CONCURRENT LISP ON A MULTI-MICRO-PROCESSOR SYSTEM

Shigeo Sugimoto*, Koichi Tabata**, Kiyoshi Agusa* and Yutaka Ohno*

*Department of Information Science
Kyoto University
Kyoto, 606 Japan

** Educational Center for Information Processing
Kyoto University
Kyoto, 606 Japan

ABSTRACT

For solving search problems in AI field, multi-process description is more elegant than the backtracking/coroutine description, from the standpoint of flexible search, high modularity and simple program structure. We propose the language specification of Concurrent LISP, which is a concurrent programming language based on LISP. We have implemented its interpreter on a large scale computer. We are working at the implementation of its interpreter on multi-micro-processor system to realize further efficient execution,

1 INTRODUCTION

This paper proposes a new concurrent programming language, Concurrent LISP, and a multi-microprocessor system for it. We have developed Concurrent LISP for the purpose of applying multiprocessing mechanism to AI programs,

LISP is the most popular language used for AI applications. Many AI systems are written in LISP. Several AI languages have been developed on the basis of LISP, such as PLANNER, CONNIVER and s0 on. These languages have built-in facilities of backtracking and coroutines for flexible and efficient searching to solve problems. However, these facilities have disadvantages; backtracking is too restrictive because an abandoned path on a search tree can not be re-searched, and coroutines may be complicated due to transfer of control among coroutines.

Multi-process facilities are considered to be better for solving search problems than backtracking/coroutine facilities, from the standpoint of flexible search, high modularity and simple program structure. Multi-process mechanism may be implemented in the environments of parallel processing to provide great computation power for AI problems. These facts have led us to design a new LISP with multi-process mechanism called Concurrent LISP[8] and a multi-micro-processor system for it.

II MULTIPROCESSING FOR AI

A. Backtracking, Coroutines and Multiprocessing

Backtracking i3 a conventional technique for problem solving, i.e. trial and error method, Many LISP systems have backtracking facilities 3uch as FAIL and FAILSET operation* In general, backtracking is expensive and restrictive since it follows depth first search method and an abandoned path on a search tree can not be re-searched[7]. Coroutine facilities provided in several LISP systems are convenient for selecting a path which is likely to reach a goal. Several AI systems perform their tasks efficiently by using coroutine facilities with a scheduling table called agenda* A disadvantage of coroutines is that control transfer among coroutines makes program structure complicated. Multi-process facilities provide us higher modularity of control structure than coroutine ones. Comparison of coroutine facilities with multi-process facilities on a simple example is presented:

"We assume that there is a search tree. Each path on a tree is allocated to a search process. Each search process goes along its given path until it finds a goal or a boundary. When . all search processes find the boundary, then the best N search processes are to be selected to continue searching forward. Thi3 activity repeats until a process finds the goal."

Fig.1 shows the outlines of both a multi-process program and a coroutine program for the search problem. There are two types of process, monitor process and search process. In the case of the multi-process program (Fig.1(a)), every search process proceeds concurrently along their given paths, then they reach a boundary or a goal where they communicate with the monitor process via shared variables *or* messages. There is no need to write operations for transfer of control between processes. Thus, each process is dedicated to its own task.

In the case of a coroutine program (Fig. 1(b)), on the other hand, search coroutines and the monitor coroutine communicate with each other by means of RESUME operation. The monitor coroutine selects a search coroutine according to the agenda and resumes the execution of the selected coroutine. The selected search coroutine searches its given path until it finds the boundary or the goal, then it resumes the execution of the monitor coroutine. Therefore, each coroutine must keep track of not only its own task, but also the next coroutine whose execution is to be resumed*

B. Models and Multi-process Programs

From the viewpoint of programming, first we usually build models for given problems and then we

derive programs from the models. Since models usually consist of many components, we must clarify what are components, what the components do and what relations exist among the components. If the components are activated sequentially, a component may be implemented as a subroutine. If the components coexist with others, a component may be implemented as a coroutine or a process in a multi-process system. In AI field, several modeling principles have been used widely, e*g. production systems, actors and so on. To derive programs from models built with those principles, Multi-process description is more elegant than the convetional description techniques because of the modularity of programs and the adaptability for parallel processing.

III CONCURRENT LISP

A. Outline

Concurrent LISP is a concurrent programming language baaed on LISP. It is designed without changing the original language features of LISP* For example, (1) literal atoms are unique, (2) functional notation is preserved, (3) dynamic binding strategy is used and so on.

The current version of Concurrent LISP follows LISP 1.5[3]. In addition to the ordinary LISP functions, Concurrent LISP has three primitive concurrent functions and special functions to manipulate data attached to processes. The three primitive functions are STARTEVAL for process activation, CR (Critical Region function) and CCR (Conditional Critical Region function) for mutual exclusion*

B. Process

A multi-process system written in Concurrent LISP is a set of many cooperating sequential processes, each of which evaluates its given form. We define a "process" in Concurrent LISP as follows:

"A process is an entity which evaluates a form self-containedly."

The process called the main process is activated when evaluation of a top-level doublet, a kind of form, starts. A non-main process is activated when the process activation function STARTEVAL is executed* We define several properties of a process as follows:

1) A process is activated by its parent process except for the main process. The main process is activated in the interpreting loop,
2) A process terminates when it finishes evaluation of the given form or its parent terminates. The value of the form evaluated by the process is called "process value" of the process.
3) Initial environments of a process is the environments of its parent process when the parent executes STARTEVAL*
4) Every process has a unique number given by the interpreter and its own name for user's convenience* In particular, the main process has name "MAIN" and number 1.

The process activation function STARTEVAL is defined below.

starteval[proc 1;proc2; • ;procn]
proci = list[namei;formi],
namei = name of i'th son process,
formi = form to be evaluated by i'th son process.
When a process executes STARTEVAL, the process activates n son processes. Each son process has its own name specified by <u>name</u> and evaluates <u>form</u>. The value of STARTEVAL is a list of names of son processes;

starteval[proci;proc2;• • ;procn]
= list[namei;name2; • ;namen]„

<u>Name</u> must be a form whose value is a literal atom. <u>Form</u> may be any type of form (a constant, a variable or an expression)* When <u>form</u> is an expression, by the parent process actual parameters

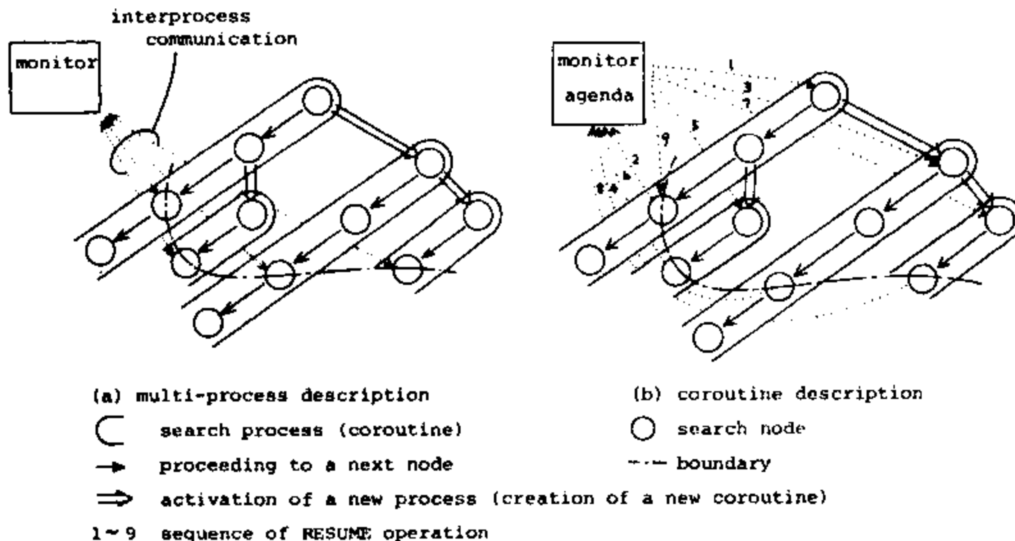

Fig.1 Multi-process and coroutine description

are processed according to the function type of car[form] and passed to the newly created process* The following form means activation of two processes, P and Q, which evaluate forms (F X) and (G Y) respectively.

```
(STARTEVAL ('P (F X)) ('Q (G Y)))
```

Recursive activation of processes is permitted* The following example shows a function for computing factorial using recursive process activation.

```
(FACT (LAMBDA (N)
  (COND ((ZEROP N) 1)
      (T ((LAMBDA (X)
            (TIMES N (CCR (TERMP X) (PROCVAL X))))
          (CAR (STARTEVAL
                 ((GENSYM) (FACT (SUB1 N))))) ))) ))
```

Data area used by processes is classified into two types, static and dynamic data area. Property lists and process control blocks are static, and association lists and variable area allocated on control stacks are dynamic. Data stored in the static area may be used by all processes. For example, bodies of functions are stored on property lists and used by all processes. Data stored in the dynamic area are possessed by each process. For example, a lambda variable bound in a function is allocated in the environments of the process which executes the function* More than one process can use concurrently the same function, which may be either built-in or defined function, because variables bound in the function are possessed by each process.

Concurrent LISP has functions to manipulate data attached to processes, e.g. process names, process values and so on. They are called process data manipulation functions and listed in Table 1.

C. Mutual Exclusion among Processes

Concurrent programming languages need to provide facilities for mutual exclusion to avoid illegal interaction among processes. Concurrent LISP has two primitive functions, CR and CCR, which are shown below. The idea of CR and CCR is originated from [1] and modified to be adapted to the environments of LISP*

The facilities to express process synchronization and interprocess communication are the most important facilities for concurrent programming languages* The two functions, CR and CCR, have enough power to express process interactions, such as P- and V- operator, Hoare's monitor, serializer and guarded command* Moreover, their functional notation strictly follows the original language features of LISP* The functions CR and CCR are defined below*

cr[form]
A process evaluates form with the exclusive rights to access the data area shared among processes* During evaluation of form the process keeps the rights except while it awaits the holding a certain condition in form. (Such wait condition may appear in CCR*) While the process has the rights, all other processes are in suspended state. The value of cr[form] is the value of form;
cr[form] = form.

ccr[condition;form]
A process waits until condition is neither NIL nor F, and evaluates form with the exclusive rights to access the data area shared among processes. (Condition means wait condition for the process to synchronize with other processes.) Pseudo-functions must not appear in condition. During evaluation of form, the process keeps the rights except while it awaits the holding a certain condition in form. The value of ccr[condition;form] is the value of form;
ccr[condition;form] = form*
Nesting of CR and CCR is allowed. Wait during evaluation of CR or CCR occurs due to the nested CCR.

The following example shows interprocess communication using CR and CCR*
Example: Process Q waits until a variable X becomes NIL and evaluates a form (F ARG). X is assumed to be shared between process P and Q.

```
(CR (SETQ X NIL))          - process P
(CCR (NULL X) (F ARG))     - process Q
```

D. Interprocess Communication

Concurrent LISP provides following objects to

Table 1 Functions to manipulate data on pcb's

1) functions which give a truth value according to a process state
termp[p]: If a specified process, p, has already terminated then the value is T else NIL.
waitp[p]: If a specified process, p, is waiting then the value is T else NIL.
asonterm[p], osonterm[p]: If all or at least one of son processes of a specified process, p, have terminated then the value is T else NIL.
asonwait[p], osonterm[p]: If all or at least one of son processes of a specified process are waiting then the value is T else NIL.
2) functions which give a process number or a process name
self[], parent[p], firstson[p], brother[p]: These functions return a process number of itself, a parent, a first son, or the next younger brother.
sonlist[p]: This function returns a list of son process names of a process p.
procname[p], procnum[p]: This function returns a process name or number of a specified process.
3) functions which give process value(s)
procval[p]: This function returns a process value of a specified process*
sonnval[p]: This function returns a process value list of son processes specified process.
4) functions used for mailing between processes
mail[mes;p]: This function appends the value of cons[self[];raes] to the contents of the mailbox of a process p* The value of this function is mes.
recmail[]: This function returns T if the mailbox of the concerned process is not empty else NIL.
getraail[]: This function returns contents of the mailbox of a concerned process. After execution of this function, the mail box is cleared.

express process synchronization and interprocess communication. Using these objects coupled with CR and CCR primitives, we can easily construct functions for interprocess communication.

1) Shared variable
Environments of a parent process can be shared among 3on processes. Son processes can refer free variables bound in the environments of their parent process. The example in III.C. shows interprocess communication via a shared variable. Shared variables are useful for communication in a family of processes.
2) Property list
In ordinary LISP, property lists of literal atoms are used frequently as static data area. In Concurrent LISP, property lists are used as static data area and are shared among all processes.
3) Mailing function
Concurrent LISP has functions called mailing functions for direct message passing between processes. In order to realize mailing functions, we have provided a "mailbox" field in a process control block (peb). See Table 1.
Example: Process P send3 a message "OK" to process Q, and Q receives it.

```
(CR (MAIL 'OK 'Q))                          - process P
(CCR (RECMAIL) (SETQ BUF (GETMAIL)))  - process Q
```

E. Concurrent LISP Interpreter

The Concurrent LISP interpreter executes a program written in Concurrent LISP. The interpreter repeats the following activity (called interpreting loop):
1) To read a top-level doublet,
2) To evaluate the doublet, and
3) To print out the process values*

The main process is activated and terminates at the second phase of the activity. During the life of the main process, its descendant processes are activated and terminate.

The followings are major components,
1) Process control block list,
2) Control stack,
3) Association list ,
4) Property list,
5) Schedule module, and
6) Interpret module.

We have implemented the interpreter on FACOM M-200 with PL/I (about 4800 lines)[6]. The current version has 140 built-in functions.

IV MULTI-MICRO-PROCESSOR SYSTEM

For realization of further efficient program execution of Concurrent LISP, we are working at implementation of its interpreter on a multi-micro-processor system.. The overview of the system is shown in Fig.2.

• processors
Master processor (MP): The main OS on MP performs process management, processor management, I/O management and memory management (stack allocation, garbage collection and so on).
Interpreter processor (IPi): A process operates on IPi to which it is allocated by the main OS.
I/O processor (IOP): IOP performs I/O operation.

* memory
PCB area: Pcb 's are 3tored on this area.
LIST area: List cells are stored on this area.
Random Access (RA) area: Area for literal atoms, numerals, arrays, strings and so on.
Stack area: Area for control stacks of processes.

A separate memory should be provided for each of the above data areas in order to solve the bus bottleneck between common memory and multi-processors and to increase the maximum number of IP's installed. Since, in Concurrent LISP programs, there may be many small processes activated, the system is designed to accept about a thousand processes.

We select 16-bit micro-processor MC68000 for MP and IP's. In [9], which reports performance of LISP systems in Japan, the ratio *of* 1:100 was reported in the performance of a 8-bit micro-processor LISP machine to LISP interpreter on a large scale computer. On the other hand, the range of ratio of performance of 68000 and the 8-bit micro-processor is about 10:1-20:1. We expect our system consisting of ten 68000's will have equal performance to LISP system on a large scale computer.

* language
The specification of Concurrent LISP is slightly modified in order to be adapted to the multi-micro-processor system environment. The major modification is that CR and CCR primitive on a multi-micro-processor system will have the form:

cr [shared object;form], and
ccr[shared object;condition;form].

The number of processes running on IP's increases by this modification, since critical region is divided into several disjoint regions using shared object parameter. This modification imposes restriction on use of list replacement functions, i.e. RPLACA and RPLACD; these are to be eliminated from the set of functions available for users.

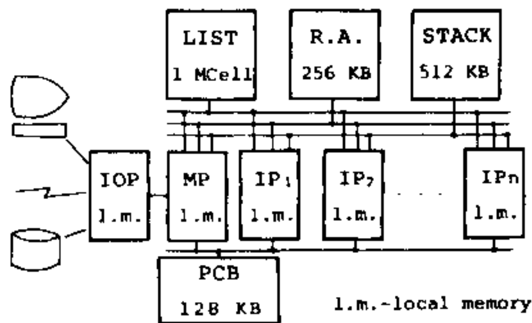

Fig.2 Configuration of multi-micro-processor system for Concurrent LISP

```
<MAIN (LAMBDA (DATA) (PROG () , DATA - the init. board(Brd).
   (CSETQ END NIL)              , END used as a flag.
   (CSETQ. NAMES (LIST (MK_PR_NAME DATA))) ; NAMES - list
                           , of names of search processes (SP's).
   (STARTEVAL ('MONIT (MONITOR))) ; Activate MONIT.
   (PUT (CAR NAMES) 'PROCESS 'PROCESS) ; Mark the Brd.
   (STARTEVAL ((CAR NAMES) (SEARCH (LIST DATA) 1))) ;Actv. SP.
   (CCR (TERMP 'MONIT) (PROGN ; Wait until MONIT terminates.
     (CSETQ END T) ; Set end flag.
     (CCR (ASONTERM ()) ; Wait until all SP's terminate.
        (RETURN (CAR (EVAL (PROCVAL 'MONIT) ()))))
>              :               Return Brd seq. from initial to goal.

<MONITOR (LAMBDA (>
   (CR (WHILE T : Do forever
   (PROGN
   (CCR (EVAL (CONS 'AND (MAPCAR NAMES 'WAITP))())
                                   ; Wait until all SP's wait.
        (COND <(EQ (SORT_SCORE) 'GOAL) ; Goal has been found?
                 (TERMINATE (CAR NAMES>)) ; Return proc. name.
   (COND ((GREATERP (LENGTH NAMES) 3) ; More than 2 proc's?
         (CSETQ DEPTH (PLUS DEPTH DELTA)) ; Change boundary.
         (MAPCAR (LIST (CAR NAMES)(CADR NAMES)(CADDR NAMES))
               '(LAMBDA (X) (MAIL 'CONT X)))) ; Resume.
        (T (CSETQ DEPTH (PLUS DEPTH DELTA)) ; Change boundary.
           (MAPCAR NAMES '(LAMBDA (X) (MAIL 'CONT X)))))
   (CCR (NOT (WAITP (CAR NAMES))) NIL)> ; Synchronization.

<SEARCH (LAMBDA (BD DPT) (PROGN ; BD - Brd seq., DPT - depth.
   (CSET (PROCNAME (SELF)) (CONS BD (EVAL_BD (CAR BD))))
              : Process name has cons[Brd seq.;score of Brd].
   (COND ((EQ (CDR (EVAL (PROCNAME (SELF))())) DPT) ; Goal?
        (CR (COND ((LESSP DPT DEPTH)(CSETQ DEPTH DPT))))
                                            j Change boundary.
        (CR (CSET (PROCNAME (SELF)) ; Set score.
              (CONS (CAR (EVAL (PROCNAME (SELF)) () ) ) 0)))
        (CCR (AND END (ASONTERM ())) (DELETE)) )
                                   ; Wait until MAIN terminates.
      ((GREATERP DPT DEPTH) ; Boundary?
         (CCR (OR (RECMAIL) END) ; Wait until resumed or end.
            (COND ((RECMAIL) (GETMAIL)) ; Resumed.
                  (T (CCR (ASONTERM ()) (DELETE))))) ; End.
         (SEARCH2 BD (ADD1 DPT)) ) ; Continue searching.
      (T (SEARCH2 BD (ADDI DPT)) > ; Continue searching.

<SEARCH2 (LAMBDA (BD DPT) (PROG (TEMP NTEMP)
   (SETQ TEMP (EXPAND (CAR BD))) ; Get next Brd's.
   (MAKE_PR (CDR TEMP)) ; Activate SP's for new Brd's.
   (SETQ NTEMP (MK_PR_NAME (CAR TEMP))) ; Next Brd visited?
   (COND ((IS_PROC NTEMP) (HALT)) ; If visited then halt else
      (T (PUT NTEMP 'PROCESS 'PROCESS))) ; mark the board.
   (SEARCH (CONS (CAR TEMP)(CAR (EVAL (PROCNAME (SELF))()))) DPT)
>                                            ; Search forward.
<MAXE_PR (LAMBDA (X) (PROG (BDLIST)
   (SETQ BDLIST (CAR (EVAL (PROCNAME (SELF))()))) ; Get Brd seq.
   (MAPCAR X '(LAMBDA (X) (CREATE_PR (CONS (CONS X BDLIST) DPT>

<CREATE_PR (LAMBDA (X) (PROG (TEMP) : Activate new SP's.
   (SETQ TEMP (MK_PR_NAME (CAAR X))) ; Current Brd has been
   (COND ((CR (IS_PROC TEMP)) (RETURN ())) ; visited?
      (T (CR (PROGN (CSETQ NAMES (CONS TEMP NAMES))
                (PUT TEMP 'PROCESS 'PROCESS) ; Activate SP.
                (STARTEVAL (TEMP (SEARCH (CAR X)(CDR X))))>

<DELETE (LAMBDA () ; Terminate SP. Proc. val. - Brd seq.
   (TERMINATE (CAR (EVAI. (PROCNAME (SELF))()) ) ) ) >

<HALT (LAMBDA () (PROGN : Halt searching.
  <CR (CSET (PROCNAME (SELF)) ; Wait until end of execution,
      (CONS (CAR (EVAL (PROCNAME (SELF))())) MAX)>
   (CCR (AND END (ASONTERM ())) (DELETE))> ; Return Brd seq.
```

NOTES: Literal atoms and auxiliary functions whose definitions are absent in the text.
MAX - maximum integer. DEPTH = depth of boundary.
DELTA - DEPTH(new) - DEPTH(old).
mk_pr_name [data] makes search process names,
sort_score[] sorts and tests scores of SP's.
eval__bd [Brd] evaluates Brd and returns the score,
expand(Brd] expands Brd and returns next Brd list.
is_proc[name] tests whether name exists or not.

Fig.3 The 8-puzzle program

V EXAMPLE

An example program for 8-puzzle is presented. The board of 8-puzzle is represented as a list of integer, Fig.3 shows the main part of the program to solve the puzzle. The strategy used in this example follows the search model shown in II.A* During execution of the program, there exist main process, MONIT process and search processes. The main process activates MONIT process and the search process which examines the initial board- MONIT process monitors the search processes. When all search processes reach the boundary, MONIT selects three processes, which are most likely to reach the goal. From their boards given by their parent process, search processes start searching. A search process examines a board, expands it and activates its sons. Fig.4 and Fig,5 show the results of the program execution and relations among processes respectively.

VI CONCLUDING REMARKS

Multi-process description mechanism Is more elegant than the conventional description mechanisms, such as backtracking and coroutine description. Since LISP is a "common language" in AI field in a sense, we consider, it is necessary to introduce LISP-based concurrent programming languages which do not change the original language features of LISP such as functional notation and simplicity*

```
MAIN(((1 2 3)(4 8 5)(7 6 9)))      ; Inputted doublet.

PROCESS NAME = MAIN
((( 1 2 3)( 4 5 6)( 7 8 9))(( 1 2 3)( 4 5 9)( 7 8 6))(( 1 2
3)( 4 9 5)( 7 8 6))(( 1 2 3)( 4 8 5)( 7 9 6))(( 1 2 3)( 4 8
5)( 7 6 9)))
 PROCESS NAME = MONIT
 #123456789
 PROCESS NAME = #123485769        ; @1
 ((( 1 9 2)( 4 8 3)( 7 6 5))(( 1 2 9)( 4 8 3)( 7 6 5))(( 1
 2 3)( 4 8 9)( 7 6 5))(( 1 2 3)( 4 8 5)( 7 6 9)))
  PROCESS NAME = #123485796       ; @2
    - - -
   PROCESS NAME = #123485976      ; @3
    - - -
   PROCESS NAME = #123945786      ; @4
    - - -
    PROCESS NAME = #123745986     ; @5
    - - -
   PROCESS NAME = #123459786      ; @6
    - - -
    PROCESS NAME = #123456789     ; @7
    ((( 1 2 3)( 4 5 6)( 7 8 9))(( 1 2 3)( 4 5 9)( 7 8 6))((
    1 2 3)( 4 9 5)( 7 8 6))(( 1 2 3)( 4 8 5)( 7 9 6))(( 1 2
    3)( 4 8 5)( 7 6 9)))
   PROCESS NAME = #139425786      ; @8
    - - -
  PROCESS NAME = #123498765       ; @9
    - - -
   PROCESS NAME = #123948765      ; @10
    - - -
   PROCESS NAME = #123468795      ; @11
    - - -
```

Search process names, #123485769 etc., are determined according to their given boards. Both DEPTH and DELTA are initialized to have 1 as their contents.

Fig.4 Results of execution of the example program

From this standpoint, Concurrent LISP is designed. Therefore, its sequential part is compatible with sequential LISP and its concurrent part allows flexible multi-process description. Also we can write easily such programs in Concurrent LISP as those which have similar control structure to backtracking, coroutines, generators and so on.

A few researches on multiprocessing were proposed at 1980 LISP Conference[4][5][10]. In [4], a pseudo-multiprocessing system implemented on ordinary LISP is described. In [5], a parameter passing mechanism which has parallelism is described. In [10], a pseudo-multiprocessing system using continuation mechanism of SCHEME is described. However, they do not satisfy our requirements for concurrent programming languages based on LISP.

We have applied Concurrent LISP to make an interpreter of an actor-based language including the message passing mechanism of PLASMA[2]. The length of the interpreter is 150 lines. Thus, Concurrent LISP can be used not only as a programming language but also as an effective host language for special purpose languages.

Multi-processor systems are becoming popular with the progress of micro-processors. The architecture of our multi-micro-processor system using general purpose micro-processors saves man-power for design, construction and maintenance.

Concurrent LISP on a multi-processor system will work sufficiently well as a self-contained programming language and as a host language for special purpose languages.

ACKNOWLEDGMENTS

We thank Dr. K. Itoh, research associate of Sophia University, and Mr. T. Masaki, B.E. of Kyoto University, for their useful discussions and important contributions to this paper.

REFERENCES

[1] Brinch Hansen, P., "Operating System Principles", Prentice Hall, 1973

[2] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages", Artif* Intell.. vol.8, 1977

[3] McCarthy, J., et al,, "LISP 1.5 Programmer's Manual", The MIT Press, 1962

[4] McKay, D, and Shapiro, S., "MULTI - A LISP-based multiprocessing system", 1980 LISP Conference, Aug. 1980

[5] Prini, C, "Explicit Parallelism in LISP-like language", 1980 LISP Conference, Aug* 1980

[6] Sugimoto, S., et al.."Concurrent LISP and its Interpreter", 16th WGAI of IPSJ, July 1980

[7] Sussman, G,J, and McDermot, D.L.,"From PLANNER to CONNIVER - A genetic approach", FJCC, 1972

[8] Tabata, K., et al», "Concurrent LISP", 21th Annual Convention of IPSJ, July 1979

[9] Takeuchi, I., "On the 2nd LISP Contest", J0H0-SH0RI, vol,20, no,3, 1979

[10] Wand, M., "Continuation-based Multiprocessing", 1980 LISP Conference, Aug. 1980

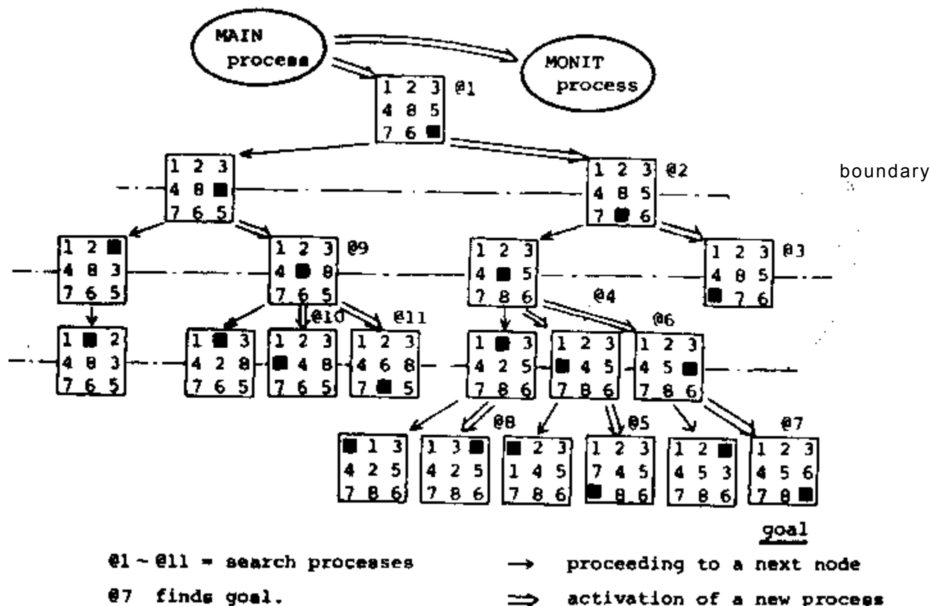

Fig.5 Relations among processes of the 8-puzzle program