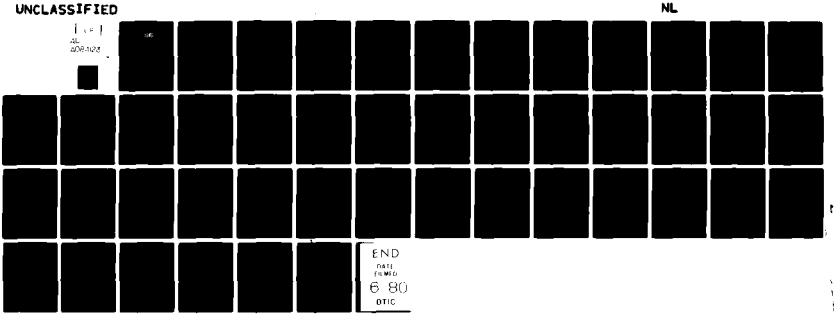
AD-A084 123 OXFORD UNIV (ENGLAND) COMPUTING LAB F/G 9/2
BCSP: A SMALL LANGUAGE FOR PARALLEL PROCESSING.(U)
FEB 80 C M HOLT N00014-79-C-0041
UNCLASSIFIED NL



END
DATE FILMED
6 80
DTIC

Report N00014-79-G-0041

BCSP : A SMALL LANGUAGE FOR PARALLEL PROCESSING

Christopher M. Holt
Programming Research Group
Oxford University Computing Laboratory
45 Banbury Road
Oxford OX2 6PE
ENGLAND

February 1980

Annual Report for Period 1 April 1979 - 31 March 1980

Prepared for

Information Systems
Mathematical and Information Sciences Division
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

MIL-STD-847A
31 January 1973

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: N00014-79-G-0041	2. GOVT ACCESSION NO.: AD-A084123
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): BCSP: A SMALL LANGUAGE FOR PARALLEL PROCESSING,	5. TYPE OF REPORT & PERIOD COVERED: ANNUAL, 4/79-3/80
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Christopher M. Holt	8. CONTRACT OR GRANT NUMBER(s): N00014-79-G-0041
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Oxford University Computing Laboratory, Programming Research Group, 45 Banbury Road, Oxford OX2 6PE England	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Information Systems, Math. & Information Sciences Division, ONR, 800 N. Quincy St., Arlington VA 22217	12. REPORT DATE: Feb [redacted] 80
	13. NUMBER OF PAGES: 42
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 12) 46	15. SECURITY CLASS. (of this report):
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

9) Annual rept. 1 Apr 79-31 Mar 80

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming languages, Parallel programming, Input, Output, Machine architecture, Abstract machines

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

BCSP, a parallel programming language, is described. It is based on Hoare's Communicating Sequential Processes, but uses tail recursion for loops and linked ports for communication. An abstract parallel machine is described that can be used to implement BCSP; the machine description assumes an arbitrary number of processors.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Figure 6. Report Documentation Page.

S/N 393448

Summary

The language BCSP is based on Hoare's Communicating Sequential Processes, but has a few significant differences:

i) Communication between processes uses linked ports, which are declared and linked before the parallel processes are executed;
ii) Repetition uses a tail-recursive syntax, rather than a special repetitive structure;
iii) Termination must be explicit, rather than implicit; and
iv) There is a primitive within the language that causes simulated time to pass.

Examples written in BCSP are provided to give a feel for the way programs are written and understood.

The language is implemented on an abstract parallel machine; the architecture of this machine constitutes the second part of the report. The machine has instructions that:

i) create new processes dynamically;
ii) communicate between parallel processes;
iii) wait for simulated time to pass, and obtain the current value of simulated time; and
iv) accept values from an input file, and send values to an output file.

The machine is described both informally, in English, and semi-formally, in an extended version of BCSP.

Preface

Many thanks must be given to C.A.R. Hoare, who continually tried to dissuade the author from following the tempting paths of unnecessary complexity, and greatly contributed to the clarity of this report.

SECTION I. A DESCRIPTION OF BCSP

BCSP is a procedural language that includes communication between parallel processes as one of its primitives. The language is currently too small to be used as a programming tool; it is intended only for research into parallelism and communication. The reader is assumed to be familiar with the basic concepts of parallel programming and with CSP. This description is divided into the following sections:

A. Language Summary -
   A short, informal explanation, comparing the language with CSP;
B. The Language BCSP -
   A more formal description of the syntax and concepts of the language;
C. Examples -
   A set of small example programs, selected to show how BCSP may be applied to solve problems; and
D. The Machine Interface -
   How to run a BCSP program on a UCSD Pascal system.

A. Language Summary

In order to give the reader a feel for BCSP, a few examples are presented that illustrate how it differs from CSP.

1. Some Syntactic Differences

A program is presented that receives an integer from the input data file, using the standard port Read, and sends the absolute value of that integer to the output data file, using the standard port Write. This is written in CSP as:

```
[ TheInputValue : Integer ;
  Read ? ( TheInputValue ) ;
  [ TheInputValue  < 0 -> Write ! ( -TheInputValue )
  ¦ TheInputValue ~< 0 -> Write ! (  TheInputValue )
] ]
```

This is here written as follows:

```
[; Read ? The input value : Int
 ; [¦ The input value  < 0 -> Write ! -The input value
   ¦ The input value ~< 0 -> Write !  The input value
   ¦]
 ;]
```

Notes :

i) Identifiers may consist of several words. Spaces and new lines may be used

to separate words; separators between words are used in distinguishing identifiers, though the number of consecutive separators is not significant. Upper case and lower case letters are treated as different characters.

```
< Identifier >    ::= < Word > | < Word > < Separators > < Identifier >

< Word >          ::= < Alphanumeric > | < Alphanumeric > < Word >

< Alphanumeric >  ::= 0 | ... | 9 | A | ... | Z | a | ... | z

< Separators >    ::= < Separator > | < Separator > < Separators >

< Separator >     ::= < Space > | < New line >
```

This definition of identifier is used throughout.

ii) The symbol "¦" is used to separate alternatives; the fat bar character of pure CSP is not used.

iii) The separators between elements of structures occur immediately within the structure delimiters "[" and "]", as well as between the elements.

iv) Variables may be declared and initialized in the same input command, as suggested within the discussion in the CSP paper.

v) Int is a type, consisting of the integer values from - MaxInt to MaxInt (a defined constant). The only other type is Bool, which consists of the values true and false (represented by the constants True and False).

vi) "Read" is the name of a port which is initially associated with the input file of a program.

vii) "Write" names a port that is initially associated with the output file of a program.

viii) The symbol "~<" is used to mean "not less than" (i.e. "greater than or equal to"). Similarly, "~=" is used for "not equals" and "~>" is used for "not greater than".

2. Loops

The above program can be modified to produce the absolute values of a series of values, instead of just one, by transforming it into a loop. In CSP, this would be:

```
*[ TheInputValue : Integer
 ; Read ? ( TheInputValue ) ->
   [ TheInputValue  < 0 -> Write ! ( -TheInputValue )
   ¦ TheInputValue ~< 0 -> Write ! (  TheInputValue )
 ]   ]
```

The BCSP version is:

Absolute value

```
:: [| Read ? The input value : Int
     -> [; [| The input value  < 0 -> Write ! -The input value
           | The input value >= 0 -> Write !  The input value
           |]
        ; Absolute value
        ;]
   | End read $ -> End write $
   |]
```

Notes:

i) As in CSP , process names (e.g. "Absolute value") are defined by following them with "::" (which may be read "is defined as"), and then the process. No parameters are used.

ii) Whereas CSP has a repeat structure, with the "*", BCSP uses tail recursion to loop. Tail recursion has the appearance and behavior of a recursive call; such a call must be the last command to be executed within a named process.

iii) "$" is used for synchronization by a signal. This is a symmetric communication; it is like the usual "!" and "?" communication, except that no value is sent, and so no expression or target follows the "$". A "$" occurs on both sides of a symmetric communication.

iv) The synchronization using the tag "End read" succeeds only when the source file is empty. The tag "End write" is used to signal the closing of the output file.

v) CSP allows implicit termination: the loop ends when the source process has terminated. BCSP requires explicit termination (here using a distinct communication, "End read $").

3. User-defined Communication

The next program to be considered contains two parallel processes. The first sends the number 4 to the second. This is written in CSP as

```
[  Sender :: Receiver ! ( 4 )
|| Receiver :: X : Integer ;
              Sender ? ( X )
]
```

This has three possible analogues in BCSP. The first, and simplest, is

```
[>> Send ! Int <-> Receive ? Int
 -- [& Send ! 4
    & Receive ? X : Int
    &]
 <<]
```

Notes :

i) This BCSP structure is called a declaration block. The symbol ">>" indicates that a declaration follows; the symbol "--" precedes the main process of the block; and the symbol "<<]" ends the block.

ii) Parallel structures use the separator "&" instead of "||". As with the other kinds of structures, the separator occurs immediately within the brackets.

iii) The symbol "<->" indicates that the ports declared on either side of it are to be linked together. These ports may be used for communication between parallel processes. Ports must be linked either in input-output pairs or in symmetric pairs. The scope of a port is from its declaration to the end of its declaration block.

iv) A port is specified by an identifier, called a tag, and a direction. If the direction is either input ("?") or output ("!") then a type must be specified as well; if the direction is symmetric ("$") no type is necessary. The same tag may be used for more than one port with no ambiguity if each such port has a different direction. For example, "Send ! Int" specifies a port with tag "Send", direction "!", and type "Int"; "Send $" specifies a different port, with tag "Send" and direction "$".

4. Restricted Ports

The second BCSP program that is equivalent to the CSP program above is

```
[>> Sender
 >> Receiver
 >> Sender ( Send ! Int ) <-> Receiver ( Receive ? Int )
 -- [& Sender :: Send ! 4
    & Receiver :: Receive ? X : Int
    &]
 <<]
```

Notes :

i) Process names, as well as port links, may be declared in declaration blocks. Such a declaration consists of writing the process name. A declared process has scope running from its point of declaration to the end of the block within which it was declared.

ii) Ports may be restricted to individual processes; this is indicated in the declaration by enclosing the port in parentheses and preceding this by the process name. A port restricted in this way may be used only within the named process. More than one port may use the same tag and direction, if each such port is restricted to a different named process.

5. Process Calls

A third BCSP program that can send a value from one process to another is

```
[>> Sender
 >> Receiver
 >> Sender ( Send ! Int ) <-> Receiver ( Receive ? Int )
  - [& Sender
    & Receiver
    &]
 << Sender :: Send ! 4
 << Receiver :: Receive ? X : Int
 <<]
```

Notes :

i) The definition of a process may be deferred until after the main process of a block, if it has been declared in the block heading. Each such definition is preceded by a "<<".

ii) Each name may be used within the main process as a call. A call causes the process named by the call to be executed. At present, a process may only be called once from without its body, as well as tail-recursively from within itself.

6. Simulated time

To model real-world situations, it is necessary to be able to simulate the passage of time; for this, a duration is used. A program that prints the current time, waits for 5 units of time, and then prints the new time is

```
[; Write ! Time
 ; 5
 ; Write ! Time
```

Notes:

i) The global variable Time may be accessed to obtain the total elapsed simulated time since the program began. One may not assign a value to Time.

ii) A process consisting of an expression is called a duration. A duration may be integer or boolean. If it is integer, as in this case, then that number of units of simulated time pass before the process terminates. If it is boolean, the process is an assertion: a "true" duration is equivalent to a duration of 0, and a "false" duration is equivalent to a duration of infinity (it never terminates). The guard of an alternative structure may contain a boolean duration, but not an integer duration.

7. Other Differences

i) A constant is declared in a declaration block, by following an identifier with a colon and the value to be associated with the identifier; the value may be any expression (Eg. X : 73). The scope of a constant is the rest of the declaration block within which it was declared.

ii) A variable may be declared in a declaration block, by following an identifier with a colon and the type to be associated with the identifier. Such a variable may not be initialized when it is declared. Its scope is the rest of the declaration block within which it was declared.

iii) Assignments may contain variable declarations (Eg. X : Int := 17).

iv) "Pass" names a predefined process that takes no time and does nothing (skip). There is also a process "Fail" that never succeeds (abort).

iv) The symbol "/" indicates integer division, and the symbol "\" is used to mean modulo; thus, 5 / 3 = 1 and 5 \ 3 = 2. Generally, (A / B) * B + A \ B = A.

v) The symbol "/\" means "And"; "\/" means "Inclusive Or".

B. The Language BCSP

< Program > ::= < Process >

The fundamental unit of execution in BCSP is called a process. A process is specified by its behavior (its interactions with its environment). A process is either a primitive, which is an elementary operation, or a structure of processes, which combines sub-processes together to form a more complicated pattern of behavior. A definition names a process; a named process can be executed when its name is used in a call.

< Process > ::= < Primitive > | < Structure > | < Definition > | < Call >

1. Primitives

A primitive process executes a single, indivisible operation, modifying some element of its environment. An assignment assigns a value to a variable. A synchronization uses a port to interact with a synchronization in another, parallel process. An output uses a port to send a value to an input in another process; the input uses its port to accept that value and assign it to a variable. A duration waits for some length of simulated time before terminating.

< Primitive > ::= < Assignment > | < Synchronization >
| < Duration > | < Input > | < Output >

1.a. Assignments

An assignment is a primitive process that evaluates an expression, and then sets a variable to that value. The variable being set is called a target. A target may be a variable declaration; then, the type of the variable must be specified.

< Assignment > ::= < Target > := < Expression >

< Target > ::= < Variable > | < Variable declaration >

< Variable declaration > ::= < Identifier > : < Type >

Examples:

```
Mike is married := True
Some random number := X * 5 - 11
Top of the stack : Int := 0
Alison is married : Bool := Mike is married /\ Mike and Alison are a pair
```

1.b. Synchronizations

A synchronization is a signal between two processes, used to inform each of the progress of the other. It specifies a port, giving a tag and a direction ("S" indicates synchronize). Whichever process is ready to synchronize first waits until the other is also ready. The synchronization using the port with

the tag "End read" succeeds when the input file is empty; that synchronization with tag "End write" closes the output file.

< Symmetric communication > ::= < Tag > $

< Tag > ::= < Identifier >

Examples:

```
Ready for next value $
P $
V $
End read $
```

1.c. Output

An output is used to calculate a value in one process, and send it to a variable in another. The output matches an input in the other process, and they are executed at the same time; if one side is not ready, the other side waits.

An output consists of an expression, denoting the value to be sent, and a port through which the value is to be sent. This port is specified by its tag (an identifier), its direction ("!" is used for send), and the type of the value to be sent. The tag "Write" may be used to send values to the standard output file.

< Send > ::= < Tag > ! < Expression >

Examples :

```
The square of X is ! X * X
Done with test ! True
Write ! Result of program
```

1.d. Input

An input specifies a port (by indicating a tag and the direction "?") and a target (whose type further restricts the port). The variable of the target is assigned the value received through the port. The tag "Read" is used to obtain integer values from the standard input file. As in the assignment, the target may be a variable declaration.

< Receive > ::= < Tag > ? < Target >

Examples :

```
Get ? Length of a piece of string
Read ? Parameter of program
Ready to proceed ? Q : Bool
```

1.e. Durations

A duration indicates that simulated time passes between its beginning and its termination. The simulated time is set to 0 at the beginning of a program, and is incremented only when every process of the program is waiting for time to pass. Then, the smallest duration is completed, and each other duration is decremented by the time that has elapsed. Although assignments take no apparent time, a synchronization, an output, and an input may all take some time to execute, if one side has to wait for the other.

If the value of a duration is a non-negative integer, then the duration causes execution to wait that number of units of simulated time before continuing with execution. If the value is a non-positive integer or the boolean true, then it is equivalent to a duration of 0, and execution continues immediately. If the value is the boolean false, then it is equivalent to a duration of infinity: it never terminates.

< Duration > ::= < Expression >

Examples :

```
3
X = 45
```

The current amount of elapsed time may be obtained, using a variable called Time; this variable may not be used as a target in an assignment or an input.

Example :

```
X : Int := Time
```

2. Structures

A structure combines processes together to form a larger process. It is executed by executing one or more of its constituents.

< Structure > ::= < Sequence > | < Parallel structure >
| < Alternative structure > | < Declaration block >

2.a. Sequences

A sequence is executed by executing each of its elements, one after another, in the order that they appear in the structure.

< Sequence > ::= [; < Sequence of processes > ;]

< Sequence of processes > ::= < Process >
| < Process > ; < Sequence of processes >

Example:

```
[; A : Int := 0
 ; B : Int := 35
```

```
    ; C : Int := 13
    ; Find F of A and B and C
    ; Write ! F
    ;]
```

2.b. Parallel Structures

A parallel structure is executed by executing each of its sub-processes in parallel. Each constituent process is started at the same time, and the structure terminates when every one of these processes has finished.

```
  < Parallel structure > ::= [& < Parallel processes > &]

  < Parallel processes > ::= < Process >
                  ¦ < Process > & < Parallel processes >
```

Example:

```
   [& Producer
    & Buffer
    & Consumer
    &]
```

2.c. Alternative Structures

An alternative structure is executed by executing exactly one of its constituents (called a guarded process). Every guarded process contains a guard and a process. Each guard is attempted; as soon as one succeeds, its associated process is chosen, and no other guarded process may be executed. If more than one guard can succeed, then a potentially successful one is chosen non-deterministically. A duration in a guard must be a boolean duration. If several communications are in different branches of guards in a alternative structure, they must all have been declared within the same declaration block.

```
  < Alternative structure > ::= [¦ < Alternative processes > ¦]

  < Alternative processes > ::= < Guarded process >
                  ¦ < Guarded process > "¦" < Alternative processes >

  < Guarded process >::= < Guard > ¦ < Guard > -> < Process >

  < Guard >           ::= < Primitive >
              ¦ < Duration > ; < Primitive >
```

Example:

```
   [¦ X = 0 ; Terminate S
    ¦ Increment X S
      -> X := X + 1
    ¦ X > 0 ; Decrement X S
      -> X := X - 1
    ¦]
```

2.d. Declaration Blocks

A declaration block is a structure with initial declarations, followed by a process, which may be followed by definitions. The block is executed by executing the process within it.

```
< Declaration block >     ::= [>> < Initial declarations >
                          -- < Process >
                            < Final definitions > <<]

< Initial declarations > ::= < Declaration >
                         ¦ < Declaration >
                           >> < Initial declarations >

< Final definitions >     ::= < Empty >
                         ¦ << < Definition >
                           < Final definitions >
```

3. Declarations, Definitions, and Calls

The initial declarations of a declaration block may introduce constants, variables, linked ports, and names. Anything declared in this part of a block has scope extending from the declaration to the end of the block.

```
< Declaration > ::= < Constant declaration >
                ¦ < Variable declaration >
                ¦ < Linked port declaration >
                ¦ < Name declaration >
```

3.a. Constant declarations

A constant declaration associates a value with an identifier called a constant. The constants "True" and "False" have the boolean values true and false; sequences of digits have the values of the integers from zero to a fixed positive number in the usual way (e.g., the constant "34" has the value thirty-four). "MaxInt" has the value of the largest expressable integer (the smallest integer value that may be expressed is its additive inverse, -MaxInt).

```
< Constant declaration > ::= < Identifier > : < Expression >
```

A constant may be used in any expression within its scope.

Examples:

```
X : 3
A True Constant : True
X Plus 7 : X + 7
```

3.b. Variable declarations

A variable declaration associates a type (a set of values) with an identifier called a variable. A variable has a single value at any one time, like a constant, but the value may be changed by an assignment or an input

(the value of a constant may never change). Each value of a variable must be an element of the type of the variable. There are two types: the set of boolean values, represented by "Bool" and "Boolean", and the set of integer values from -MaxInt to MaxInt, represented by "Int" and "Integer".

< Variable declaration > ::= < Identifier > : < Type >

< Type > ::= Bool ¦ Boolean ¦ Int ¦ Integer

Examples:

Y : Int
P : Bool
A variable that is an integer and has a long identifier : Integer

A variable may be used in any expression within its scope if it has been assigned a value. A variable may be used in any target within its scope if the value being assigned to it is within its type.

3.c. Linked Port Declarations

A linked port declaration specifies and links two ports, which may then be used by parallel processes for communication. (The symbol "<->" may be read "is linked to").

< Linked port declaration > ::= < Port declaration > <-> < Port declaration >

A port may have scope ranging over all of the remainder of its declaration block, or it may be restricted to a single named process.

< Port declaration > ::= < Port declaration >
¦ < Restricted port declaration >

< Restricted port description > ::= < Name > (< Port description >)

A port is described by :

i) its tag, an identifier;
ii) its direction: "!" for sending, "?" for receiving, and "$" for synchronizing;
iii) the types of the values that may be communicated through it (if the port is directional).

The same tag may be used for different ports in the same process, if the ports have different directions or types. A send port must be linked with a receive port, and a synchronization port must be linked with another synchronization port.

< Port description > ::= < Send port description >
¦ < Receive port description >
¦ < Synchronization port description >

< Send port description > ::= < Identifier > ! < Type >

< Receive port description > ::= < Identifier > ? < Type >

< Synchronization port description > ::= < Identifier > S

Examples:

```
Tag 1 S <-> Tag 2 S
A ( Get ? Int ) <-> Send ! Int
B ( Done with test ! Bool )
  <-> Record Proc ( Progress ? Bool )
```

The ports that are connected to the standard input file and the standard output file are:

```
Read ? Int
End read S
Write ! Int
End write S
```

3.d. Name Declarations and Definitions

A name declaration introduces a process name that may be used in restricting ports and in calls and definitions. Names take no parameters.

< Name declaration > ::= < Identifier >

Each name has as scope the entire block from the declaration, with two restrictions: direct recursive calls must be tail-recursive, as with definition calls above; and, if an identifier is used in a named process, the same identifier must be in the environment at each point the process is called, referring to the same declaration.

A name declared in a declaration block may be defined anywhere within its scope. However, not every definition need by declared; a definition only names a process, and may be used anywhere that a process may. The name may be used in a call, which occurs as a process; then, executing the call consists of executing the process associated with the name.

< Definition > ::= < Name > :: < Process >

A call may use a name only within the scope of that name. If it has not been declared, then it may be called only as a tail-recursive call within the definition (i.e. the call must be the last sub-process to be executed within the definition).

< Call > ::= < Name >

Examples :

```
[>> Increment stack pointer
 >> Exchange X and Y
 -- [; Stack pointer : Int := 0
    ; Increment stack pointer
```

```
      ; X : Int := 5
      ; Y : Int := 9
      ; Exchange X and Y
      ;]

   << Increment stack pointer
      :: Stack pointer := Stack pointer + 1

   << Exchange X and Y
      :: [; Temp : Int := X
         ; X := Y
         ; Y := Temp
         ;]

   <<]

   Do A and B
    :: [& A & B &]

   A process call

   Fail if X is less than 5
    :: [¦ X < 5 -> Fail
       ¦ X >= 5 -> Pass
       ¦]

   Increment I to 5
    :: [¦ I < 5 -> [; I := I + 1 ; Increment I to 5 ;]
       ¦ I >= 5 -> Pass
       ¦]
```

4. Expressions

An expression may contain constants, variables, and denotations of operators; it is evaluated by applying the operators to their operands. A constant evaluates to its value, and a variable evaluates to its value if it has already been set (else the expression is undefined). The standard integer and boolean operators are provided, with the standard precedence rules; parentheses can be used to force the order of evaluation. Operators are presented here in order of precedence, weakest first.

< Expression > ::= < Bool expression > ¦ < Int expression >

4.a. Bool expression

A boolean expression is one or more boolean terms, separated by logical ors ("\/").

```
< Bool expression > ::= < Bool term >
                     ¦ < Bool term > \/ < Bool expression >
```

A boolean term is one or more boolean factors, separated by logical ands ("/\").

```
< Bool term > ::= < Bool factor >
                | < Bool factor > /\ < Bool term >
```

A boolean factor is one of a boolean constant or variable, a logical not ("~") followed by a boolean factor, a relation, or a boolean expression surrounded by parentheses.

```
< Bool factor > ::= < Bool constant >
                  | < Bool variable >
                  | ~ < Bool factor >
                  | < Relation >
                  | ( < Bool expression > )
```

4.b. Relation

A relation is either two boolean factors separated by equals ("=") or not equals ("~="), or two integer expressions separated by a relational operator (equals ("=") , not equals ("~="), less than ("<"), greater than (">"), not less than ("~<"), not greater than ("~>"). The first boolean factor cannot be a "logical not" followed by a boolean factor.

```
< Relation > ::= < Bool factor > < Equality operator > < Bool factor .
               | < Int expression > < Equality operator > < Int expression >
               | < Int expression > < Ordering operator > < Int expression >

< Equality operator > ::=  = | ~=

< Ordering operator > ::=  < | ~< | > | ~>
```

4.c. Int expression

An integer expression is one or more integer terms, separated by pluses ("+") or minuses ("-"), evaluated from left to right.

```
< Int expression > ::= < Int term >
                     | < Int term > < Sign > < Int expression >

< Sign > ::=  + | -
```

An integer term is one or more integer factors, separated by times ("*"), divided by's ("/"), or modulos ("\"), evaluated from left to right.

```
< Int term > ::= < Int factor >
               | < Int factor > < Multiplicative operator > < Int term >

< Multiplicative operator > ::=  * | / | \
```

An integer factor is one of an integer constant, an integer variable, a plus or minus followed by an integer factor, or an integer expression surrounded by parentheses.

```
< Int factor > ::= < Int constant >
```

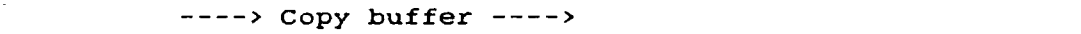
| < Int variable >
| < Sign > < Int factor >
| (< Int expression >)

C. Examples

Learning a programming language consists of both finding out what it allows one to do, and also of developing a "style". Programming style has two components: finding a readable, consistent syntactic structure; and building up a "library" of ways to solve frequently-occurring, small problems. The purpose of these examples is to aid the user in both these ways, by illustrating a syntax format, and by presenting solutions to problems that should give an idea of how the language can be used.

1. Copy buffer

This program is a simple buffer between the input and output files; it copies integers until the input file is no longer able to transmit values.



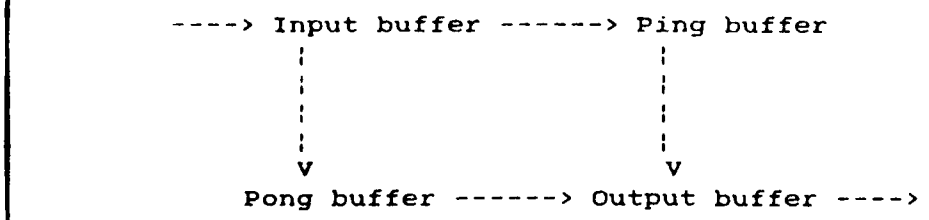

```
Copy buffer
:: [¦ Read ? X : Int -> [; Write ! X ; Copy buffer ;]
   ¦ End read $
   ¦]
```

Note : At the time this document was prepared, End read $ was not implemented. Thus, the copy buffer is terminated when it reads a zero from the input file. This is written:

```
Copy buffer
:: [; Read ? X : Int
   ; Write ! X
   ; [¦ X ~= 0 -> Copy buffer
     ¦ X  = 0
     ¦]
   ;]
```

2. Ping pong buffer

This program uses processes in parallel as buffers between the input and output files. The Input buffer reads values from the input file, and shunts them alternately to the Ping buffer and the Pong buffer, which then pass them on to the Output buffer, which writes them out to the output file.



Ping pong buffer

```
:: [>> Input buffer >> Ping buffer >> Pong buffer >> Output buffer
   >> Input buffer ( Give ping ! Int ) <-> Ping buffer ( Read ? Int )
   >> Input buffer ( Close ping $ )    <-> Ping buffer ( Close down $ )
   >> Input buffer ( Give pong ! Int ) <-> Pong buffer ( Read ? Int )
   >> Input buffer ( Close pong $ )    <-> Pong buffer ( Close down $ )
   >> Ping buffer ( Write ! Int )   <-> Output buffer ( Get from ping ? Int )
   >> Ping buffer ( Terminated $ )     <-> Output buffer ( Ping ended $ )
   >> Pong buffer ( Write ! Int )   <-> Output buffer ( Get from pong ? Int )
   >> Pong buffer ( Terminated $ )     <-> Output buffer ( Pong ended $ )
   -- [& Input buffer
      & Ping buffer
      & Pong buffer
      & Output buffer
      &]

   << Input buffer
     :: [; [¦ Read ? X : Int
           -> [; Give ping ! X
              ; [¦ Read ? X -> [; Give pong ! X ; Input buffer ;]
                 ¦ End of read $
                 ¦]
              ;]
         ¦ End of read $
         ¦]
       ; [& Close ping $ & Close pong $ &]
       ;]

    << Ping buffer
      :: [¦ Read ? X : Int -> [; Write ! X ; Ping buffer ;]
        ¦ Close down $   -> Terminated $
        ¦]

    << Pong buffer
      :: [¦ Read ? X : Int -> [; Write ! X ; Pong buffer ;]
        ¦ Close down $   -> Terminated $
        ¦]

    << Output buffer
      :: [; [¦ Get from ping ? X : Int
            -> [; Write ! X
               ; [¦ Get from pong ? X
                  -> [; Write ! X ; Output buffer ;]
                 ¦ Pong ended $ -> Ping ended $
                 ¦]
              ;]
         ¦ Ping ended $ -> Pong ended $
         ¦]
       ;]
    <<]
```

Note : As with the copy buffer, a few minor changes must be made to accomodate the lack of a Read end $. The input buffer is all that needs to be changed; it becomes:

```
    << Input buffer
      :: [; Read ? X : Int
```

```
        ; Give ping ! X
        ; [¦ X ~= 0 -> [; Read ? X
                       ; Give pong ! X
                       ; [¦ X ~= 0 -> Input buffer
                         ¦ X  = 0
                         ¦]
                       ;]
          ¦ X  = 0
          ¦]
        ; [& Close ping $ & Close pong $ &]
        ;]
```

3. Dining Philosophers

This is the traditional simulation of five dining philosophers, each of of whom does nothing but think and eat. They sit around a table to eat, with five forks that must be shared among them. This program has reduced the number of philosophers to three, in order to keep the listing fairly small.

```
Dining philosophers
:: [>> Dining room
   >> Phil 1 >> Phil 2 >> Phil 3
   >> Fork 1 >> Fork 2 >> Fork 3
   >> Think >> Eat
   >> Get washed
   >> Time to stop : 2000
   >> Dining room ( Enter 1 $ ) <-> Phil 1 ( Enter $ )
   >> Dining room ( Enter 2 $ ) <-> Phil 2 ( Enter $ )
   >> Dining room ( Enter 3 $ ) <-> Phil 3 ( Enter $ )
   >> Dining room ( Exit 1 $ ) <-> Phil 1 ( Exit $ )
   >> Dining room ( Exit 2 $ ) <-> Phil 2 ( Exit $ )
   >> Dining room ( Exit 3 $ ) <-> Phil 3 ( Exit $ )
   >> Phil 1 ( Pick up left fork $ ) <-> Fork 1 ( Picked up from right $ )
   >> Phil 2 ( Pick up left fork $ ) <-> Fork 2 ( Picked up from right $ )
   >> Phil 3 ( Pick up left fork $ ) <-> Fork 3 ( Picked up from right $ )
   >> Phil 1 ( Pick up right fork $ ) <-> Fork 2 ( Picked up from left $ )
   >> Phil 2 ( Pick up right fork $ ) <-> Fork 3 ( Picked up from left $ )
   >> Phil 3 ( Pick up right fork $ ) <-> Fork 1 ( Picked up from left $ )
   >> Phil 1 ( Put down left fork $ ) <-> Fork 1 ( Put down from right $ )
   >> Phil 2 ( Put down left fork $ ) <-> Fork 2 ( Put down from right $ )
   >> Phil 3 ( Put down left fork $ ) <-> Fork 3 ( Put down from right $ )
   >> Phil 1 ( Put down right fork $ ) <-> Fork 2 ( Put down from left $ )
   >> Phil 2 ( Put down right fork $ ) <-> Fork 3 ( Put down from left $ )
   >> Phil 3 ( Put down right fork $ ) <-> Fork 1 ( Put down from left $ )
   >> Dining room ( Phil 1 died $ ) <-> Phil 1 ( Died $ )
   >> Dining room ( Phil 2 died $ ) <-> Phil 2 ( Died $ )
   >> Dining room ( Phil 3 died $ ) <-> Phil 3 ( Died $ )
   >> Dining room ( Fork 1 quit $ ) <-> Fork 1 ( Quit $ )
   >> Dining room ( Fork 2 quit $ ) <-> Fork 2 ( Quit $ )
   >> Dining room ( Fork 3 quit $ ) <-> Fork 3 ( Quit $ )
   -- [& Dining room
      & Phil 1  & Phil 2  & Phil 3
      & Fork 1  & Fork 2  & Fork 3
      &]
```

```
<< Dining room
  :: [; Number of live philosophers : Int := 3
     ; Number of eaters : Int := 0
     ; Loop
      :: [; [¦ Number of eaters < 2 ; Enter 1 $
             -> [; Number of eaters := Number of eaters + 1 ; Loop ;]
           ¦ Number of eaters < 2 ; Enter 2 $
             -> [; Number of eaters := Number of eaters + 1 ; Loop ;]
           ¦ Number of eaters < 2 ; Enter 3 $
             -> [; Number of eaters := Number of eaters + 1 ; Loop ;]
           ¦ Exit 1 $
             -> [; Number of eaters := Number of eaters - 1 ; Loop ;]
           ¦ Exit 2 $
             -> [; Number of eaters := Number of eaters - 1 ; Loop ;]
           ¦ Exit 3 $
             -> [; Number of eaters := Number of eaters - 1 ; Loop ;]
           ¦ Phil 1 died $
             -> [; Number of live philosophers
                     := Number of live philosophers - 1 ; Loop ;]
           ¦ Phil 2 died $
             -> [; Number of live philosophers
                     := Number of live philosophers - 1 ; Loop ;]
           ¦ Phil 3 died $
             -> [; Number of live philosophers
                     := Number of live philosophers - 1 ; Loop ;]
           ¦ Number of live philosophers = 0
           ¦]
        ; [& Fork 1 quit $  & Fork 2 quit $  & Fork 3 quit $  &]
        ;]
     ;]

<< Phil 1 ::
  [; Think
   ; Write ! 1
   ; Write ! Time
   ; [¦ Time < Time to stop
      -> [; Enter $ ; Pick up left fork $  ; Pick up right fork $
         ; Eat      ; Put down left fork $ ; Put down right fork $
         ; Exit $   ; Phil 1 ;]
    ¦ Time ~< Time to stop -> Died $
    ¦]
   ;]

<< Phil 2 ::
  [; Think
   ; Write ! 2
   ; Write ! Time
   ; [¦ Time < Time to stop
      -> [; Enter $ ; Pick up left fork $  ; Pick up right fork $
         ; Eat       Put down left fork $ ; Put down right fork $
         ; Exit $   ; Phil 2 ;]
    ¦ Time ~< Time to stop -> Died $
    ¦]
   ;]

<< Phil 3 ::
```

```
    [; Think
     ; Write ! 3
     ; Write ! Time
     ; [! Time < Time to stop
        -> [; Enter $ ; Pick up left fork $  ; Pick up right fork $
           ; Eat      ; Put down left fork $ ; Put down right fork $
           ; Exit $   ; Phil 3 ;]
      ! Time >= Time to stop -> Died $
      !]
     ;]

  << Think :: 200
  << Eat :: 30

  << Fork 1 ::
    [! Picked up from left $
      -> [; Put down from left $ ; Get washed ; Fork 1 ;]
     ! Picked up from right $
      -> [; Put down from right $ ; Get washed ; Fork 1 ;]
     ! Quit $
     !]

  << Fork 2 ::
    [! Picked up from left $
      -> [; Put down from left $ ; Get washed ; Fork 2 ;]
     ! Picked up from right $
      -> [; Put down from right $ ; Get washed ; Fork 2 ;]
     ! Quit $
     !]

  << Fork 3 ::
    [! Picked up from left $
      -> [; Put down from left $ ; Get washed ; Fork 3 ;]
     ! Picked up from right $
      -> [; Put down from right $ ; Get washed ; Fork 3 ;]
     ! Quit $
     !]

  << Get washed :: 10
  <<]
```

D. How to Compile and Run a BCSP Program

1. Overview

A BCSP program is compiled under the UCSD Pascal system by executing the file COMPILE.CODE. The user is then prompted for parameters that must be given to the compiler. There are five parameters:

```
 i) the source file   ( default SOURCE.TEXT     );
ii) the code file     ( default BCSP.CODE.TEXT  );
iii) the errata file  ( default BCSP.ERR.TEXT   );
iv) listing wanted    ( default True            ); and
 v) the listing file  ( default BCSP.LIST.TEXT  )
```

Each parameter is set to the string typed in (a string is terminated by either a comma or a new line). If the parameter is a file, and is not set to CONSOLE:, then a .TEXT is appended to the string. If the default is desired, a comma or newline should be typed immediately. A right parenthesis sets any parameters left unset to the defaults, and starts the compiler. An up-arrow aborts the compiler while parameters are being set; a control B aborts the compiler while it is running.

The file INT1.CODE is executed to run a compiled program. The code must be in the file INPUTVM1.TEXT; any input for the program must be in the file CARDRDRVM1.TEXT, and any output is sent to PRINTERVM1.TEXT.

2. To recompile the Compiler

The following text files contain the compiler:

GlobalDecl.Text	- contains the unit GlobalDe
LexAnal.Text	- contains the unit LexicalA
ErrAndList.Text	- contains the unit ErrAndLi
ErrorGen.Text	- is included in the unit ErrAndLi
CodeGen.Text	- contains the unit CodeGene
CoreUtil.Text	- contains the unit CoreUtil
BCSP.Text	- contains the host program
CoreInit.Text	- contains a segmented procedure; included in BCSP
ExprParser.Text	- included in BCSP
PrimParser.Text	- included in BCSP
DeclParser.Text	- included in BCSP
StruParser.Text	- included in BCSP

To recompile the Compiler:

i. Compile GlobalDecl.Text
ii. Put the code file into the library BCSP.Library
iii. Compile LexAnal.Text
iv. Put the code file into the library BCSP.Library
v. Compile ErrAndList.Text
vi. Put the code file into the library file BCSP.Library
vii. Compile CodeGen.Text
viii. Put the code file into the library file BCSP.Library
ix. Compile CoreUtil.Text
x. Put the code file into the library file BCSP.Library
xi. Compile BCSP.Text into BCSP.Main.Code
xii. Link BCSP.Main.Code, using the library file BCSP.Library into the file Compile.Code

To put a file into a library:

i. X)ecute Library
ii. type the destination file (BCSP.Library)
iii. type the first source file (usually, BCSP.Library)
iv. copy the source file into the destination file
v. type n for new source file
vi. type new source file
vii. copy new unit into destination file

SECTION II. THE PARALLEL MACHINE

In this section, a machine is described that can execute programs written in BCSP. The specification makes no assumptions about the number of processors; thus, present-day machines may be used, although multi-processor machines are more efficient. The design is divided into:

A. Physical Architecture -
   How the machine is put together;
B. Software Architecture -
   The conventions assumed by the machine about its processes; and
C. The Instruction Set -
   The effect of executing each instruction.

This machine is based on that designed by Wirth for his PL/0 language [Wir76]; it has been expanded to cope with parallelism, communication, alternatives, and durations. It is process oriented, where a process is a processor paired with a workspace; the processor obtains instructions, and executes them by modifying either its state or that of a workspace. This description is both informal, in English, and semi-formal, using a BCSP-like notation; extensions to this notation are explained as they are introduced.

A. Physical Architecture

The parallel machine has five basic components.

```
Parallel machine : (& . Code store
                   & . Data store
                   & . Processor array
                   & . Register set
                   & . IO channels
                   &)
```

Note: This notation indicates that Parallel machine is declared to have five components, each of which is accessible at any time; this is roughly equivalent to Pascal's record notation. In Pascal, this could be written

```
ParallelMachine : Record
                    CodeStore : CodeStore ;
                    .
                    .
                  End
```

A field is accessed by subscripting the main identifier with one of its constituents.

1. The code store is an array of instructions indexed by addresses; it is initialized with the program to be executed. A program is a sequence of instructions; the textual representation is

```
< program >      ::= < instruction >
                   | < instruction > < new line > < program >
```

Details concerning instructions will be given later.

Each processor can access any instruction in the code store; however, no processor can modify any instruction. The way in which a program is loaded into the code store is not dealt with here.

```
>> Size of code store == 1500
>> Address == 0 ... Size of code store
>> Code store : Address --> Instruction
```

Notes:

i) The symbol ("==") means "is equivalent to"; the text following this symbol may be substituted for the identifier with no change in meaning. Such identifiers are referentially transparent. Thus, Size of code store is a constant, and Address is a type.

ii) The symbol ("...") indicates a range, much as in Pascal.

iii) The arrow indicates that Code store is a function, mapping addresses into instructions. Here, since addresses are finite, Code store is an array.

2. The data store is uninitialized. Each workspace is kept in the data store. Each processor can access and modify each location in the data store; however, the hardware ensures that if a data location is being modified, then no other processor can access it or modify it until the modification is finished. The store consists of an array of data values, indexed by locations, and a register containing the top location used in the data store.

```
>> Size of data store == 4500
>> Location == 0 ... Size of data store
>> Data value == Integer
>> Data store : (& . Top location used --> Location
          & Location --> Data value
          &)
```

Note : Data store . Top location used returns a Location. The earlier notation (E.g. for Parallel machine) is like this, but if the selector and the result use the same identifier, it need not be repeated.

3. There are assumed to be k processors in the machine; k is greater than or equal to 1. The number of active processors is initially 0; when a program is loaded into the machine, this number is set to 1.

```
>> Processor array : (& . Active processors --> 0 ... k
             & 1 ... k --> Processor
             &)
```

Each processor can access the code store, the data store, the register set, and the IO channels. Each of these may be modified except for the code store.

Each processor has a location register, which contains the location of the workspace currently being executed; and an instruction register, which contains the instruction currently being executed.

```
>> Processor : (& . Location register --> Location
          & . Instruction register --> Instruction
          &)
```

4. The register set contains four registers. The ready queue register contains the location of the workspace that is to be executed next. When a processor tries to find a workspace, it first checks the head of the ready queue. The hold queue register contains the location of the workspace that is waiting the least amount of time. When a processor increases the time, it transfers the head of the hold queue to the ready queue. The time register contains the current value of elapsed simulated time. This is initialized to zero, and must be a non-negative integer. The time slice contains the maximum number of instructions that may be executed within a process before the process is put back on the ready queue, and another process is given a chance to execute.

```
>> Register set : (& . Ready queue --> Location
            & . Hold queue --> Location
            & . Time register --> 0 ... MaxInt
            & . Time slice --> 1 ... MaxInt
            &)
```

5. The I/O channels connect the parallel machine with the outside world. The input wire is the only source of input for a program for the machine. It may be accessed and modified by any process. It has two states; either it has a value that is waiting to be accepted by a process, or it is empty. The output wire is the destination of any output of a program. Any process may send it values, until it is closed by a process.

```
>> Integer file == (¦ (; Integer
                     ; Integer file
                     ;)
                  ¦ End of file
                  ¦)
>> IO channels : (& . Input wire --> ? Integer file
            & . Output wire --> ! Integer file
            &)
```

Notes :

i) The main structure of the integer file is an alternative; this is like the union type, or the variant record in Pascal.

ii) The structure within the alternative is a sequence; each element of the sequence may be accessed or modified exactly once before moving on to the next element of the structure.

iii) Integer file is a tail-recursive type; this avoids the expense of the general recursive type, which requires trees with pointers.

iv) A type preceded by "?" indicates that it may be accessed, but not modified; a type preceded by "!" indicates that it may be modified, but not accessed.

B. Software Architecture

Each process keeps information relevant to that process in its workspace. A workspace is a contiguous set of locations in the data store. The code instructions assume that each workspace is set up in a particular way; this is the software architecture of the machine.

A workspace has three parts:

```
>> Work space : (& . Process control block
              & . Stack
              & . Size of work space
              &)
```

These parts are in this order; the process control block occupies the first few locations, the stack is in the middle, and the size of work space is the last location of the work space.

1. The Process control block (PCB) requires five locations. The parent process contains the location of the workspace of the process that created the current process. The location zero is used as the parent of the first process created. The link and child field contains zero initially. If the process is waiting in the ready queue, the hold queue, or a communication queue, then this field contains the location of the next process in the queue. If the process has children, and so is inactive, then this field contains the number of children that have not terminated. The address to be executed is initialized when the workspace is created; it is an index into the code store. If the process has children, the communications queue is used to mediate any communications among them. This field contains the location of the first child process that wishes to communicate at this level. The instruction count contains the number of instructions that have been executed within the current process in the current time slice. This number is initialized to zero, and must remain less than the value of the time slice register.

```
>> Process control block : (& . Parent process --> Location
                        & . Link and child --> Data value
                        & . Next address --> Address
                        & . Communication queue --> Location
                        & . Instruction count --> 0 ... MaxInt
                        &)
```

2. The stack contains a top of stack marker and a set of locations. The stack size differs for each process, and so cannot be specified further here.

```
>> Stack : (& . Top of stack --> 0 ... Size of data store
          & 1 ... Stack size --> Location
          &)
```

3. The size of the workspace is the size of the stack plus the overhead for the process control block, the top of stack marker, and this size of workspace marker; the total is the stack size plus 7. This value is stored at the top o

the workspace.

>> Size of work space : 0 ... Size of data store

C. The Instruction Set

Each instruction of the parallel machine is described in terms of its effects on the hardware machine and the software workspace of the process within which it was executed. An instruction is of the form

```
< instruction > ::= < mnemonic > < level > < argument >

< mnemonic >    ::= ACT | ALT | COM | FRK | HOL | INN | IOF | JMP
                  | JON | JPC | JPM | LIT | LOD | NOP | OOF | OPR
                  | OUT | PRO | STO | STP | SWI | TIM | UNS

< level >       ::= 0 | ... | 256

< argument >    ::= 0 | ... | 4095
```

When a processor executes an instruction, it begins by loading its instruction register with the instruction in the address of its workspace's TheAddressToBeExecuted register, and then incrementing the workspace's register.
The machine begins by initializing its components:

```
[; Read the program into the code store
 ; Data store . Top location used := 0
 ; Register set . Ready queue := 0
 ; Register set . Hold queue := 0
 ; Register set . Time register := 0
 ; Register set . Time slice := Instructions per time slice
 ; Reset ( IO channels . Input wire , Input file name )
 ; Rewrite ( IO channels . Output wire , Output file name )
 ; Processor array . Active processors := 1
 ; Processor array . 1 . Location register := 0
 ; Processor array . 1 . Instruction register := Code store . 0
 ;]
```

Note : A period is used to separate field selectors from function names, as in Pascal. However, the period is used for arrays here as well.

There are some standard operations that several instructions perform. An informal description of these utilities is presented here.

Push (A process , A value)
puts the value on the top of the stack of the given process.

Pop (A process)
removes the value on the top of the stack of the given process.

The top of (A process)
is a function that returns the value of the top element of the stack of the given process.

Obtain a process
causes a processor to check the Ready queue; if this is not empty, then the first process is removed from that queue and execution of that process

begins. If the Ready queue is empty, then the number of active processes is checked. If that number is greater than one, then the processor goes to sleep; if that number is one, then the Hold queue is checked. If the Hold queue is empty, then the program is in deadlock, and an error message is produced; if the Hold queue is not empty, then the first process on it, and each successive process with the same hold time, is transferred to the Ready queue. The processor then wakes up processors and gives them the processes on the Ready queue until either there are no more free processors or there are no more processes on the Ready queue. If there is still a process on the Ready queue, then the processor begins to execute it; if not, then the processor goes to sleep.

1. Activate
   ACT 0 A

This is the first instruction of each program; it directs the processor to initialize the machine and create a workspace, with a stack of size A, which is used by the processor as the current process.

```
[; Processor . Location register := 1
 ; Data store . Top location used := A + Work space overhead
 ; Current PCB == Work space . Process control block
 ; Current PCB . Parent process := 0
 ; Current PCB . Link and child field := 0
 ; Current PCB . Address being executed := 1
 ; Current PCB . Communication queue := 0
 ; Current PCB . Instruction count := 0
 ; Work space . Stack . Top of stack := 0
 ; Work space . Size of work space := A + Work space overhead
 ;]
```

2. Alternative
   ALT L 0

The stack has a set of addresses, each of which points to a communication (a zero is used to mark the end of the set). Each communication is attempted until one succeeds (if any do). If none succeed, then the process is put on the communication queue, until an outside communication succeeds with one of them and wakes the process up. Once a communication is successful, all the communication addresses are removed from the stack, and execution continues following the successful communication. The communication arguments are different from those in the ordinary communication: their levels must all be the same, and this level is the level argument L of the alternative; the level fields of the communications have the absolute values of their transmissions. Since communications in alternatives may not be outputs, any non-zero transmission signifies an input.

```
[; Find L'th ancestor of the current process
 ; Temporary stack index := Top of stack
 ; Loop ::
  [] Stack . Temporary stack index > 0 ->
   [] The communication pointed to by ( Stack . Temporary stack index )
                        can succeed ->
     [; Strip the communication addresses and the 0 marker from the stack
```

```
      ; Execute the communication
      ;]
    ¦ ~ ( The communication pointed to by ( Stack . Temporary stack index )
                        can succeed ) ->
      [; Temporary stack index := Temporary stack index - 1 ; Loop ;]
     ¦]
   ¦ Stack . Temporary stack index = 0 ->
     [; Attach workspace to end of communication queue
      ; Obtain a process from the ready queue
      ;]
   ¦]
 ;]
```

3. Communicate
COM L A

The top element of the stack is the transmission, which describes the number of values passed, and their direction. The level L indicates the number of process levels to skip to find the appropriate communication queue. The argument A is the tag number, which must match for the communication to succeed. The processor finds the communication queue L levels back, and checks each process on the queue until it finds one that is trying to communicate using the tag number A and the transmission, which is on the top of the other process' stack if it is at a COM and which is the L field of each of the other COMs if it is at an ALT. If it finds such a process, it executes the communication; if the transmission is 0, no values are transferred; if the transmission is 1, a value is transferred to the other process' stack; and if the transmission is -1, a value is obtained from the other process' stack. Then, the other process is put at the end of the ready queue. If no appropriate process was found on the communication queue, then the current process is put on the communication queue and the processor obtains the next process from the ready queue.

```
 [; Find the L'th ancestor of the current process
  ; Temporary communication location
          := The communication queue of the ancestor process
  ; While ( A does not match the tag of the queue process' instruction )
                And ( ~ At the end of the queue ) Do
    Temporary communication location
       := The next workspace on the queue
  ; [¦ At the end of the queue ->
    [; Attach the current process to the end of the queue
     ; Obtain a process from the ready queue
     ;]
    ¦ ~ At the end of the queue ->
    [; [¦ The queue process is at an ALT ->
         Strip off the addresses and the 0 marker from the queue process
       ¦ The queue process is at a COM ->
         Pop ( Queue process ) (* remove the transmission *)
       ¦]
     ; [¦ Top of ( Current process ) (* the transmission *) = -1
       -> [; Pop ( Current process )
          ; Push ( Current process , Top of ( Queue process ) )
               (* transmitted value *)
          ; Pop ( Queue process )
          ;]
```

```
      | Top of ( Current process ) (* the transmission *) = 0
         -> Pop ( Current process )
      | Top of ( Current process ) (* the transmission *) = 1
         -> [; Pop ( Current process )
            ; Push ( Queue process , Top of ( Current process ) )
                  (* transmitted value *)
            ; Pop ( Current process )
            ;]
      |]
    ;]
  |]
;]
```

4. Fork
   FRK 0 0

The stack has on it a set of data addresses, each of which points to a process (a zero is used to indicate the end of the set). The processor puts each process on the end of the ready queue, and removes the address from the stack. Then, the zero is removed from the stack, the child count is set to the number of processes put on the ready queue, and the processor leaves the current process and gets the next process off of the ready queue.

```
[; Link and child field := 0 (* Set the children counter to 0 *)
 ; Loop ::
  [| Top of stack > 0 ->
   [; [| Active processors < k ->
       [; Get an idle processor
        ; Load the top of the stack into its Location register
        ; Start it executing
        ;]
      | Active processors = k ->
       [; Find the end of the Ready queue
        ; Attach the process whose location is on the top of the stack to
             the end of the Ready queue
        ;]
      |]
    ; Link and child field := Link and child field + 1
    ; Pop ( Current process )
    ;]
  | Top of stack = 0
  |]
; Obtain a process
;]
```

5. Hold
   HLD 0 0

The processor calculates the time at which the process is to be woken up (the current time plus the value on the top of the stack) and loads this onto the stack. It then puts the current process on the hold queue immediately before the process (if any) on the queue that is to be woken up after the current process. It then obtains a new process to execute.

```
[; Top of stack := Top of stack + Time register
 ; [¦ Hold queue = 0 (* is empty *) -> Hold queue := Location register
   ¦ Hold queue > 0 (* is not empty *) ->
     [; Temporary queue element := The first element on the Hold queue
      ; Loop ::
       [¦ Top of ( Temporary queue element )
                    < Top of ( Current process ) ->
          [; Advance along the queue by one process ; Loop ;]
        ¦ Top of ( Temporary queue element )
                    "< Top of ( Current process )
        ¦]
      ; Insert the current process into the queue at this point
      ;]
   ¦]
 ; Obtain a process
 ;]
```

6. Input
   INN 0 0

   A value is accepted from the input file, and put on the top of the stack.

```
 Read ( Input wire , Top of ( Current process ) )
```

7. Input end of file
   IOF 0 0

The input file is checked to see if it is empty; if it is not, then the process waits until the input file is empty before continuing.

```
 [¦ Input wire is at end of file -> Close and lock the input wire
 ¦ " ( Input wire is at end of file ) ->
    [; Put the current process on the communication queue of
                                  the first process
     ; Obtain a process
     ;]
 ¦]
```

Note : This instruction is not yet implemented.

8. Jump
JMP 0 A

If the argument A is zero, then the top of the process stack is removed and transferred into the process address register; otherwise, the argument A is copied into the address register.

```
[¦ A = 0 -> [; Next address := Top of ( Current process )
            ; Pop ( Current process ) ;]
 ¦ A > 0 -> Next address := A
 ¦]
```

9. Join
JON 0 0

The processor erases the current process workspace, and decrements the value of its parent's child counter. If that value becomes zero, then the current process is the last child to terminate, and the processor puts the parent on the ready queue. It then obtains a new process to execute.

```
[; Next address := 0
 ; [¦ Current process uses the top workspace in the data store ->
      Reduce the top location used until it is at
                  the top of a process that has not terminated
   ¦ Current process does not use the top workspace in the data store
   ¦]
 ; Decrement the child counter of the parent process
 ; [¦ Child counter of ( Parent process ) = 0 ->
      Location register := Parent process
   ¦ Child counter of ( Parent process ) > 0 ->
      Obtain a process
 ;]
```

10. Jump if false
JPC 0 0

If the top value of the process stack is True, then the top two values of the process stack are removed; if the top value is False, then the process address becomes the second value of the process stack, and the top two values of the process stack are removed.

```
[¦ Top of ( Current process ) = True ->
  [; Pop ( Current process ) ; Pop ( Current process ) ;]
 ¦ Top of ( Current process ) = False ->
  [; Pop ( Current process )
   ; Next address := Top of ( Current process )
   ; Pop ( Current process )
   ;]
```

11. Jump and mark
JPM 0 A

The next address is loaded onto the process stack, and the process address becomes A.

```
[; Push ( Current process , Next address + 1 )
 ; Next address := A
 ;]
```

12. Literal
  LIT 0 A

The argument A is loaded onto the process stack.

```
 Push ( Current process , A )
```

13. Load
  LOD L A

The value in the location with offset A from the process L levels back is loaded onto the process stack.

```
 Push ( Current process , The A'th element in the stack of
                      the L'th ancestor of the current process )
```

14. NoOp
  NOP 0 0

The processor does nothing, and moves on to the next instruction.

```
 Pass
```

15. Output end of file
  OOF 0 0

The output file is closed.

```
 Close and lock the output wire
```

Note : This instruction is not yet implemented.

16. Operator
  OPR 0 A

The argument A determines which of various standard operations is applied to the top element(s) of the process stack, which is (are) removed. The result of the operation is put on the process stack. The arguments to OPR, with their corresponding operations, are:

```
   0 : Arithmetic complement
   1 : Arithmetic addition
   2 : Arithmetic subtraction
   3 : Arithmetic multiplication
   4 : Arithmetic division
   5 : Arithmetic modulo
   6 : Logical not
```

```
 7 : < Undefined >
 8 : Equals
 9 : Not equals
10 : Greater than
11 : Greater than or equal to
12 : Less than
13 : Less than or equal to
14 : Logical and
15 : Logical or
```

17. Output

```
OUT 0 0
```

The top value of the process stack is transferred to the output file.

```
Write ( Output wire , Top of ( Current process ) )
```

18. New process

```
PRO 0 0
```

A new process workspace is created, whose size is the top value on the current process stack. The new PCB is loaded as follows:

i) the parent is the current process;
ii) the address is the second value on the current process stack;
iii) the level is one plus that of the current process.

The top two values of the current process stack are removed, and the data location of the new workspace is placed on the current process stack.

```
[; Temporary location := Top location used + 1
 ; Top location used := Top location used + Work space overhead
                     + Top of ( Current process )
 ; Size of work space := Work space overhead
                     + Top of ( Current process )
 ; Pop ( Current process )
 ; Current PCB == Work space . Process control block
 ; Current PCB . Parent process := Location register
 ; Current PCB . Link and child field := 0
 ; Current PCB . Next address := Top of ( Current process )
 ; Current PCB . Communication queue := 0
 ; Current PCB . Instruction count := 0
 ; Work space . Stack . Top of stack := 0
 ; Top of ( Current process ) := Temporary location
 ;]
```

19. Store

```
STO L A
```

The top element of the process stack is removed and transferred to a location with offset A that is L processes back.

```
[; Find the process L levels back along the parent process chain
 ; Transfer ( Top of ( Current process ) ) to the location of
```

```
     the found process' stack with offset A
  ; Pop ( Current process )
  ;]
```

20. Stop

```
   STP 0 A
```

The processor

i. frees its workspace;
ii. zeros its registers;
iii. tells the system's bookkeeping that it is free.

The argument A indicates the kind of termination:

0: Successful termination
1: Unsuccessful termination
2: An alternative failure

```
 [; Top location used := 0
  ; [¦ Active processors > 1 ->
     [; Kill off all other active processors
      ; [¦ A = 0 ->
          Write an error message saying that other processes are still alive
        ¦ A > 0 ->
          Write an error message describing the kind of termination
        ¦]
      ;]
    ¦ Active processors = 1 /\ A > 0 ->
        Write an error message describing the kind of termination
    ¦ Active processors = 1 /\ A = 0
    ¦]
  ; Terminate the current processor
  ; Active processors := 0
  ;]
```

21. Switch

```
   SWI 0 0
```

The top two values on the process stack are switched.

```
 [; 1st temporary value := Top of ( Current process )
  ; Pop ( Current process )
  ; 2nd temporary value := Top of ( Current process )
  ; Pop ( Current process )
  ; Push ( Current process , 1st temporary value )
  ; Push ( Current process , 2nd temporary value )
  ;]
```

22. Load time

```
   TIM 0 0
```

The current time is loaded onto the top of the process stack.

```
 Push ( Current process , Time register )
```

23. Unstack
UNS 0 A

If the argument A is zero, then values are removed from the process stack until the value 0 is found and removed. If the argument A is greater than 0, then A values are removed from the process stack.

```
[¦ A = 0 -> Loop :: [¦ Top of ( Current process ) ~= 0
                    -> [; Pop ( Current process ) ; Loop ;]
                  ¦ Top of ( Current process )  = 0
                    -> Pop ( Current process )
                  ¦]
 ¦ A > 0 ->  A -;> Pop ( Current process )
 ¦]
```

Note : The symbol -;> indicates sequential iteration here, to be done A times on the process to the right of the -;>.

References

[HOA78] Hoare, C.A.R. Communicating Sequential Processes. Comm. ACM 21, 8 (August 1978), pp. 666-677.

[WIR76] Wirth, N. Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs, N.J. 1976.

GLOSSARY OF TERMS

A. Key words and phrases used in description of BCSP

Alternative structure -
a structured process of guarded processes, from which exactly one is chosen to be executed

Assignment -
a primitive process that sets the value of a variable

Call -
a process consisting of a name, which when executed causes the process associated with that name to be executed

Constant -
an identifier associated with a single, unchanging value

Constant declaration -
a declaration in a declaration block that associates an identifier with a value

Declaration block -
a structured process that delimits the scope of constants, variables, linked ports, and names declared within it

Definition -
either a process or one of the last parts of a declaration block; it associates a name with a process

Direction -
a characteristic of a port, distinguishing an output ("!") from an input ("?") from a synchronization ("$")

Duration -
a primitive process that causes simulated time to pass; if boolean, this may occur as part of a guard

Execution -
the obeying of the directives of a process

Expression -
a description of values and operators on values that yield a value; this may occur in a duration, assignment, or output

Guard -
a primitive process, possibly preceded by boolean durations, used to test whether a process within an alternative structure should be chosen; the primitive must be one of an input, a synchronization, or a boolean duration

Guarded process -
an element of an alternative structure, consisting of a guard and a process

Identifier -
a sequence of words, separated with spaces and new lines

Input -
a primitive process that accepts a value from an output in another, parallel process and assigns that value to a variable

Linked port declaration -
declaration of two ports and the connection between them, that enables one process to affect the behavior of another

Name -
an identifier associated with a process

Operator -
a function on boolean or integer values that returns a boolean or integer value

Output -
a primitive process which evaluates an expression and sends it to an input in another, parallel process

Parallel structure -
a structured process of processes, in which each constituent process is executed in parallel

Port -
an object specified by a tag and a direction that can be used by a process to interact with another parallel one

Port declaration -
a constituent of a linked port declaration

Port description -
a specification of a port

Primitive -
an indivisible process that is supplied in all implementations of the language

Process -
the fundamental unit of execution, specified by its behavior

Program -
a process that is a self-contained unit for solving a problem

Recursion -
calling a name from within the process associated with that name

Restricted port declaration -
a port declaration that is only valid for use within a specified named process

Scope -
the portion of text within which a declaration is valid

Sequence -
a structured process of processes in which each constituent process is executed one after another, in the order in which they appear

Simulated time -
a monotonic function used to regulate relative orders of execution of processes

Structure -
a combination of processes that can have more complicated behavior than any of its constituents

Synchronization -
a primitive process that is executed simultaneously in two parallel processes, used to signal the current point of execution of each

Tag -
an identifier that forms part of the specification of a port

Tail-recursion -
a form of recursion in which the call is the last process within the named process, so that there is no need to save the return address

Target -
a variable that is given a new value, or an initialized variable declaration

Type -
an identifier associated with a set of values

Variable -
an identifier associated with a value and a type; its value must always be an element of its type, and may be changed when the variable is a target

Variable declaration -
a declaration that introduces a variable in a declaration block or a target; the variable is initialized only if the declaration is a target

B. Identifiers and Symbols Used in BCSP

1. Identifiers

Each identifier here has synonyms obtained by translating the characters into entirely upper case and entirely lower case.

Bool , Boolean
   a type associated with the boolean values
End read
   a tag for the synchronization that matches when the input file is empty
End write
   a tag for the synchronization that indicates that the output file should be closed
Fail
   a name whose associated process never terminates
False
   a constant associated with the value false
Int , Integer
   a type associated with the integer values from -MaxInt to MaxInt
MaxInt
   the largest integer value allowed in this language: 2040
Pass
   a name whose associated process does nothing and terminates immediately
Read
   a tag used to obtain values from the input file
Time
   a variable that is associated with the elapsed simulated time; this variable may not be used as a target
True
   a constant associated with the boolean value true
Write
   a tag used to send values to the output file
0 , 1 , 2 , 3 , ...
   constants associated with the integer values

2. Symbols

```
:=    is assigned to              ; used in assignments
$     sync                        ; indicates a synchronization port
!     output                      ; indicates an output port
?     input                       ; indicates an input port
:     is declared as              ; used in constant and variable declarations
::    is defined as               ; used in definitions
<->   link                        ; declare and link the two surrounding ports
[>>   begin declaration block     ; indicates a new declaration block
>>    declare                     ; separates declarations in a declaration block
--    main process                ; precedes the process in a declaration block
<<    where                       ; precedes each definition in a declaration
block
<<]   end of block                ; concludes a declaration block
```

```
[|   begin alternative          ; indicates a new alternative structure
->   implies                    ; separates a guard from its process
|    or do                     ; separates guarded processes in disjunction
|]   end alternative            ; concludes alternative structure
[&   begin parallel structure   ; indicates a new parallel structure
&    and do                     ; separates processes in conjunction
&]   end parallel structure     ; concludes parallel structure
[;   begin sequence             ; indicates a new sequence
;    then do                    ; separates processes in sequence
;]   end sequence               ; concludes sequence
(    left paren                 ; used in port restrictions, expressions
)    right paren                ; used in port restrictions, expressions
+    plus                       ; an operator on Int and Int x Int, returns Int
-    minus                      ; an operator on Int and Int x Int, returns Int
*    times                      ; an operator on Int x Int, returns an Int
/    divide                     ; an operator on Int x Int-(0), returns an Int
\    modulo                     ; an operator on Int x Int-(0), returns an Int
=    equals                     ; an operator on Bool x Bool or Int x Int,
                    returns a Bool
~=   not equal                  ; an operator on Bool x Bool or Int x Int,
                    returns a Bool
<    less than                  ; an operator on Int x Int, returns a Bool
>    greater than               ; an operator on Int x Int, returns a Bool
~>   not greater than           ; an operator on Int x Int, returns a Bool
~<   not less than              ; an operator on Int x Int, returns a Bool
/\   and                        ; an operator on Bool x Bool, returns a Bool
\/   or                         ; an operator on Bool x Bool, returns a Bool
~    not                        ; an operator on Bool, returns a Bool
(*   begin comment              ; signifies that a comment is to follow
*)   end comment                ; concludes a comment
```

C. Key words and phrases used in description of the Parallel Machine

```
Address                 - index to element of Code store
Address being executed  - element of Process control block
Code store              - element of Parallel machine
Communication queue     - element of Process control block
Data store              - element of Parallel machine
Data value              - element of Data store
Hold queue              - element of Register set
Input wire              - element of IO channels
Instruction             - element of Code store
Instruction count       - element of Process control block
Instruction register    - element of Processor
IO channels             - element of Parallel machine
Link and child field    - element of Process control block
Location                - index to element of Data store
Location register       - element of Processor
Output wire             - element of IO channels
Parallel machine        - the data structure representing a hardware machine
Parent process          - element of Process control block
Process                 - a Processor and Work space pair
Process control block   - element of Work space
Processor               - element of Processor array
```

Processor array	- element of Parallel machine
Ready queue	- element of Register set
Register set	- element of Parallel machine
Size of code store	- largest Address allowed
Size of data store	- largest Location allowed
Size of work space	- largest value allowed for Top of stack
Stack	- element of Work space
Time register	- element of Register set
Time slice	- element of Register set
Top of stack	- element of Stack
Top location used	- element of Data store
Work space	- a software unit, in Data store

D. Machine Code Instructions for the Parallel Machine

ACT	**0**	**A**	Activate
ALT	**L**	**0**	Alternative
COM	**L**	**A**	Communicate
FRK	**0**	**0**	Fork
HLD	**0**	**0**	Hold
INN	**0**	**0**	Input
IOF	**0**	**0**	Input end of file
JMP	**0**	**A**	Jump
JON	**0**	**0**	Join
JPC	**0**	**0**	Jump if false
JPM	**0**	**A**	Jump and mark
LIT	**0**	**A**	Literal
LOD	**L**	**A**	Load
NOP	**0**	**0**	NoOp
OOF	**0**	**0**	Output end of file
OPR	**0**	**A**	Operator
OUT	**0**	**0**	Output
PRO	**0**	**0**	New process
STO	**L**	**A**	Store
STP	**0**	**A**	Stop
SWI	**0**	**0**	Switch
TIM	**0**	**0**	Load time
UNS	**0**	**A**	Unstack