W, H))* message, it creates a process *Process* and a window with label *Label* monitoring the process's input and output and sends a *show* message to the window. *PI* and *PO* are variables representing the primary input and primary output channels of the process respectively, and may appear in the goal *Process*. *X0*, *Y0*, *W* and *H* are window parameters. On receiving the *resume(Label)* message, it finds the input channel to the process with a name *Label* from the list of processes *Processes* and connects the input stream of the window manager and the input channel of the process by creating the object *distribute*. At the same time, it picks up the process and places it in the top of the list of the processes. On receiving a *close* message, it closes the input channel of the process currently resumed. On reaching the end of the input stream, the window manager closes all the input channels of the processes and terminates.

The object *distribute* has four arguments. The first argument is an input

channel from the terminal. The second and the third arguments are output channels to the window associated with the process currently resumed and to the window manager respectively. The fourth argument returns the updated input channel of the window when the connection is cut. The *distribute* object peeks ahead into the input, and if the input is a window manager command, then it returns the input stream and the window's input channel to the window manager and terminates. Otherwise it passes the input to the window process (Fig. 5).

```
distribute([X | Input], PI, [X | Input], PI) :-
        member(X, [resume(_), close, create(_, _, _, _)]) | true.
distribute([X | Input], [X | PI], Input1, PI1) :-
        otherwise | distribute(Input?, PI, Input1, PI1).
distribute([], PI, [], PI).
```

find- process and close-input are written in the following way.

```
find_process(Label, [Label, PI) | Processes], PI, Processes).
find_process(Label, [PD | Processes], PI, [PD | Processes1]) :-
        find_process(Label, Processes, PI, Processes1).

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

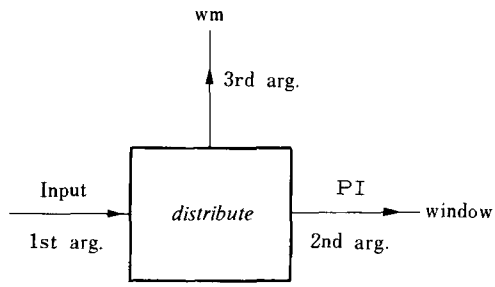


Fig. 5 A distribute object

§ 5 New Object-Oriented Programming Techniques in Concurrent Prolog

Concurrent Prolog supports several powerful object-oriented programming techniques not available easily in the Actor system and other object-oriented languages. These techniques heavily depend on properties of unification.

In object-oriented languages, a message is sent by specifying the name of

the target objects. However, in Concurrent Prolog, objects are connected by shared variables, and a message is sent by instantiating a shared variable to it. Therefore, the name of the target object does not necessarily appear in the message passing phase. Furthermore, broadcasting becomes quite simple, because a message is sent to all the objects that share the variable at once. Generally, shared variables are made at the moment when a process creates a new system of processes, as in the following clause :

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

In the example above, Y is created and used as a communication channel from q to r. Shared variables can also be made dynamically by sending a variable as a part of a message. This means that a communication channel can be made dynamically and it can be sent to other objects as well. A message that contains variables is called an incomplete message. Section 5.1. explains this concept.

As described before, information can be sent implicitly to any number of objects without knowing who the receivers are by simply instantiating a shared variable to it. This can be seen also as if a sender and receivers of a message do not know each other beyond knowing that the variable is shared with some objects. All that the sender has to do is to instantiate a variable as soon as possible, and all that the receivers have to do is to wait until the variable becomes instantiated. This kind of implicit communication is useful for constructing a dependency network like Constraints,[7,8,9] as explained in Section 5.2.

5.1 Incomplete Messages

The concept of incomplete messages* is a new, encompassing programming paradigm, which includes the basic communication mechanism between objects, pipelined processing on stream data, and yields new object-oriented programming techniques. As in the Actor system, Concurrent Prolog is a model of parallel computation, and provides communication methods based on message passing through shared variables. A message is sent by instantiating a shared variable. A message that contains a variable is called an incomplete message. It makes a new variable shared between the sender and the receiver of the message, that is, it creates a new communication channel. Since once a variable is instantiated it will never be rewritten, it can carry only one message. In order to enable subsequent communication, generally a shared variable is instantiated to a pair of a message and a variable which will be used in a next communication, which gives the effects of a stream. Although pipelined processing on stream data usually requires adding new constructs to a language, it is subsumed naturally by the paradigm of the incomplete message.

A prime generator based on a Eratosthenes' sieve algorithm is a typical example of pipeline processing on partially obtained data.

$$primes :- integers(Z, I), sift(I?, J), outstream(J?).$$

---

* In TR-003[9] they are called *partially_determined_messages*.

integers(*N*, [*N* | *I*]) :– *N1 is N*+*1* | *integers*(*N1*, *I*).
sift([*P* | *I*], [*P* | *R1*]) :– *filter*(*I*?, *P*, *R*), *sift*(*R*?, *R1*).
filter([*N* | *I*], *P*, *R*) :– *0 is N mod P* | *filter*(*I*?, *P*, *R*).
filter([*N* | *I*], *P*, [*N* | *R*]) :– *M is N mod P*, *M*>*0* | *filter*(*I*?, *P*, *R*).
outstream([*X* | *S*]) :– *write*(*X*) | *outstream*(*S*?).

The predicate *primes* is the top level goal which is invoked by a user first. It creates three objects : *integers*, which generates an infinite sequence of integers, *sift*, which sifts the integer sequence by the prime numbers obtained so far and *outstream*, which prints out the sequence of prime numbers one by one. All the communication among objects is carried by streams. For example, every time a new integer is obtained, the object *integers* sends it with a new variable which corresponds to a stream (not obtained yet) of integers larger than it. Another logic program implementation of this algorithm, which seems to require a more elaborate control mechanism, appears in.[12)]

As in the case of a *show*(*X*) message to a *counter* object presented in Section 3.2, when a message requires a response, it is sent with a variable which will be instantiated by the receiver to the response, This is also an example of an incomplete message. However, this use of incomplete messages is different from streams, because the object that instantiates the variable in the message is the receiver of the message, not the sender. Once a message is sent to an object, the sender and the receiver run independently as long as they can. If the response variable is not instantiated yet by the receiver when the sender refers to it, then the sender suspends.

This programming technique is extremely useful when implementing managers of shared resources. The following implementation of a queue manager demonstrates this. The queue manager handles the messages *enqueue*(*X*) and *dequeue*(*X*), which represent requests to append *X* at the end of the queue and to return an element positioned at the head of the queue respectively. The predicate *qm* takes three arguments. The first argument is an input channel of requests from users, and the second and the third arguments are pointers to the head of the queue and the tail of the queue respectively. To ensure that the *qm* is invoked with an empty queue, these two pointers must be the same variable in the first invocation. For example, the situation where there are two user processes accessing the queue manager is described as follows.

user1(*X*), *user2*(*Y*), *merge*(*X*?, *Y*?, *Z*), *qm*(*Z*?, *Q*, *Q*)

user1 and *user2* are user processes sending requests to the queue. Those two request streams are merged into one stream by the object *merge* and the resulting stream is sent to the queue manager. The *merge* program is :

merge([*A* | *Xs*], *Ys*, [*A* | *Zs*] :– *merge*(*Xs*?, *Ys*, *Zs*).
merge(*Xs*, [*A* | *Ys*], [*A* | *Zs*]) :– *merge*(*Xs*, *Ys*?, *Zs*).

The queue manager program is :

qm([*dequeue*(*X*) | *S*], [*X* | *Head*], *Tail*) :– *qm*(*S*?, *Head*, *Tail*).
qm([*enqueue*(*X*) | *S*], *Head*, [*X* | *Tail*]) :– *qm*(*S*?, *Head*, *Tail*).

On receiving a *dequeue*(X) message, it instantiates X to the top element of the queue. On receiving *enqueue*(X) message, it inserts X at the end of the queue. The behavior of *qm* is quite interesting when the queue is empty and the queue manager receives a *dequeue*(X) message. It never returns a negative response to the sender of the message. It only unifies the variable X with a variable which is located at the top of the queue. This variable will be instantiated to a queue element sometime in the future, when the *qm* will receive an enqueue message. After the unification, *qm* becomes free from the dequeue request and tries to serve the next request from the input stream. The point is that the interaction between the *qm* and the sender of a dequeue message is completed at the moment of the unification. The sender will not need to send another message to *qm* whether X is instantiated or not, and *qm* will never send any additional message to the sender. The response will be conveyed indirectly by instantiating X to an enqueued element when *qm* will receive an enqueue message.

The behavior of the *qm* is also interesting when it receives the message *enqueue*(X) with X uninstantiated. As in the case above, it unifies the variable X with the tail element of the queue and finishes the processing of the request. The object which will send a dequeue message in the future will receive the variable X if the object sending *enqueue*(X) will not have instantiated X at that time as in the case above. From the point of view of the sender of a *dequeue* message, the situation is the same as in the above example, and it will have to wait for the value of X when it will need to refer to it. The situation can be seen as the object sending *enqueue*(X) reserves the place to which it will really enqueue something only later (Fig.6).

From these observations, it is clear that the behavior of the object *qm* is only to connect logically the arguments of *dequeue* and *enqueue* messages, in their arrival order, and real information is sent from the enqueueing object to the dequeueing object directly. However the important point here is that this logical connection cannot be seen by the users of *qm*. All they can see is the *qm*

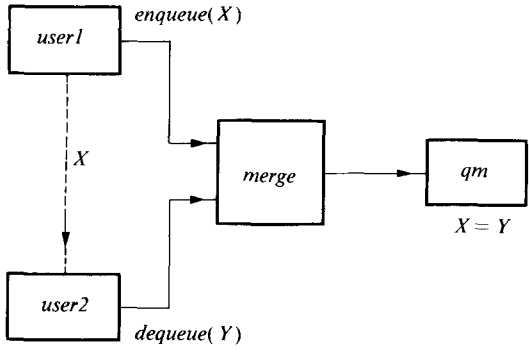


Fig. 6 The queue manager

object. This highly reduces the overhead on the resource manager because the manager will never be locked and request will never be refused. Using incomplete messages, we can create dynamically a new information path, which is hidden from the objects taking part in the message passing. This cannot be expressed in other object-oriented languages as simply as in Concurrent Prolog. This use of incomplete messages reduces message exchanging overhead and gives great expressive power to Concurrent Prolog.

5.2 Constraints

A constraint specifies a dependency relation among properties of objects. It is associated with procedures for satisfying it and can determine properties of objects if enough information about other properties of the objects are determined. There is no static input and output relation among the properties of objects ; rather, which property is input and which is output is determined dynamically, in an indeterminate way.

For example, the constraint *plus(X, Y, Z)* specifies the relation,

$$X + Y = Z$$

where X, Y and Z are properties of some objects. Because the degree of freedom of this relation is two, it can find the value of an unknown property when two of the arguments are determined.

plus(2, 3, Z) ⟶ *instantiate Z to 5.*
plus(2, Y, 5) ⟶ *instantiate Y to 3.*
plus(X, 3, 5) ⟶ *instantiate X to 2.*

A constraint becomes active only when a sufficient number of its arguments are determined. Otherwise, it is suspended.

plus(X, 3, Z) ⟶ *suspended*
plus(X, 3, 5) ⟶ *active and instantiates X to 2.*

Constraints can form a dependency network over properties of objects. In the case of the *plus* constraints, it can represent simple equation systems, like :

$$\begin{array}{lcl} plus(X,A,5) & & X+A=5 \\ \quad \& & & \\ plus(Y,1,X) & \Rightarrow & Y+1=X \\ \quad \& & & \\ plus(Y,5,Z) & & Y+5=Z \end{array}$$

In this network, each *plus* node plays the role of propagating values of properties of objects. For example, if the network receives *1* for A, it instantiates X, Y, Z to *4,3,8* respectively, and if it receives *4* for X, it instantiates A, Y, Z to *1,3,8* respectively, and so on.

Generally the representation of a constraint consists of methods to satisfy the relation. The constraint *plus* is defined in Concurrent Prolog simply as follows.

plus(X, Y, Z) :− wait(X), wait(Y) | Z is X + Y.
plus(X, Y, Z) :− wait(Y), wait(Z) | X is Z − Y.

plus$(X,Y,Z) :- wait(Z),\ wait(X) \mid Y\ is\ Z - X.$

where "*X is Y*"is a system predicate that evaluates an arithmetic expression Y and unifies the result with X. This definition shows that indeterminate computation is realized by Or-parallelism, and that the activation of a constraint is specified by *wait* predicates in guards. A dependency network of constraints can be formed by parallel-And and shared variables. For example, the equation system above is represented as follows.

plus$(X,A,5)$, *plus*$(Y,1,X)$, *plus*$(Y,5,Z)$

The behavior of this network is :: (Fig. 7)

Initially : All constraints are suspended
A is instantiated to 1 externally
then plus$(X,A,5)$ *is now active and instantiates X to 4*
then plus$(Y,1,X)$ *becomes active and instantiates Y to 3*
then plus$(Y,5,Z)$ *becomes active and instantiates Z to 8*

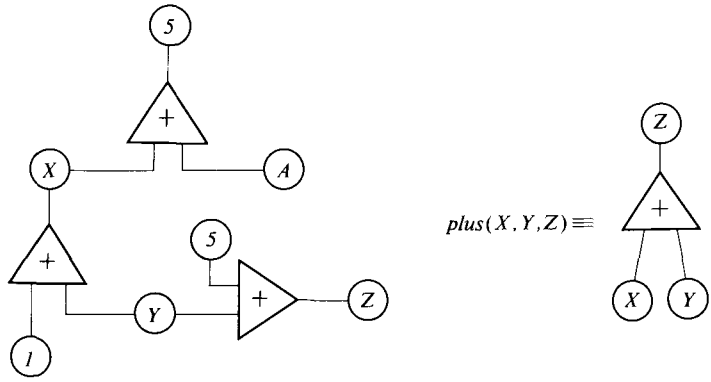


Fig. 7 The constraint network

The current implementation of constraints does not treat the methods for updating values of properties because logical variables are single assignment and can never be rewritten, and the dependency network cannot find the solution, even if there exists a unique solution, when the network contains some circular dependency, for example *plus*$(X,Y,5)$, *plus*$(X,1,Y)$. However, this presents another example of the expressive power of Concurrent Prolog.

§ 6 Conclusion

The name of the game in designing a logic programming language is to find a control regime over logic programs that achieves a desired behavior. One is constrained by the requirements that any result of the computation must be a logical consequence of the axioms in the program, and that the con-

trol regime be both expressive and simple, to make the efficient implementation of algorithms possible without inducing too much runtime overhead. Sequential Prolog (side-effects excluded) is an example of such a logic programming language, designed to run efficiently on a von Neumann machine. Concurrent Prolog is another example.

Given these constraints, there was little freedom in choosing the target programming style when designing Concurrent Prolog. Hence it was an experimental finding, almost a surprise, that Concurrent Prolog lends itself most naturally to a very specific concurrent programming style, namely object-oriented programming. This paper has attempted to convey this finding via programming examples.

These examples have shown that the basic operations of object-oriented programming languages — creating an object, sending and receiving messages among objects, modifying an object's state, and forming class-superclass hierarchies — all correspond naturally to Concurrent Prolog programming techniques rather than to specialized programming language constructs. It is felt that showing that a programming technique in one language subsumes specialized constructs in another language is among the strongest evidence of the expressive power of a programming language.

In addition, a new object-oriented programming paradigm unique to Concurrent Prolog, called incomplete messages, has been presented. This technique subsumes stream communication and greatly simplifies the complexity of communication networks and the communication overhead usually associated with managing shared resources.

This paper provides only a glimpse of the potential of Concurrent Prolog to implementing constraint systems. The ability to determine dynamically the inputs and outputs of an object seems to be an invaluable asset for this task. A subsequent paper will explore further the application of Concurrent Prolog to this task.

Acknowledgements

The authors would like to thank Dan Ingalls for suggesting the problem of implementing a multi-window system as a generic object-oriented programming problem. We would also especially like to thank Kazuhiro Fuchi, Director of the ICOT Research Center, Kouichi Furukawa, Chief of the second research laboratory of the ICOT Research Center, and the other members of the ICOT staff, both for help with this research and for providing a stimulating place in which to work.

References

1) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical

Report, *TR-003* (1983).
2) Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, *8* (1977).
3) The XEROX Learning Research Group: The Smalltalk-80 System, BYTE (Aug, 1981).
4) Kowalski, R.: Predicate Logic as Programming Language, Proc. of IFIP 74 (1974).
5) Clark, K. L. and Gregory, S.: A Relational Language for Parallel Programming, Proc. of the ACM Conf. on Functional Programming Languages and Computer Architecture (1981).
6) van Emden, M. H. and de Lucena, G. J.: Predicate logic as a programming language for parallel programming, Logic Programming, (K. L.Clark and S. A. Tärnlund eds.) (Academic Press, 1982).
7) Steele, G. L.: The Definition and Implementation of a Computer Programming Language based on Constraints, MIT *AI-TR-595* (1980).
8) Borning, A.: The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, ACM Trans.on Programming Languages and Systems, *3, No.4* (1981).
9) Sussman, G. J. and Steele, G. L.: Constraints — A Language for Expressing Almost-Hierarchical Descriptions, Artificial Intelligence, *14* (1980).
10) Weinreb, D. and Moon, D.: Flavors: Message Passing in the Lisp Machine, MIT AI memo *no.* 602 (1980).
11) Ellis, J. R., Mishkin, N., van Leunen, M. and Wood, S. R.: Tools: An Environment for Timeshared Computing and Programming, Research Report, *232* (Department of Computer Science, Yale University, 1982).
12) Pereira, L. M.: Logic Control with Logic, Proc. of the 1st Int. Logic Programming Conf. (1982).