On Effective Transformations of Communicating Sequential Processes

A Thesis Submitted to
the Faculty of Yamanashi University for
the Master's Degree

[redacted]
Musha, Hiroyuki

March 19, [redacted]

Contents

1. Introduction

Distributed computing models are natural and powerful systems for describing both concurrent and sequential computing phenomena and they gain growing interests in connection with VLSI technology. The system of Communicating Sequential Processes, which we call CSP in the following of this thesis, is one of those models proposed by C.A.R. Hoare in [41].

In CSP, input and output of processes are considered basic primitives. Combined with nondeterminism, those primitives provide us simple and transparent descriptions of algorithms. Algorithms described in CSP, as they are, however, are not efficiently executable under conventional computing environment. Effective scheduling algorithms of processes are needed.

In this thesis, methods for transformation of descriptions of algorithms in CSP (CSP programs for short) into sequentially executable programs are presented. Coroutines are set to the target of the transformation and three different algorithsm for transformation are described; the first algorithm is simple and can be applied to general CSP programs but not efficient, the other two algorithms can only be applied to restricted classes of CSP programs but more efficient than the first one. Those algorithms must contribute to the problem of scheduling of processes.

The rest of this thesis is organized as follows. In the next chapter, definitions of CSP, coroutines and necessary concepts in graph theory are given. Chapter 3 overviews the algorithms for transformation. In Chapter 4, syntax and semantics of the target of transformation, a language with a kind of coroutine facility, is described. Chapter 5, 6 and 7 show the algorithms for

transformation. In Chapter 8, we compare the method of transformation taken in this thesis with other related works. Conclusion is presented in Chapter 9.

2. Preliminaries

In this chapter, preliminary concepts for discussions given in the following chapters are defined. In 2.1, definitions of CSP, their syntax and semantics, are given, definitions of coroutines are given in 2.2, and 2.3 states definitions of terms related to graph theory.

2.1 Definitions of CSP

In this section, definitions of CSP are given. At first informal introductions are given, then in the following section, we show their syntax and semantics rigorously, and in Section 2.1.3, examples of CSP programs are shown.

2.1.1 Overview of CSP

In this section, sysntax and semantics of CSP are informally described. A CSP program is a collection P of processes, which share no common variables at all and are supposed to be executed concurrently. Communication between two processes p and q of P is expressed by the input and output commands

q?v

and

p!e,

where e is an expression and v is a variable in which the received value of the expression of e is assigned. Execution of these commands are synchronous, i.e. p waits at "q?v" until q is ready to output the message at "p!e" and vice versa.

Constituents of each process are commands based on Dijkstra's guarded commands [14,15,25], which can be classified into two types: simple commands and structured commands. The

members of the simple commands are the assignment command, the input command, the output command and the null command which does nothing. Structured commands, alternative and repetitive commands, are organized by a set of guarded commands and express selective and repetitive execution.

A guarded command is executed when its guard does not fail. An alternative command fails if all guards fail. A repetitive command specifies as many iterations as possible of its constituent alternative commands, and it terminates when all guards fail.

An input command can appear in the end of a guard and is executed only when a corresponding output command is executed; it is called an input guard. The input command fails if the process specified is terminated; the execution suspends if the corresponding process is not ready to output, which can result in deadlock. In recent papers [1,19,28,50] output guards are also permitted, however, in the rest of this thesis output guards are not supposed to appear in guards.

2.1.2 Formal Description of CSP

In this section, syntax and semantics of CSP are defined. At first the whole sysntax is shown in terms of extended BNF, and then, the meaning of each command is explaind. Examples of usage of theses commands are in the next section.

The whole syntax of CSP is as follows:

```
<command> ::= <simple command>|<structured command>
<simple command> ::= <null command>|<assignment command>
                    |<input command>|<output command>
<structured command> ::= <alternative command>
                        |<repetitive command>
<null command> ::= skip
<command list> ::= {<declaration>; | <command>;}<command>
```

```
<parallel command> ::= [<process>{||<process>}]
<process> ::= <process label><command list>
<process label> ::= <empty>|<identifier>::
  |<identifier>(<label subscript>{,<label subscript>})::
<label subscript> ::= <integer constant>|<range>
<integer constant> ::= <numeral>|<bound variable>
<bound variable> := <identifier>
<range> ::= <bound variable>:<lower bound>..<upper bound>
<lower bound> ::= <integer constant>
<upper bound> ::= <integer constant>

<assignment command> ::= <target variable>:=<expression>
<expression> ::= <simple expression>|<structured expression>
<structured expression> ::= <constructor>(<expression list>)
<constructor> ::= <identifier>|<empty>
<expression list> ::= <empty>|<expression>{,<expression>}
<target variable> ::= <simple variable>|<structured target>
<structured target> ::= <constructor>(<target variable list>)
<target variable list> ::= <empty>|<target variable>
      {,<target variable>}

<input command> ::= <source>?<target variable>
<output command> ::= <destination>!<expression>
<source> ::= <process name>
<destination> ::= <process name>
<process name> ::= <identifier>|<identifier>(<subscripts>)
<subscripts> ::= <integer expression>{,<integer expression>}

<repetitive command> ::= *<alternative command>
<alternative command> ::= [<guarded command>
      {[]<guarded command>}]
<guarded command> ::= <guard>--><command list>
      |(<range>{,<range>})<guard>--><command list>
<guard> ::= <guard list>|<guard list>;<input command>
      |<input command>
<guard list> ::= <guard element>{;<guard element>}
<guard element> ::= <boolean expression>|<declaration>
```

A CSP program is a collection of disjoint processes each of which is organized by a list of commands. Commands can be classified into two types: structured commands and simple commands. The members of simple commands are the null command, the assignment command, the input command and the output command. The members of structured commands are the alternative command and the repetitive command.

A command specifies the behavior of a device executing the command and returns one of the two values "success" or "fail"

when it is executed. If the command is executed and returns "success", it changes the states of the process (or the processes) involved. If the command returns "fail", the execution of the whole system aborts.

A null command, which is denoted by "skip", does nothing and never fails.

An assignment command specifies evaluation of its expression, and assignment of the denoted value to the target variable. A simple target variable may have assigned to it a simple or a structured value. A structured target variable my have assigned to it a structured value, with the same constructor. The effect of such assignment is to assign to each constituent simpler variable of the structured target the value of the corresponding component of the structured value. Thus, the value denoted by the target variable, if evaluated after a successful assignment, is the same as the value denoted by the expression, as evaluated before the assignment.

An assignment fails if the value of its expression is undefined, or if that value does not match the target variables, in the following sense: A simple target variable matches any value of its type. A structured target variable matches a structured value, provided that:

(1) they have the same constructor,

(2) the target variable list is the same length as the list of components of the value, and

(3) each target variable of the list matches the corresponding component of the value list. A structured value with no components is known as a "signal."

Input and output commands specify communication between two

processes. Communication occurs between two processes when

(1) an input command in one process specifies as its source the process name of the other process;

(2) an output command in the other process specifies as its destination the process name of the first process; and

(3) the target variable of the input command matches the value denoted by the expression of the output command.

On these conditions, the input and output commands are said to correspond. Commands which correspond are executed simultaneously, and their effect is to assign the value of the expression of the output command to the target variable of the input command.

An input command fails if its source is terminated. An output command fails if its destination is terminated or if its expression is undefined.

The requirement of synchronization of input and output commands means that the process which become ready to communicate first have to be delayed its execution until the corresponding command in the other process also becomes ready, or the other process terminates. It is possible that the delay will never be ended, that is a deadlock.

A set of guarded commands constitutes an alternative or a repetitive command. A guarded command can be executed only if the execution of its guard does not fail. First its guard is executed and then its command list is executed. A guard is executed by execution of its constituent commands from left to right. Boolean expressions are evaluated: if it denotes false, the guard fails; but an expression that denotes true has no effect. A declaration introduces a fresh variable with a scope that extends from the

place of the declaration to the end of the guraded command. An input command at the end of a guard is executed only if and when a corresponding output command is executed.

An alternative command specifies execution of exactly one of its constituent guarded commands. Consequently, if all guards fail, the alternative command fails. Otherwise an arbitrary one with successfully executable guard is selected and executed.

A repetitive command specifies as many repetitions as possible of its constituent alternative command. Thus, when all guards fail, the repetitive command terminates with no effect, with returning "success." Otherwise, the alternative command is executed once and then the whole repetitive command is executed again. Consider a repetitive command when all its true guard list end in an input guard. Such a command have to be delayed until either

(1) an output command corresponding to one of the input guards becomes ready, or

(2) all the sources named by the input guards have terminated.

In case (2), the repetitive command terminates. If neither event ever occurs, the process fails in deadlock.

2.1.3 Examples of CSP Programs

In this section, examples of descriptions of algorithms in CSP are shown.

(1) Subroutines (Procedures)

The following is an example of subroutines, which receives x and y from process X and returns (x div y) and (x mod y).

```
   /*  a solution of Knight's Tour in CSP  */
[TRY(i:2..NSQ)::
    board:(N,N) integer;
    x, y, u, v, k: integer;
    *[TRY(i-1)?(x,y,board) -->
       k := 1;
       *[k<=8 -->
          nextplace!(x,y,k);
          nextplace?(u,v);
          [1<=u;u<=8; 1<=v;v<=8; board(u,v)=0 -->
             board(u,v) := i;
             TRY(i+1)!(u,v,board);
             board(u,v) := 0;
          □u<1;8<u; v<1;8<v; board(u,v)< >0 -->
             skip
          ];
          k := k+1
     ] ]
||
 TRY(1)::
    board: (N,N) integer;
    j, k:integer
    j := 1;
    *[j<=N -->
       k := 1;
       *[k<=N -->
          board(j,k) := 0;
          k := k+1
        ];
       j := j+1
     ];
    j := 1;
    *[j <= (N+1)/2 -->
       k := j;
       *[k <= (N+1)/2 -->
          board(j,k) := 1;
          TRY(2)!(j,k,board);
          board(j,k) := 0;
          k := k+1
        ];
       j := j+1
]   ]
```

```
 /* 8-Queens Problem in CSP */
[TRY(i:1..8)::
   A:(1..8) boolean; B:(2..16) boolean;
   C:(-7..7) boolean; X:(1..8) integer;
   *[TRY(i-1)?(A,B,C,X) -->
      j:integer; j:=1;
      *[j<=8; A(j); B(i+j); C(i-j) -->
         X(i) :=j;
         A(j)   := false;
         B(i+j) := false;
         C(i-j) := false;
         TRY(i+1)!(A,B,C,X);
         A(j)   := true;
         B(i+j) := true;
         C(i-j) := true;
         j := j+1        ]]
||
TRY(Ø)::
   A:(1..8) boolean; B:(2..16) boolean;
   C:(-7..7) boolean; X:(1..8) integer;
   i:integer;
   i:=1; *[i<=8  --> A(i):=true; i:=i+1];
   i:=2; *[i<=16 --> B(i):=true; i:=i+1];
   i:=-7;*[i<=7  --> C(i):=true; i:=i+1];
   i:=1; *[i<=8  --> X(i):=Ø;    i:=i+1];
   TRY(1)!(A,B,C,X)
||
TRY(9)::
   A:(1..8) boolean; B:(2..16) boolean;
   C:(-7..7) boolean; X:(1..8) integer;
   *[TRY(8)?(A,B,C,X) -->
      PRINT!X           ]
]
```

(5) Description of a Sorter

The following program describes a sorter which sorts an array of integers.

```
  /* a description of a sorter in CSP */
[ sorter(i:1..MAX)::
   num, ord, next, ordnext: integer;
   sorter(i-1)?(num,ord) -->
      *[sorter(i-1)?(next,ordnext) -->
         [ num>next --> ord := ord+1
          num=next --> skip
          num<next --> ordnext := ordnext+1
         ];
         sorter(i+1)! (next,ordnext)
       ];
      source!(num,ord)
||
  sorter(0)::
   num, ord, next, ordnext: integer;
   source?(num,ord) -->
      *[source(next,ordnext) -->
         [ num>next --> ord := ord+1
          num=next --> skip
          num<next --> ordnext := ordnext+1
         ];
         sorter(1)!(next,ordnext)
       ];
      source!(num,ord)
||
  source ::
   a: (0..MAX)integer; i, j, num, ord: integer;
   i := 0;
   *[i<=MAX; input?a(i) -->
      sorter(0)!(a(i),0);
      i := i+1
    ];
   j := 0;
   *[j<i -->
      sorter(i)?(num,ord);
      a(ord) := num;
      output!a(j);
      j := j+1
    ];
]
```

2.2 Coroutines

A set of coroutines [11] is the target of transformation explained in the following chapters. In this section, definitons of courintes and semicoroutines [59], a restricted kind of coroutines, are given. The target of our transformation, a concrete language with coroutine facility, is described in Chapter 4.

A coroutine is defined as a routine (subprogram) which has the following two features:

(1) the values of the variables local to the routine are retained between successive activations of the routine, and

(2) when the control reenters the routine, the execution resumes at the point where it left off last time.

A semicoroutine, a restricted kind of coroutines, is defined as a routine which satisfies (1) and (2) above and, in addition,

(3) a semicoroutine must be activated by a caller to which it returns the control on completion of its task.

2.3 Graph Theory

In this section, terms in graph theory necessary to understand the discussions given in the following chapters are defined; definitions of the terms are due to [32].

A DIGRAPH D consists of a finite set V of POINTS and a collection of ordered pairs of distinct points. Any such pair (u,v) is called an ARC or DIRECTED LINE and will usually be denoted by uv. The arc uv goes from u to v and is INCIDENT with u and v. We also say that u is ADJACENT TO v and v is ADJACENT FROM u. The OUTDEGREE od(v) of a point v is the number of points adjacent from it, and the INDEGREE id(v) is the number adjacent

to it.

A (DIRECTED) WALK in a digraph is an alternating sequence of points and arcs, v0, x1, v1, ..., xN, vN in which each arc xI is vI-1 vI. The LENGTH of such a walk is N, the number of occurrences of arcs in it. A CLOSED WALK has the same first and last points and a SPANNING WALK contains all the points. A PATH is a walk in which all points are distinct; a CYCLE is a nontrivial closed walk with all points distinct (except the first and last). If there is a path from u to v, then v is said to be REACHABLE FROM u, and the DISTANCE, d(u,v), from u to v is the length of any shortest such path. A SEMIWALK is an alternating sequence v0, x1, v1, ..., xN, vN of points and arcs, but each arc xI may be either vI-1 vI or vI vI-1. A SEMIPATH, SEMICYCLE, and so forth, are defined as expected. A digraph is STRONGLY CONNECTED, or STRONG, if every two points are mutually reachable; it is UNILATERALLY CONNECTED, or UNILATERAL, if for any two points at least one is reachable from the other; and it WEAKLY CONNECTED, or WEAK, if every two points are joined by a semipath.

A STRONG COMPONENT of a digraph is a maximal strong subgraph. Let s1, s2, ..., sN be the strong components of a digraph D. The CONDENSATION D* of D has the strong components of D as its points, with an arc from sI to sJ whenever there is at least one arc in D from a point of sI to a point in sJ.

An ACYCLIC digraph contains no directed cycles. A SOURCE in D is a point which can reach all others; an OUT-TREE is a digraph with a source having no semicycles.

3. Overview of Transformation

In Chapter 4, the target language of the transformation is described and in Chapter 5 through Chapter 7, three kinds of algorithms for transformation of CSP programs into coroutines are described. In this chapter, a general methods which are common to those three algorithms and also the differences of those three algorithms are presented.

All the three algorithms in the following chapters transform CSP programs in the following manner:

(1) a process of a CSP program will be transformed into a coroutine,

(2) a special routine called scheduler is introduced to manage the selection and the execution of those coroutines,

(3) each command of a process, except for input and output commands, will be transformed into the same command of the target coroutine,

(4) commands which transfer the control of the execution is introduced,

(5) globally accessible variables are introduced; in those variables, values of the messages to be passed, types of the messages to be passed, and states of the processes are stored, and

(6) communication among processes are realized through the global variables: a sender of a message first writes the value and the type of the message there, and then the receiver of the message reads and assigns the value into the target variable of the message.

We compare the three algorithms for transformation described in Chapter 5, 6 and 7. The first difference of those three

algorithms to be discussed is the range of CSP programs to which those algorithms can be applied. The first algorithm, which is described in Chapter 5, can be applied to any kind of CSP programs. On the other hand, the second and the third algorithms, which are described in Chapter 6 and Chapter 7 respectively, can only be applied to CSP programs that satisfy certain conditions. The second algorithm can be applied to CSP programs which do not contain any input guard. The third algorithm can be applied to CSP programs which satisfies three conditions stated in terms of two graphs which represents the form of communication among processes.

Concerning about efficiency of the execution of transformed programs, we can say the following. The duty of the scheduler, the special routine that manages the execution, is the heaviest in the programs transformed according to the first algorithm. In those programs, every coroutine has to be activated by the scheduler every time it is activated. In programs transformed according to the second algorithm, once the scheduler activates a coroutine, the execution can proceed, without returning the control to the scheduler, as far as the execution reaches the end of the coroutine. The control flows among coroutines which must be activated for the execution of the first coroutine. Programs transformed according to the third algorithm are executable most efficiently. The only thing the schduler has to do is to activate a special routine called source which leads the execution. The control flows among processes almost in the same manner as in the programs transformed according to the second algorithms but there is no need for the scheduler to activate other processes after receiving the control again from the source; the execution

terminates.

For the sake of convenience, in the following chapters of this thesis, constituents of the target of transformation, coroutines, will also be called processes; we use the words processes and coroutines interchangeably when we discuss the target language.

4. Target of Transformation: A Coroutine Language ASL

The target language of transformation, which we call ASL, is defined in this chapter. As stated in Chapter 3, commands which are not related to input or output commands are borrowed from CSP preserving their syntax and semantics. Thus, nondeterminism is also contained in this language. Commands related to input or output of processes are changed. Moreover, commands which express transfer of the control among routines and global variables to be used for communication of processes are introduced.

In Section 4.1, we describe auxiliary variables to be introduced. In Section 4.2, syntax and semantics of commands of ASL are formally described.

4.1 Auxiliary Variables

In order to explain the meanings of the commands of ASL, we introduce auxiliary variables which do not appear in the text of ASL programs. Each process (coroutine) has several globally accessible variables other than local variables which are inaccessible from other processes. One of those is a variable which contains the state of the process and is reffered by

p.status,

where p is the name of a paticuler process. The variable p.status contains one of the following value:

(1) ready (denoted by RE) -- which indicates that the process is ready to proceed its execution if it is activated,

(2) output waiting (denoted by OW) -- which indicates that the process was suspended when it tried to output a message and can not proceed its execution unless the corresponding process reaches the place of the rendezvous and receives the message,

(3) input waiting (denoted by IW) -- which indicates that the process was suspended when it tried to input a message and can not proceed its execution unless the corresponding process reaches the place of the rendezvous,

(4) input waiting in guard (denoted by IWIG) -- which indicates that the process is suspended when it tried to execute an alternative or a repetitive command which contains input guards but none of the guards suceeded, and

(5) terminated (denoted by TE) -- which indicates that the process has already finished its execution.

Other than status explained above, for each process, we prepare globally accessible variables whose names and functions are stated in the following:

(1) message box (denoted by MSGB) -- which contains the value of of the message to be passed from the process,

(2) message type (denoted by MSGT) -- which contains the type of the message to be input or output at an input or an output command,

(3) partner (denoted by PTNR) -- which contains the name of the partner of the communication to be taken place at the time, and

(4) caller -- which contains the name of a process which activated the process by a call command or a resume command which will be explained in the next sebsection. We refer those variables as

<process name>.<name of the variable>.

For clear explanations of the mechanisms of the execution of ASL, we introduce one more common variable named executing-process and a local variable named local-sequence-control. The variable executing-process, which will be denoted by EXECP,

contains the name of the process to be executed next. The variable local-sequence-control, which will be denoted by LSC, points the command of the process to be executed next. We assume that the processer of ASL programs picks the content EXECP before executing a command if necessary, and then executes a command pointed by the LSC of the process. We also assume that the content of the LSC is renewed properly after executing the commnd and it is retained between succesive acctivations of the process; i.e. processes behave as coroutines. In the following sections, we define the meanings of commands of ASL using those variables defined in this section.

4.2 Commands of ASL

Syntax and semantics of commands of ASL is described by using the variables defined in the previous section.

In order to transfer the control of the execution, we introduce control commands as follows:

```
<control command>  ::= <call command>
                     | <return command>
                     | <resume command>
<call command>     ::= call <process name> (<status>)
<return command>   ::= return (<status>)
<resume command>   ::= resume <process name> (<status>)
<status>           ::= RE | IW | IWIG | OW | TE.
```

A call command

```
call q (ST)
```

in a process p, transfers the control of the execution to the named process q, sets the status of p to ST (here ST is one of the following: RE, IW, IWIG, OW, or TE), and assigns the name of the activating process, p, into q.caller, that is:

```
call q (ST)        /* in p */
```

```
≡q.caller := p;
  p.status := ST;
  EXECP    := q .
```

The following return command

```
return ( ST )
```

in a process p returns the control to the caller of p and sets the status of p to ST, that is:

```
return ( ST )      /* in p  */

≡p.status := ST;
  EXECP    := p.caller.
```

The following resume command

```
resume q ( ST )
```

in a process p activates the process p and sets the p.caller if p.call is not q. The command also sets the status of p to ST. That is:

```
resume q ( ST )      /* in q */

≡p.status := ST;
  [p.caller = q --> skip
  []p.caller <>q --> q.caller := p ];
  EXECP := q
```

Message commands which take charge of message passings have the following form:

```
<message command> ::= <receive command>
                    | <send command>
<receive command>
   ::= receive (<targetvariable>) from <process name>
<send command>
   ::= send (<expression>) to <process name>.
```

A receive command assigns the value of the message to the target variable and returns the value success as the result if it can receive the correponding message. It returns the value fail if it can not receive the message. A send command writes the value of

the expression in MSGB, writes the type of the expression in MSGT and changes the status of the partner of the rendezvous if it is suspended and the message corresponds.

5. Algorithm for Transformation (1)

Three algorithms for transformation are described in Chapter 5 through Chapter 7. In this chapter the first algorithm which can be applied to any kind of CSP programs are presented. Its informal description is given in Section 5.1 and the rigorous description of this algorithm is given in 5.2.

5.1 Overview of the First Algorithm

As stated in Chapter 3, every process of a CSP program is transformed into a coroutine and each command of each process is transformed into the same command except for input and output commands. A special routine called scheduler is added to those coroutines and it always keeps the status of every process and control the execution.

The scheduler repeats the following cycle until no process is executable.

(1) The scheduler chooses and activates a process which is executable;

(2) The chosen process executes its commands as far as it can proceed (the process can not proceed when it can not send or receive a message at an input or an output command, or when it is terminated.);

(3) The suspended process returns the control to the scheduler.

The mechanism of the message passing is as follows. In principle the process which reaches the place of the rendezvous first writes the type of the message (and the value of the message if the process is the sender of the message) in globally accessible space, then the other process which reaches the input or the output command checks the correspondence of the message,

$[]_{i=1..\ell}$ BE_i; $q_i?v_i$ --> CL_i,

where BE and CL stand for Boolean expressions and command list respectively.

(4) ELSE in

```
[ []i=1..ℓ BEi --> CLi
  ELSE     --> CL ]
```

is defined as ELSE $\equiv \bigwedge_{i=1}^{\ell}$ (not BE_i).

An output command in a process q which has the form

p!e

is transformed into the following commands:

send (e) to p; return (OW).

An input command which is not appearing in a guard of a process p and has the form

q?v

is transformed into the following command:

```
[receive (v) from q --> skip
 ELSE               --> return(IW); receive (v) from q
]
```

An alternative command with input guards in a process p which has the form

```
[[]i=1..ℓ BEi;qi?vi --> CLi
 []]i=ℓ+1..m BEi    --> CLi ]
```

is transformed into the following:

```
[[]i=1..ℓ BEi;receive(vi) from qi --> CLi
 []]i=ℓ+1..m BEi --> CLi
 []∧i=1ℓ (not BEi or not receive(vi) from qi); []i=ℓ+1..m BEi -->
    return (IWIG);
    [] i=1..ℓ receive(vi) from qi --> CLi    ]
```

A repetitive command with input guards in a process p which has the form

```
*[[]i=1..ℓ BEi;qi?vi --> CLi
 []]i=ℓ+1..m BEi     --> CLi ]
```

is transformed into the following:

```
flag: boolean; flag := true;
*[[]λ=1..ℓ  flag; BEλ; receive(vλ) from qλ --> CLλ
 []  []ι=ℓ+1..m flag; BEλ --> CLλ
 []flag;\ ∧ℓι=1 ( not BEι or not receive(vι) from qλ);
          ∧ℓι=1 ( not BEλ ) --> rflag: boolean; rflag := true;
   *[flag -->
      [rflag --> return (IWIG)
      []not rflag --> skip    ]
      [] []ι=1..ℓ  BEλ; receive(vλ) from qλ--> CLλ
      [] []ι=ℓ+1..m BEλ --> CLλ; rflag := false
      [] ELSE --> flag := false
]  ]  ]
```

At the end of each process, we insert

```
return (TE)
```

to inform its termination.

6. Algorithm for Transformation (2)

If a CSP program to be transformed has no input guard, a strategy called demand driven reduces the duty of the scheduler, since in those cases the pair of participants of a paticular communicatin is always determined uniquely.

If a process p reaches an input or an output command, the process transfer the control to the partner of the rendezvous. The activated process proceeds execution, until it reaches the place of the rendezvous and answer the request of the activator. It is possible that the called process also activates other processes, however, those called processes can not call any process which is already waiting for the partner to respond the request, since this indicates the presence of deadlock. Thus, if there does not arise any deadlock, the control must return to the first process p. In this way the execution proceeds until the process p chosen by the scheduler terminates and return the control to the scheduler. Thus, the number of the scheduler's choice of the process to activate decreases comaring with the first algorithm.

The transformation proceeds in the following manner.

(1) If an input command

```
q?v
```

appears, it is changed to

```
[receive v from q --> return (RE)
[not (receive v from q) -->
   call q (IW); receive v form q ]
```

(2) If an output command

```
q!e
```

appears, it is changed to

send (e) to q; resume q (OW)

(3) At the end of each process

return (TE)

is inserted to inform its termination.

7. Algorithm for Transformation (3)

In this chapter the third algorithm for transformatin is presented. CSP programs which satisfy certain conditions can be efficiently executable. The first section of this chapter shows the conditions to be satisfied, the second section explains the algorithm informally, and in the last section, rigorous rules for transformation are presented.

7.1 Conditions to be Satisfied

If a CSP program satisfies certain conditions stated in this section, the role of the scheduler is to only activate a special process called source process which leads the computation. In this section we first states the conditions to be satisfied, and then, describe the way of transformation.

Any program in CSP that satisfies three conditions stated below in terms of communication graph [19] and activation graph can be executed without arbitrary choice of the process by the scheduler. At first the communication graph Gc is defined as follows:

Definition 1 (communication graph Gc).

For a given CSP program, the COMMUNICATION GRAPH Gc is defined as follows:

(1) each process is a point of Gc, and

(2) if there is a communication (i.e. transfer of messages) from a process p to a prcess q, pq is contained in Gc as an arc.

Let P be the set of processes of the CSP program, or, in other words, the set of points of Gc, and let Ac be the set of arcs in Gc. We write as Gc=(P,Ac). The first condition to be satisfied is

as follows:

Condition 1.

Gc is acyclic.

By this condition a partial order $\preceq$ can be naturally induced into the set of points P of Gc. We shall define the order of $p,q \in P$ as

$$pq \in Ac \quad ==> \quad p \preceq q.$$

Suppose that an expression in CSP satisfies the above condition. The next condition we take into consideration is:

Condition 2.

There exists a process $s \in P$ such that for any $p \in P$, $p \preceq s$ or $s \preceq p$ holds.

The process which satisfies the above condition may not be unique. If there exists more than one process, we select an arbitrary one and fix it from now on.

Definition 2 (source process s).

We select a process which satisfies Conditin 2 and fix it. We call the process the SOURCE (PROCESS) of P, which will be denoted by s.

Another kind of graph called activation graph is now defined.

Definition 3 (activation graph Ga).

If Condition 1 and 2 are satisfied, the ACTIVATION GRAPH Ga=(Pa,Aa) of a CSP program can be defined as follows:

(1) Pa=P (the set of points is the same as that of Gc; thus we use P instead of Pa), and

(2) $pq \in Aa \iff qp \in Ac$ (if $q \preceq s$)

$pq \in Ac$ (if $s \preceq p$).

Note that indegree of the graph Ga of the source process s is $\emptyset$. The following proposition is proved easily.

Proposition 1.

Ga is acyclic.

By this proposition, a new partial order $(P, \overset{a}{\preceq})$ which is different from $(P, \preceq)$ is induced; it represents the order of the activation.

The last condition to be satisfied is stated in terms of the activation graph Ga as follows:

Condition 3.

Ga is an out-tree.

Remark.

The above condition is equivalent to the following one:

Condition 3'.

With respect to Ga, the indegree of the source s is $\emptyset$ and the indegrees of all other processes are 1.

For convenience, we will define several terms.

Definition 4 (producer and consumer).

A process p is said to be a PRODUCER (PROCESS) if $p \not\preceq s$, and is said to be a CONSUMER (PROCESS) if $s \not\preceq p$.

Definition 5 (parent, son, ancestry and descendant).

For each $p \in P$, we define the following:

(1) if $qp \in Aa$, q is said to be the parent of p (which is uniquely defined),

(2) if $pq \in Aa$, q is said to be a son of p,

(3) $\{q \in P \mid q \overset{Da}{\leq} p\}$ is said to be ancestry of p, and

(4) $\{q \in P \mid p \overset{Da}{\leq} q\}$ is said to be descendant of p.

7.2 Overview of the Third Algorithm

Those programs which satisfy above three conditions can be executed in the following manner.

(1) The shceduler activates the source process s.

(2) The source process makes all the producer processes be ready to send messages in the following way:

(2-1) it activates each of its sons that is a consumer, and then

(2-2) each of those sons also makes their sons be ready to send messages by activating them.

(2-2) proceeds until all the producers are activated and become ready to send messages to their parent processes.

(3) The computation proceeds by the leading of the source process preserving the condition that all the producer processes are always be ready to send messages unless they are terminated. (Note that the result of the input guards can be always determined since all the producers are ready to send messages.)

(4) When the source process is reached its end, it broadcasts its termination to all the consumers in the same manner as (2-2).

(5) All the consumers change their status to "terminated" and return the control to their parents.

(6) The source process gets the control again and returns it to the scheduler.

(7) The execution terminates.

7.3 Description of the Third Algorithm

Let P be the set of processes of a CSP program to be transformed. We transform a process $p \in P$ in the following manner.

(1) If $p \preceq s$, insert the following call commands at the top:

call q_1 (RE); ...; call q_M (RE),

where each q_λ is a son of p.

(2) A command which is not or does not contain any input or output command is not changed.

(3) An input command which does not appear in guard is changed as follows:

(3-1) if $p \preceq s$, and input command

q?v

is changed to

receive (v) from q; call q (RE),

and

(3-2) if $s \not\preceq p$, an input command

q?v

is change to

receive (v) from q; return (RE).

(4) An output command is changed as follows:

(4-1) if $p \not\preceq s$, and output command

q!e

is changed to

send (e) to q; return (OW),

and

(4-2) if $s \preceq p$, an output command

q!e

is changed to

send (e) to q; call q (OW).

(5) An alternative command with input guards in $p \preceq s$

$$[\!]_{i=1..\ell}\ BE_i;\ q_i?v_i \longrightarrow CL_i \quad [\!]_{i=\ell+1..m}\ BE_i \longrightarrow CL_i\ ]$$

is changed to

$$[\!]_{i=1..\ell}\ BE_i;\ \text{receive } (v_i) \text{ from } q_i \longrightarrow \text{call } q_i\ (RE);\ CL_i \quad [\!]_{i=\ell+1..m}\ BE_i \longrightarrow CL_i\ ].$$

If $s \not\preceq p$, the command

call q (RE)

in the above is replaced by

return (RE).

(6) An repetitive command with input guards in $p \preceq s$

$$*[[\!]_{i=1..\ell}\ BE_i;\ q_i?v_i \longrightarrow CL_i \quad [\!]_{i=\ell+1..m}\ BE_i \longrightarrow CL_i\ ]$$

is changed to

$$*[[\!]_{i=1..\ell}\ BE_i;\ \text{receive } (v_i) \text{ from } q_i \longrightarrow \text{call } q_i\ (RE);\ CL_i \quad [\!]_{i=\ell+1..m}\ BE_i \longrightarrow CL_i\ ].$$

If $s \not\preceq p$, the command

call q (RE)

in the above is replaced by

return (RE).

(7) If $p \not\preceq s$, the following command is inserted in the end:

return (TE).

(8) If $s \preceq p$, the following command is inserted in the end:

call q_1 (TE); ...; call q_m (TE); return (TE),

where each q_i is a son of p.

The following is an example of this algorithms of

transformation. A CSP program shown in 2.1.3 of 8-Queens Problem is transformed as follows, where TRY(9) is chosen as the source process.

```
  [TRY(i:1..8)::
   A:(1..8) boolean; B:(2..16) boolean;
   C:(-7..7) boolean; X:(1..8) integer;
   call TRY(i-1) (RE);
   *[receive( (A,B,C,X) ) from TRY(i-1) -->
      call TRY(i-1) (RE);
      j:integer; j:=1;
      *[j<=8; A(j); B(i+j); C(i-j) -->
         X(i)   := j;
         A(j)   := false;
         B(i+j) := false;
         C(i-j) := false;
         send( (A,B,C,X) ) to TRY(i+1);
         return;
         A(j)   := true;
         B(i+j) := true;
         C(i+j) := true;
         j := j+1        ]]
||
TRY(0)::
   A:(1..8)boolean; B:(2..16) boolean;
   C:(-7..7)boolean; X:(1..8) integer;
   i:integer;
   i:=1; *[i<=8  --> A(i):=true; i:=i+1];
   i:=2; *[i<=16 --> B(i):=true; i:=i+1];
   i:=-7;*[i<=7  --> C(i):=true; i:=i+1];
   i:=1; *[i<=8  --> X(i):=0;    i:=i+1];
   send( (A,B,C,X) ) to TRY(1);
   return (OW)
||
TRY(9)::
   A:(1..8) boolean; B:(2..16) boolean;
   C:(-7..7) boolean; X:(1..8) integer;
   call TRY(8) (RE);
   *[receive( (A,B,C,X) ) from TRY(8) -->
      call TRY(8) (RE);
      send( X ) to PRINT; call PRINT (OW) ]
]
```

8. Comparison with Related Works

In this chapter, we compare the method of transformation presented in this thesis with other related works: T. Katayama's work [44,45] of translation of attribute grammars [48] into procesures, A.N. Habermann and I.R. Nassi's work [33] of Ada tasks [78] into procedures and T. Hagino's work [28] of transformation of CSP programs into sequental programs.

8.1 Comparison with Katayama's Method

Katayama's method for translation of attribute grammers into procedures is designed for effective evaluation of value of attributes and its outline is as follows [44].

Let X be a nonterminal symbol of an attribute grammar G=(Vn,Vt,P,S), where Vn, Vt, P and S are a set of nonterminal symbols, a set of terminal symbols, a set of production rules and the start symbol respectively, and let s be a synthesized attribute of X. We associate with each pair (X,s) a procedure of the form

R (Vl, ..., Vm, T; V),

where Vl, ..., Vm are parameters corresponding to the inherited attributes which are necessary to evaluate s, T is an derivation tree and V is a parameter which corresponds to s. Thus parameters to the left (right) of ';' are input (output) parameters. This procedure R is intended to evaluate the synthsized attribute s when supplied with the values of attributes which are necessary for the evaluation of s and a derivation tree T. When we are given the initial derivatin tree TØ and a synthesized attribute sØ of the initial symbol S, we begin evaluation of sØ by executing the procedure call statement

call R1 (T0;V0),

where V0 is a variable corresponding to s0. The execution proceeds, recursively calling the procedures which are necessary to evaluate sunthesized attributes, until the desired value V0 is computed.

Attribute grammar systems, as well as CSP, can be considered useful means of description of algorithms. Thus, motivations of Katayama's method and the method presented in this thesis are quite similar. Since an attribute grammar is an augmented form of a context free grammar, there are strong relations among symbols by means of production rules of the grammar, and usually those grammers contain recursiveness by nature. Thus, it is natural to evaluate the values of the attributes by activating mutually recursive procedure. Among processes of general CSP programs, however, no such relation can be found. Hence, the method taken by Katayama can not be applied to CSP programs directly.

8.2 Comparison with Habermann and Nassi's Method

The Habermann-Nassi Method for efficient implementation of Ada tasks is a method which transforms the calls of entries by tasks into simple procedure calls. Its outline is as follows [77].

In their method, each task with entries is transformed into a procedure and each entry is embedded as a block of the procedure. However, it is not sufficient to replace the body of an entry by the body of a corresponding procedure, since in that case demands from other tasks which originally have to be delayed will succeed at once. To avoid this error, we prepare locks for each entry and control them as if the control proceeds

sequentially (coroutine like way) through the task. We show an simple example which is adopted from [77], where four semaphores are introduced to control the execution.

```
task body T is
begin
   loop
      accept E1 do
         - 1 -
      end E1;
      accept E2 do
         - 2 -
      end E2;
      accept E3 do
         - 3 -
      end E3;
   end loop;
end T;
```

Before Transformation

```
S1 := 1;
S2,S3 := Ø;
SØ := Ø; P(SØ);
E1: P(S1);
   - 1 -
    V(S2);
    return;
E2: P(S2);
   - 2 -
    V(S3);
    return;
E3: P(S3);
   - 3 -
    V(S1);
    return;
```

After Transformation

The mechanisms of rendezvous of two tasks in Ada are strongly influenced by CSP, however, the roles of tasks involved in a rendezvous are not symmetric; a task calls and the other task accepts the request when it reaches an appropriate entry. There is no need to write explicitly the name of the task with which an entry is concerned. These phenomena enable the method stated above effective. As for CSP, the same method can not be applied directly, since there is no caller-callee relation between any two processes.

8.3 Comparison with Hagino's Method

In [28], Hagino completely changes CSP programs into sequential programs, i.e. there is only one routine after the transformation. The transformation proceeds in the way shown below, where

(<command list>) ⊕ (<command list>)

denotes the result of the transformation.

A CSP program

```
[P:: x:=1; Q!x; Q?x
Q:: P?y; y:=y+2; P!y ]
```

will be transformed in the following way.

```
(x:=1; Q!x; Q?x) ⊕ (P?y; y:=y+2; P!y)
x:=1; (Q!x; Q?x) ⊕ (P?y; y:=y+2; P!y)
x:=1; y:=x; (Q?x) ⊕ (y:=y+2; P!y)
x:=1; y:=x; y:=y+1; (Q?x) ⊕ (P!y)
x:=1; y:=x; y:=y+1; x:=y.
```

This method of trnsformation is proposed for verifications of CSP porgrams and the efficiency of transformation is of little concern. In fact, the procedures have to be executed are complicated and the meanings of the result of the transformation is almost impossible to understand. On the other hand, the method taken in this thesis preserves the structure of the processes and the rules of transformation are simple and efficiently executable.

9. Conclusion

As a way to solve the problem of effective scheduling of processes of CSP, three algorithms for transformation of CSP programs into coroutines are presented; The first algorithm is simple and can be applied to general CSP programs but not efficient, the other two algorithms can only be applied to restricted classes of CSP programs but more efficient than the first one.

Comparisons with three other related works are also presented. The method taken in this thesis has similarity with each of those works, but methods of those works can not be applied directly to the problem of efficient scheduling of processes of CSP.

Acknowledgement

The author wishes to thank Prof. T. Tokuda for his kind and helpful advice and encouragements, and also, wishes to thank the members of the faculty of Yamanashi University including Prof. M. Arisawa, Mr. M. Iuchi and Prof. T. Yoshizawa for their advice and encouragements.

References

[1] Apt, K.R. et al.: "A Proof System for Communicating Sequential Processes," ACM Trans. on Prog. Lang. and Sys., Vol. 2, No. 3 (Jul. 1980), pp. 359-385.

[2] Arsac, I. et al.: "Some Techniques for Recursion Removal for Recursive Functions," ACM Trans. on Prog. Lang. and Sys., Vol. 4, No. 2 (Apr. 1982), pp.295-322.

[3] Ashcroft, E.A. et al.: "Lucid, a Nonprocedural Language with Iteration," Comm. ACM, Vol. 20, No. 7 (Jul. 1977), pp. 519-526.

[4] Bernstein, A.J.: "Output Guards and Nondeterminism in Communicating Sequential Processes," ACM Trans. on Prog. Lang. and Sys., Vol. 2, No. 2 (Apr. 1980), pp. 234-238.

[5] Bezivin, J. et al.: "Another View of Coroutines," ACM SIGPLAN Notices, Vol. 13, No.5 (May. 1978), pp. 23-25.

[6] Bird, R.S.: "Notes on Recursion Elimination," Comm. ACM, Vol. 20, No. 6 (Jun. 1977), pp. 434-439.

[7] Birtwistle, G.M. et al.: SIMULA BEGIN, Lund, 1981.

[8] Clarke, E.M.: "Proving Correctness of Coroutines Without History Variables," Acta Informatica, No. 13 (1980), pp. 169-188.

[9] Clint, M.: "Program Proving: Coroutines," Acta Informatica, No. 2 (1973), pp. 50-63.

[10] Clocksin, W.F. et al.: Programming in prolog, Springer-Verlag, 1981.

[11] Conway, M.E.: "Design of a Separable Transition Diagram Compiler," Comm. ACM, Vol. 6, No. 7 (Jul. 1963), pp. 122-134.

[12] Dahl, O.J. et al.: Structured Programming, Academic Press, 1972.

[13] Dijkstra, E.W. et al.: "Goto Statement Considered Harmful," Comm. ACM, Vol. 11, No. 5 (Mar. 1968), pp. 147-148.

[14] Dijkstra, E.W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Comm. ACM, Vol. 18, No. 8 (Aug. 1975), pp. 453-457.

[15] Dijkstra, E.W.: A Discipline of Programming, Prentice-Hall, 1976.

[16] Dijkstra, E.W. et al.: "Termination Detection for Diffusing Computations," Information Processing Letters, Vol. 11, No. 1 (Aug. 1980), pp. 1-4.

[17] Doi, N.: "Processes and Their Synchronization - Rendezvous," bit, Vol. 14, No.,4 - 7 (1982).

[18] Floyd, R.W.: "Nondeterministic Algorithms," J. ACM, Vol. 14, No. 4 (Oct. 1967), pp. 636-644.

[19] Francez, N.: "Distributed Termination," ACM Trans. on Prog. Lang. and Sys., Vol. 2, No. 1 (Jan. 1980), pp. 42-55.

[20] Furukawa, K. et al.: "On Verification of Prolog Coroutine Interpreter," Proc. of 1st Meeting of WGSF of IPSJ, 1982.

[21] Furuya, T.: "Concurrent Programming by High Level Language," Joho-Shori, Vol. 21, No. 9 (Sep. 1980), pp.949-958.

[22] Gentleman, W.M.: "A Portable Coroutine System," Information Processing 71, pp. 419-424, North-Holland Publishing Company (1972).

[23] Gries, D.: "Some Ideas on Data Types in High-Level Languages," Comm. ACM, Vol. 20, No.6 (Jun. 1977), pp. 414-420.

[24] Gries, D.(Ed.): Programming Methodology, A Collection of Articles by members of IFIP WG 2.3, Springer-Verlag, 1978.

[25] Gries, D.: The Science of Programming, Springer-Verlag, 1981.

[26] Griswold, R. E. et al.: "Generators in Icon," ACM Trans. on Prog. Lang. and Sys., Vol. 3, No. 2 (Apr. 1981), pp. 1

[27] Grune, D.: "A View of Coroutines," ACM SIGPLAN Notices, Vol. 12, No. 6, (Jul. 1977), pp. 75-81.

[28] Hagino, T.: "Verification of Communicating Sequential Processes," Proc. of 2nd Meeting of WGSF of IPSJ, 1982.

[29] Hansen, P.B.: "The Programming Language Concurrent Pascal," IEEE Trans. on Soft. Eng., Vol. SE-1, No. 2 (Jun. 1975), pp. 199-207.

[30] Hansen, P.B.: "Distributed Processes: A Concurrent Programming Concept," Comm. ACM, Vol. 21, No. 11 (Nov. 1978), pp. 934-941.

[31] Hansen, D.R. et al.: "The SL5 Procedure Mechanism," Comm. ACM, Vol. 21, No. 5 (May. 1978), pp. 392-400.

[32] Harary, F.: Graph Theory, Addison-Wesley, 1969.

[33] Harbermann, A.N. et al.: "Efficient Implementation of Ada Tasks," Dept. of Computer Science, Carnegie-Mellon Univ. (1980).

[34] Hehner, E.C.R.: "do Considered od: A Contribution to the Programming Calculus," Acta Informatica, Vol. 11 (1979), pp. 287-304.

[35] Henderson, P.: Functional Programming Application and Implementation, Prentice-Hall International, 1980.

[36] Hewitt, C.E. et al.: "Towards a Programming Apprentice," IEEE trans. on Soft. Eng., Vol. SE-1, No. 1 (Mar. 1975), pp. 26-45.

[37] Hikita, T. et al.: "Introduction to Ada," Nikkei Electronics, Dec. 21 (1981), pp. 130-162.

[38] Hoare, C.A.R.: "An Axiomatic Approach to Computer Programming," Comm. ACM, Vol. 12, No. 10 (Oct. 1969), pp. 576-580, 583.

[39] Hoare, C.A.R.: "Proof of Correctness of Data Representations," Acta Informatica, Vol. 1 (1972), pp. 271-281.

[40] Hoare, C.A.R.: "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 10 (Oct. 1974), pp. 549-557.

[41] Hoare, C.A.R.: "Communicating Sequential Processes," Comm. ACM, Vol. 21, No. 8 (Aug. 1978), pp. 666-677.

[42] Jacobsen, T.: "Another View of Coroutines," ACM SIGPLAN Notices, Vol. 13, No. 4 (Apr. 1978), pp. 68-75.

[43] Kahn, G. et al.: "Coroutines and Networks of Parallel Processes," Information Processing 77, North Holland (1977), pp. 993-998.

[44] Katayama, T.: "Translation of Attribute Grammar into Procedures," Tech. Rep. CS-K8001 (1980), Dept. of Comp. Sci., Tokyo Inst. of Tech.

[45] Katayama, T.: "HFP: A Hierarchical and Functional Programming Based on Attribute Grammar," Proc. of 5th Int. Conf. on Soft. Eng., 1981.

[46] Kieburtz, R.B.: "Comments on Communicating Sequential Processes," ACM Trans. on Prog. Lang. and Sys., Vol. 1, No. 2 (Oct. 1979), pp. 218-225.

[47] Kimura, I. et al.: Theory of Expressions of Algorithms, Iwanami, 1982.

[48] Knuth, D.E.: "Semantics of Context Free Languages," Math. Syst. Theory, J.2 (1968), pp. 127-145, Correction, Math. Syst. Theory, J.5 (1971), pp. 95-96.

[49] Knuth, D.E.: The Art of Computer Programming, Vol. 1, Addison Wesley, 1969.

[50] Levin, G.M. et al.: "A Proof Technique for Communicating Sequential Processes," Acta Informatica, Vol. 15 (1981), pp. 281-302.

[51] Lewis, B: "Further Comments on 'A View of Coroutines'," ACM SIGPLAN Notices, Vol. 13, No. 7 (Jul. 1978), pp. 31-33.

[52] Lindstrow, G.: "Referencing and Retention in Block-Structured Coroutines," ACM Trans. on Prog. Lang. and Sys., Vol. 3, No. 3 (Jul. 1981), pp. 263-292.

[53] Lindstrow, G.: "Backtracking in a Generalized Control Setting," ACM Trans. on Prog. Lang. and Sys., Vol. 2, No. 1 (Jul. 1979), pp. 8-26.

[54] Liskov, B.H. et al.: "Specification Techniques for Data Abstractions," IEEE Trans. on Soft. Eng., Vol. SE-1, No. 1 (Mar. 1975), pp. 7-19.

[55] Liskov, B.H. et al.: "Abstraction Mechanisms in CLU," Comm. ACM, Vol. 20, No. 8 (Aug. 1977), pp. 564-576.

[56] Liskov, B.H. et al.: CLU Reference Manual, Lecture Notes in Computer Science 114, Springer-Verlag, 1981.

[57] Lynning, E.: "Letter to editor," ACM SIGPLAN Notices, Vol. 13, No. 2 (Feb. 1978), pp. 12-14.

[58] Marlin, C.D.: "Coroutines and Return Addresses (Letter to editor)," ACM SIGPLAN Notices, Vol. 13, No. 9 (Sep. 1978), pp. 19-20.

[59] Marlin, C.D.: Coroutines, Lecture Notes in Computer Science 95, Springer-Verlag, 1980.

[60] Misra, J. et al.: "Termination Detection of Diffusing Computations in Communicating Sequential Processes," ACM Trans. on Prog. Lang. and Sys., Vol. 4, No. 1 (Jan. 1982), pp. 37-43.

[61] Musha, H. et al.: "Coroutine Ratfor - Its Implementation and Usage," Proc. of 24th Conf. of IPSJ, pp. 229-230, 1982.

[62] Nakata, I.: "Programming with Streams of Data," unpublished paper (1982).

[63] Noel, J.G.: "Letter to editor," ACM SIGPLAN Notices, Vol. 12, No. 12 (Dec. 1977), p. 23.

[64] Pauli, W. et al.: "Coroutine Behaviour and Implementation," Software-Practice and Experience, Vol. 10 (1980), pp. 189-204.

[65] Pratt, T. W.: Programming Languages: Design and Implementation, Prentice-Hall Inc., 1975.

[66] Ritchie, D.M. et al.: "The UNIX Time-Sharing System," Comm. ACM, Vol. 17, No. 7 (Jul. 1974), pp. 365-375.

[67] Roper, T.J. et al.: "A Communicating Sequential Process Language and Implementation," Software-Practice and Experience, Vol. 11 (1981), pp. 1215-1234.

[68] Sassa, M. et al.: "Stream Functions and Their Implementation," Proc. of 23rd Conf. of IPSJ, pp. 175-176, 1981.

[69] Sassa, M. et al.: "Programming with Streams," Proc. of 25th Conf. of IPSJ, pp. 257-258, 1982.

[70] Sahara, M. et al.: "Implementation of Stream Functions by Coroutines," Proc. of 24th Conf. of IPSJ, pp. 227-228, 1982.

[71] Schneider, F.B.: "Synchronization in Distributed Programs," ACM Trans. on Prog. Lang. and Sys., Vol. 4, No. 2 (Apr. 1982), pp. 125-148.

[72] Shaw, M. et al.: "Abstraction and Verification in Alphand: Defining and Specifying Iteration and Generators," Comm. ACM, Vol. 20, No. 8 (Aug. 1977), pp. 553-564.

[73] Skordalakis, E. et al.: "Coroutines in Fortran," ACM SIGPLAN Notices, Vol. 13, No. 9 (Sep. 1978), pp. 76-84.

[74] Staunstrup, J: "Message Passing Communication Versus Procedure Call Communication," Software-Practice and Experience, Vol. 12 (1982), pp. 223-234.

[75] Stevens, W.P.: "How Data Flow Can Improve Application Development Productivity," IBM Systems Journal, Vol. 21, No. 2 (1982), pp. 162-178.

[76] Togawa, H.: "Introduction to Simula," bit, Vol. 11, No. 8, pp. 28-34, No. 9, pp. 42-49, No. 10, pp. 92-97 (1979).

[77] Tokuda, T.: "Notes on Ada Implementability Issues," Joho-Shori, Vol. 21, No. 3 (Mar. 1980), pp. 226-232.

[78] U.S. DoD: Reference Manual for Ada Programming Language, 1980.

[79] Vanek, Z.I. et al.: "Hierarchical Coroutines: A Mechanism for Improved Program Structure," Proc. of 4th Int. Conf. on Soft. Eng., pp. 274-285, 1979.

[80] Wang, A. et al.: "Coroutine Sequence in a Black Structured Environment," BIT, Vol. 11 (1971), pp. 425-449.

[81] Ward, S.A. et al.: "A Syntactic Theory of Message Passing," J. ACM, Vol. 27, No. 2 (Apr. 1980), pp. 365-383.

[82] Wirth, N.: Algorithm + Data Structures = Programs, Prentice-Hall, 1976.

[83] Yamamoto, K.: "SIMULA," Joho-Shori, Vol. 22, No. 6 (Jun. 1981), pp. 477-482.

[84] Yamano, K. et al.: "Applicative Communication Function for Parallel Programming," Proc. of 3rd Meeting of WGSF of IPSJ, 1982.

[85] Yonezawa, A.: "A Tutorial on ACTOR Theory," Joho-Shori, Vol. 20, No. 7 (Jul. 1979), pp. 580-589.

[86] Yourdon, E. et al.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall Inc., 1979.

[87] Yuasa, T. et al.: "Programming Environment for Modular Programming," Joho-Shori, Vol. 23, No. 5 (May. 1982), pp. 433-441.