The Estelle E-Code

1 The Stack

The E-Code machine uses an array of 16-bit integers called the code store, which holds the program code (of fixed length). Each process is also allocated a stack (also an array of 16-bit integers). Each stack begins at address 1 and grows upwards.

Three index registers are used to access the stack:

- p – (program) address of current instruction
- b – (base) start of stack (current activation)
- s – (stack) current stack pointer

Each process has its own p, b and s registers.

Initially the stack is empty. When execution begins, space is allocated on the stack for the exported variables, parameters, and variables defined in the process. The exported variables occur first, then the parameters, followed by a dummy activation record (see below), and then the variables.

When the program activates a procedure, space is allocated in an activation record; this is freed up on exit from the procedure. Activation records consist of:

- Parameters
- Context (static link, dynamic link, return address)
- Variables
- Temporary scratch area

The first three parts are of fixed length and are created when the procedure is activated. The temporary part holds operands and results during the execution of statements. It is empty at the beginning and end of every statement. The b register contains the address of the static link. This is called the base address of the activation record. Accesses to variables and parameters is done by using (positive and negative) displacements from the b register.

The dynamic link of an activation record contains the base address of the previous activation record. When a procedure terminates, the dynamic link stored in the current activation record is assigned to the b register. These links represent the dynamic sequence in which blocks are activated.

The static links define the set of variables that are accessible within the current block (the current context). These links represent the static block structure of the program text.

To access a variable or parameter, we need to know the activation record and displacement. To aid this, the compiler assigns a level number to each block in a program. During code generation, the compiler assigns a level number and displacement to each variable. The relative level of a block is the number of static links that must be followed to reach a variable. Relative level 0 is the current activation record.

Accesses to variables, parameters, etc, are handled by instructions which calculate the address of the variable as an offset in the stack. A reference to an exported variable of a

child process results in a negative address. By examining the address, a process can tell whether the variable exists on its own stack, or the stack of a child. In the latter case, the index of the child process will have been pushed on to the stack as well, so this can be popped, the correct child stack identified, and the variable accessed (after first making the address positive).

The use of negative addresses for exported variables is an elegant solution to a complex problem, but has a side effect - the address 0 can never be used. For this reason, stack addressing starts at address 1. However, this has an advantage, as it allows easy checking for null pointer assignments - any attempt to access address 0 is an error.

We now turn to the instructions used in the program code. We will first examine the Pascal P-code subset, and then look at the extended E-code instructions. Throughout the description, we assume that:

push(v)	*is the same as*	**stack[++s] = v**
pop(v)	*is the same as*	**v = stack[s–]**
chain(level,x)	*is the same as*	**x=b; while (level–) x=stack[x]**

Note that the P-code is not any particular standard; it is adapted from the P-code used by Per Brinch-Hansen in 'Brinch-Hansen on Pascal Compilers'.

The format used in describing the operations is to give the stack entry and exit states, where important, followed by an informal description of the action of the instruction, and a pseudo-code algorithm for the instruction. As we progress, less algorithms will be given, as they will be assumed to be self-evident except in complex cases (although in some complex cases, particularly the E-Code instructions, algorithms are not given as they depend too much on internal implementation details for processes, transitions, etc). Sometimes several instructions are grouped together; an algorithm is then given which is representative of all those in the group. The stack states only show information which is relevant to and altered by the instruction - it is assumed that everything below this information remains unchanged.

2 The Pascal P-Code Subset

2.1 Variable(Relative Level, Displacement)

Stack on Exit: → Absolute Address of Variable

Action

Used to access value parameters and local variables. Compute the absolute address of a variable and place it on the stack. This is done by traversing the specified number of levels of the static chain, obtaining the relative base address, adding the displacement, and pushing the result on to the stack.

Algorithm:

```
void variable(level,displ)
{
        chain(level,x);
        push(x+displ);
        p += 3;
```

```
}
```

2.2 VarParam(Relative Level, Displacement)

Stack on Exit: → Address of variable bound to Parameter

Action

Used to access variable parameters. Variable parameters are implemented by storing the address of the bound variable as the value of the parameter. Thus, to access the address of a variable parameter, we need to find the address of the parameter, and push its value (the address of the bound variable) onto the stack. This is a minor modification of the action of Variable.

Algorithm:

```
void VarParam(level, displ)
{
    chain(level,x);
    push(stack[x+displ]);
    p += 3;
}
```

2.3 Index(Lower, Upper, Element size)

Stack on Entry: → Array Index Value
Base Address of Array

Stack on Exit: → Base Address + (EltSize * IndexValue)

Action

Used to access array elements. Removes an index value from the top of the stack, checks that it is in the specified range (if not, reports an error), and increments the address value on the top of the stack by the index value multiplied by the element size; ie, the address of the indexed element. The index must be checked for sign; if it is negative, it represents a child's exported array. In this case, the final address pushed must also be negative.

Algorithm:

```
void Index(Lower, Upper, EltSize)
{
    exported = FALSE;
    pop(ix);
    if (ix<0) { exported = TRUE; ix = –ix; }
    if (ix<Lower || ix>Upper) Error;
    else stack[s] += (ix–Lower)*EltSize;
    if (exported) stack[s] = –stack[s];
    p += 5;
}
```

2.4 Field(Displ)

Stack on Entry: → Base Address of Record

Stack on Exit: → Base Address + Field Displacement

Action

Used to access fields of records. Adds the specified field displacement to the address on top of the stack. If the base address is negative, the record is a child's exported record.

Algorithm:
```
void Field(Displ)
{
    if (stack[s]<0) stack[s] -= Displ;
    else stack[s] += Displ;
    p += 2;
}
```

2.5 Constant(Value)

Stack on Exit: → Constant Value

Action

Pushes a constant value on to the stack.

Algorithm:
```
void Constant(Value)
{
    push(Value);
    p += 2;
}
```

2.6 Value(Length)

Stack on Entry: → Address of Variable
[Child number]

Stack on Exit: → Value of Variable

Action

Used to access the value of a variable. Pops the address on the top of the stack and pushes the value held at that address (the number of words to push is specified by the Length).

Algorithm:
```
void Value(Length)
{
        pop(address);
        if (address<0)
                {
                pop(child);
                select child's stack;
                address = –address;
                }
        while (Length–-)
        push(stack[address++]);
        p += 2;
}
```

2.7 Not(), Minus(), Abs(), Sqr(), Odd()

Stack on Entry: → Value

Stack on Exit: → Result of performing unary op on value

Action

Negates the (Boolean or Integer) value, calculates the absolute value, calculates the square root, or returns a Boolean 1 if the value is odd, 0 otherwise, respectively.

Algorithm:
```
void Not(void)
{
        stack[s] = !stack[s];
        p++;
}
```

2.8 Multiply(), Div(), Modulo(), Add(), Subtract(), And(), Or()

Stack on Entry: → Value2
Value1

Stack on Exit: → Value1 *op* Value2

Action

Multiplies (etc) the top two elements of the stack, and replace them with the result.

Algorithm:
```
void Multiply(void)
{
        pop(val);
        stack[s] *= val;
```

```
        p++;
}
```

2.9 Less(), Equal(), Greater(), NotGreater(), NotEqual(), NotLess()

Stack on Entry: → Value2
Value1

Stack on Exit: → Value1 *relop* Value2

Action

Replaces the top two values on the stack with a Boolean TRUE if the first value is less (etc) than the second one; otherwise FALSE.

Algorithm:

```
void Less(void)
{
        pop(val);
        stack[s] = (stack[s] < val);
        p++;
}
```

2.10 Assign(Length)

Stack on Entry: → Value to Assign
Address to Assign to
[Child number]

Stack on Exit: →

Action

Removes the value on the top of the stack, the address below it, and assign the value to the address.

Algorithm:

```
void Assign(Length)
{
        s –= length;
        pop(rvalue);
        lvalue = s+2;
        if (rvalue<0)
                {
                rvalue = –rvalue;
                pop(child);
                deststack = child stack;
                }
```

```
        else deststack = stack;
        while (Length--)
        deststack[rvalue++] = stack[lvalue++];
        p += 2;
}
```

2.11 Goto(Displ)

Action

Increment the p register by the displacement.

Algorithm:
```
void Goto(Displ)
{
        p += Displ;
}
```

2.12 Do(Displ)

Stack on Entry: → Boolean Value

Stack on Exit: →

Action

Either proceed to the next instruction or jump to another instruction, depending on the value on the top of the stack (which is removed).

Algorithm:
```
void Do(Displ)
{
        if (pop()) p+=2;
        else p+= Displ;
}
```

2.13 For(Displ, Mode)

Stack on Entry: → End Value
Start Value
Address of Control Variable

loop unfinished:
Stack on Exit: → End Value
Address of Control Variable

loop finished:
Stack on Exit: →

Action

Initialise a loop control variable with a start value, compare it to an end value, and if the loop is over, jump out using a displacement. Otherwise push the address and end value back on to the stack for use by the Next instruction. The Step field is the STEP size, adjusted to be positive (TO) or negative (DOWNTO).

Algorithm:
```
void For(Displ, Step)
{
        pop(end);
        pop(start);
        pop(address);
        stack[address] = start;
        if (Step*start > Step*end)
        p += Displ;
        else {
                push(address);
                push(end);
                p+=3;
                }
}
```

2.14 Next(Displ, Mode)

Stack on Entry: → End Value
Address of Control Variable

loop unfinished:
Stack on Exit: → End Value
Address of Control Variable

loop finished:
Stack on Exit: →

Action

Check whether or not a FOR loop is complete. If not, leave stack as it was and bramnch back using the Displ. If it is, leave stack clear and drop through to next instruction. See also For.

Algorithm:
```
void Next(Displ, Step)
{
        pop(end);
```

```
        pop(address);
        stack[address]+=Step;
        if (Step>0 && stack[addr]<end) or
              (Step<0 && stack[addr]>end)
              {
              push(address);
              push(end);
              p += Displ;
              }
        else p+= 3;
}
```

2.15 ProcCall(Level, Displ, ReturnLen)

Stack on Exit: → Return Address
Dynamic Link
Static Link

Action

Creates the context part of an activation record, and jumps to the procedure code.

Algorithm:

```
void ProcCall(Level, Displ, ReturnLen)
{
        chain(Level,x);
        s += ReturnLen;
        push(x); /* static link */
        push(b); /* dynamic link */
        push(p+3); /* return address */
        b = s–2;
        p += Displ;
}
```

2.16 Procedure(VarLength, TempLength, Displ)

Stack on Exit: → Last Word of Local Variable Space
Local Variable Space

Action

Allocates space on the stack for a procedures variables, ensures that there is sufficient space remaining for the temporaries, and branches to the statement part of the procedure.

Algorithm:

```
void Procedure(VarLen, TmpLen, Displ)
{
        if ((s+VarLen+TmpLen)>stacksize) expand stack;
```

```
        s += VarLen;
        p+= Displ;
}
```

2.17 EndProc(ParamLength, ReturnLen)

Stack on Entry: → Last Word of Local Variable Space
Local Variable Space
Activation Record
[Return Value]
Parameters

Stack on Exit: → [Return Value]

Action

Uses the dynamic link to remove the activation record of the procedure and jump to the return address, and pushes the return value, if any.

Algorithm:

```
void EndProc(ParamLen)
{
        s = b+2; /* pop variables */
        ReturnValTop = b–1;
        pop(p);
        pop(b);
        pop();
        s –= ParamLen+ReturnLen;
        while (ReturnLen–-)
                push(stack[ReturnVarTop–ReturnLen]);
}
```

2.18 ReadStr(Length), ReadCh(), ReadInt()

Stack on Entry: → Address to Read to
[Child number]

Stack on Exit: →

Action

Input a string, character or integer and assign it to the address on the top of the stack. System dependent!

Algorithm:

```
void ReadStr(Length)
{
        Read string from keyboard;
```

```
        pop(address);
        if (address<0)
                {
                address = –address;
                pop(child);
                select child's stack;
                }
        while (Length–--)
                stack[address++] = next character of string;
        p+=2;
}

void ReadInt() /* ReadCh similar */
{
        Read integer from keyboard;
        pop(address);
        if (address<0)
                {
                address = –address;
                pop(child);
                select child's stack;
                }
        stack[address] = integer;
        p++;
}
```

2.19 WriteStr(Length), WriteCh(), WriteInt()

Stack on Entry: → Value to Write

Stack on Exit: →

Action

Outputs the value on the top of the stack. System Dependent!

Algorithm:

```
void WriteStr(Length)
{
        s –= Length;
        Output Length characters from stack[s+1] on.
        p+=2;
}

void WriteInt(void) /* WriteCh is similar */
{
        pop(value);
        Output value as an integer;
```

```
        p++;
}
```

2.20 EndWrite()

Action

Indicates the end of a WRITE or WRITELN statement. The action is system-dependent; in the PEW this results in the I/O window popping up and being displayed for one second.

2.21 DefArg(Label,Value)

Action

Inform the linker of the actual value of a symbolic label, ie L[Label] = Value. These instructions are removed from the code output by the linker.

2.22 DefAddr(Label)

Action

Inform the linker of the actual address of a symbolic label, ie L[label] = current address. These instructions are removed from the code output by the linker.

2.23 AddSetElement()

Stack on Entry: → Element Number (0...63)
Set Value (4 words)

Stack on Exit: → New Set Value (4 words)

Action

The specified element is added to the set. Sets are represented using 64-bits (ie, four 16-bit words), and the element number is a value in the range 0...63. This specified bit is set in the set value.

Algorithm:

```
void AddSetElement(void)
{
        pop(elt);
        word = elt/16;
        bit = elt%16;
        stack[s–word] |= 1<<bit;
        p++;
}
```

2.24 AddSetRange()

Stack on Entry: → Range End (0...63)
Range Start (0...63)
Set Value (4 words)

Stack on Exit: → New Set Value (4 words)

Action

All bits in the specified range are set. See AddSetElement (41) for information on the representation of sets.

Algorithm:

```
void AddSetRange(void)
{
    pop(end);
    pop(start);
    while (start<=end)
        {
        word = elt/16;
        bit = elt%16;
        stack[s–word] |= 1<<bit;
        start++;
        }
    p++;
}
```

2.25 In()

Stack on Entry: → Stack Value (4 words)
Element Number (0...63)

Stack on Exit: → Boolean result

Action

If the specified bit is set in the set value, push 1 on to the stack, else push 0 on to the stack.

Algorithm:

```
void In(void)
{
    s –= 4;
    pop(elt);
    word = elt/16;
    bit = elt%16;
    if (stack[s+5–word] && (1<<bit)) push(1);
    else push(0);
```

```
        p++;
}
```

2.26 SetAssign()

Stack on Entry: → Stack Value (4 words)
Destination address
[Child number]

Stack on Exit: →

Action

Assigns the four words holding the set value to the four words starting at the specified destination address. If the destination address is negative, the variable is a child's exported variable, so the child number is used to determine which child stack to use.

Algorithm:

```
void SetAssign(void)
{
        dest = stack[s–4];
        source = s–4;
        if (dest>=0) deststack = current stack;
        else {
                pop(child);
                deststack = child's stack;
                }
        while (source!=s)
                deststack[dest++] = stack[source++];
        s –= 5;
        p++;
}
```

2.27 SetEqual()

Stack on Entry: → Set1 (4 words)
Set2 (4 words)

Stack on Exit: → Boolean value

Action

Pushes 1 on the stack if the two sets are identical.

2.28 SetInclusion, SetInclusionToo

Stack on Entry: → Set1 (4 words)
Set2 (4 words)

Stack on Exit: → Boolean value

Action

Pushes 1 on the stack, if Set2 is a subset of Set1 (SetInclusion), or Set1 is a subset of Set2 (SetInclusionToo), else pushes 0.

2.29 Intersection(), Union(), Difference()

Stack on Entry: → Set1 (4 words)
Set2(4 words)

Stack on Exit: → Result Set (4 words)

Action

Computes the union, intersection or difference (Set2–Set1) of two sets, and leaves the result on the stack.

2.30 StringEqual(Len), StringGreater(Len), StringLess(Len), StringNotEqual(Len), StringNotGreater(Len), StringNotLess(Len)

Stack on Entry: → String1 (of length 'Len')
String2 (of length 'Len')

Stack on Exit: → Boolean value

Action

Pushes a 1 on to the stack if String2 satisfies the specified relationship with String1, else pushes 0. For example, if String1 is gretaer than String2, then StringLess would push 1. 'Len' specifies the length of the strings.

2.31 IndexedJump(Offset)

Action

This is used for CASE statements. It should be followed by a number of GOTO instructions, which correspond to the different cases. The offset is popped, multiplied by 2 (the size of a GOTO instruction), and added to the p register to get the new instruction, which should be a GOTO.

2.32 Case()

Action

Generates a run-time error. This instruction is used to signal the end of a case statement, and corresponds to the situation where the case expression has no corresponding label.

2.33 Copy()

Stack on Entry: → Value

Stack on Exit: → Value
Value

Action

Copy duplicates the value on the top of the stack.

2.34 Pop()

Stack on Entry: → Value

Stack on Exit: →

Action

Pop the value from the top of the stack.

2.35 New(Numwords)

Stack on Exit: → Address of Allocated Block

Action

Attempts to find an unallocated block of memory on the heap (free stack space) of the specified number of words. In the *PEW*, if such a block is found, it is linked in to a chain of allocated blocks, and the address is pushed on to the stack. If the allocation is not possible, a run-time error occurs. The use of a chain of allocated blocks is inefficient - it would be better to use a chain of free blocks; however, such details are implementation dependent. If the New instruction is the first such instruction to be executed by the containing process, its stack is expanded before allocation. Once a New has occurred, a process' stack is fixed in size and cannot be re-expanded.

2.36 Dispose()

Stack on Entry: → Address of Allocated Block

Stack on Exit: →

Action

Locates the allocated block on the chain of allocated blocks (see 'New'), and removes it from the chain. If no corresponsing block is found, a run-time error occurs.

2.37 EnterBlock(Stack Space)

Action

Specifies the minimum amount of stack space that is required to execute the block about to be entered. A check is performed to see if the stack has sufficient free space, and if not, an attempt is made to expand the stack. If this fails, a run-time error is reported.

2.38 Newline(Linenum)

Action

Used to synchronise the E-Code with the original Estelle source code.

2.39 Random()

Stack on Entry: → Range of Random Numbers

Stack on Exit: → Random Number

Action

Pops the Range value off the stack, generates a random number between *0* and *Range–1* inclusive, and push this value on the stack.

2.40 Range(Minimum, Maximum)

Stack on Entry: → Value

Stack on Exit: → Value

Action

Checks that the value on the top of the stack is in the specified range; if not, generates a run-time error.

2.41 ReturnVar(Displ)

Stack on Entry: →

Stack on Exit: → Address of Variable

Action

Adds the displacement to the activation record base address to get the return variable address, and pushes this on to the stack.

2.42 Scope(Level,Offset)

Action

Used to keep track of symbol table scope. The parameters are the scope level, and offset into the symbol table storage area of the first symbol table entry at this level. This is highly implementation dependent.

2.43 With()

Stack on Entry: →

Stack on Exit: → Activation Record for WITH statement

Action

Used to identify the new scope level of a WITH statement. It simply pushes a new activation record on to the stack.

2.44 WithField(Field Offset, With Level)

Stack on Entry: →

Stack on Exit: → Address of Field

Action

References to fields within WITH statements cause this instruction to be generated. The With Level is the number of activation records to be traversed to get to the record address associated with the field, while the Field Offset is the offset of the field within that record.

Algorithm:

```
void WithField(FieldOffset, WithLevel)
{
    chain(WithLevel,addr);
    recaddr = stack[addr–1];
    if (recaddr<0) fldaddr = recaddr – FieldOffset;
    else fldaddr = recaddr + FieldOffset;
    push(fldaddr);
}
```

2.45 EndWith(Numlevels)

Action

Pops the specified number of activation records off the stack (each record in the list of a WITH statement causes an activation record to be pushed on to the stack, **preceded** by the address of the record being specified), and then pops the address of the first record.

3 E-Code extensions to P-Code

3.1 NumChildren()

Stack on Exit: → Number of Module Variables

Action

Pushes the number of module variables that the currently executing process has on to the stack.

3.2 GlobalTime()

Stack on Exit: → Current Time

Action

Pushes the current simulation time on to the stack. WARNING - the time is pushed as a 16-bit word, whereas times are actually represented in the *PEW* as 32-bits. Truncation is thus a very real possibility.

3.3 FireCount()

Stack on Exit: → Number of Executions

Action

Pushes the number of times the currently executing transition has been executed on to the stack.

3.4 QLength()

Stack on Entry: → IP Index
IP Identifier

Stack on Exit: → IP Queue Length

Action

Pushes the length of the reception queue associated with the specified IP on to the stack. The IP Identifier is not used, but is generated by the compiler for IP references anyway, so it is simply discarded.

3.5 SetOutput(Directive Value)

Action

Implements the *PEW* compiler directive for controlling the destination of output due to WRITE and WRITELN procedure calls.

3.6 Domain(Numvars)

Stack on Entry: →

Stack on Exit: → Activation Record
Space for Domain Variables

Action

Used for EXIST, FORONE and FORALL constructs in Estelle. It allocates space on the stack for the specified number of domain variables, and then pushes a new activation record.

3.7 EndDomain(NumVars)

Stack on Entry: → Result
Activation Record
Domain Variables

Stack on Exit: → Result

Action

Identifies the end of a EXIST, FORONE or FORALL Estelle construct. The result is popped off the stack, the activation record popped, the domain variables released, and the result pushed back on to the stack.

3.8 ExistMod(Header Ident, Offset to Failure Code)

Stack on Entry: → ...
Domain Activation Record
Module Domain Variable

Stack on Exit: → ...
Domain Activation Record
Module Domain Variable

Action

Checks whether the child process identified by the module domain variable has the specified header identifier. If it does, control falls through to the next statement. If

not, control passes to the failure code. This is used for domain constructs (EXIST, FORONE, FORALL), where the domain type is a module domain. For example, to execute an EXIST factor, the module domain variable is iterated from 0 through to *numchildren–1*, and ExistMod is executed for each value, until all values are exausted, or a successful value is found.

3.9 ExpVar(Level,Displacement)

Stack on Entry: → Child Number

Stack on Exit: → Exported Variable Address
Child Number

Action

Specifies a reference to an exported variable - the alter-ego of the Variable instruction. The activation chain is traversed on the *child's* stack to find the address.

Algorithm:
```
void ExpVar(Level,Displ)
{
    pop(childnum);
    select child's stack;
    chain(Level,x);
    select parent's stack;
    push(childnum);
    push(–x–Displ); }
```

3.10 DefIPs(Start,Size)

Action

Used to specify which IP's share a common reception queue. The linker rearranges the E-Code so that all DefIPs instructiosn for a process occur contiguously. This block of instructions is then processed when a module is created. The parameters specify the start index of the set of IPs, and the number of IPs in the set. For example, if a process has 7 IPs, and of these the 3rd, 4th and 6th share a common reception queue, two instructions would be generated, namely DEFIPS(4,2) and DEFIPS(7,1) (the numbering of IPs begins with 0).

3.11 Attach()

Stack on Entry: → Child IP Index
Child IP Identifier
Child Index
Parent IP Index
Parent IP Identifier

Stack on Exit: →

Action

Attaches the child external IP to the parent IP. Any interactions queued at the parent IP are prepended onto the queue associated with the *downattach* of the child IP. For details on the semantics of Attach, refer to the Estelle literature. This instruction corresponds exactly to the high level ATTACH instruction in its effects.

3.12 Connect(IsInternal1, IsInternal2)

Stack on Entry: → IP Index1
IP Ident1
[ModVar Index1 if IP1 is external]
IP Index2
IP Ident2
[Modvar Index2 if IP2 is external]

Stack on Exit: →

Action

Connect the two specified interaction points. The indices of the interaction points are on the stack. If either or both of the interaction points are not internal (ie, they are child external), then the index of the appropriate child module variable must also be on the stack.

3.13 Output(Length of Arguments, Reliability)

Stack on Entry: → [Interaction Arguments]
Interaction Length
Interaction Identifier
IP Index
IP Identifier

Stack on Exit: →

Action

Pops and saves the Interaction Arguments, creates a new interaction with these saved arguments and the other parameters on the stack, timestamps the interaction, and enqueues it on the appropriate queue specified by the Interaction Point Index. The reliability is used to randomly determine whether it should actually be output, or whether it should be lost.

3.14 Detach(Mode)

if Mode is 0:
Stack on Entry: → IP Index
IP Identifier

if Mode is 1:
Stack on Entry: → IP Index
Child Index
IP Identifier

if Mode is 2:
Stack on Entry: → Child Index

Stack on Exit: →

Action

The Mode specifies whether all child external IPs of a specified child are to be detached (mode 2), a specific child external IP is to be detached (mode 1), or a specific IP belonging to the current process is to be detached (mode 0). The IP Identifier is not used in any case, but is discarded if present. The appropriate module (current or child) is selected, and attachments of IPs for that module are broken - either a single specific attachment, or all attachments (mode 2). Interactions that arrived through the higher IP are removed from the *downattach* of the lower IP and returned to the queue of the higher IP.

3.15 Disconnect(Mode)

if Mode is 0:
Stack on Entry: → IP Index
IP Identifier

if Mode is 1:
Stack on Entry: → IP Index
Child Index
IP Identifier

if Mode is 2:
Stack on Entry: → Child Index

Stack on Exit: →

Action

The Mode is as for Detach. This instruction is very similar to Detach, except that no interactions are moved.

3.16 Module(Body Identifier,Header Identifier, Class, Variable Space, Number of Module Variables, Number of Initialisation Transitions, Number of Body Transitions, Number of Internal IPs, Number of External IPs, Offset to first Initialisation Transition, Offset to first Body Transition, Offset to DefIP instructions, Offset to Scope instructions)

Action

This is the instruction that initialises modules. Together with TRANS, it is the modt complex instruction in the instruction set.

Various data structures for the new process are created, including a module variable table, transition table, IP table, and stack. Reception queues for the IPs are alloacted, with the DefIPs instructions being processed to determine which IP table entries share the same reception queue. If the Offset to DefIPs instructions has the value –1, there are no common queues.

A scope table is allocated for the module, and the Scope instructions are processed to load up this table.

The initialisation parameters are popped off the parent stack (see Init) and pushed on to the new stack. Space is reserved for the exported variables, an activation record is created, and then space is reserved for the (non-exported) variables.

If the module has initialisation transitions, one of these is selected for execution and executed. Finally, the parent process is reactivated, and execution continues from the parents Init instruction.

3.17 Init(Displacement to Module instruction, ParamLen)

Stack on Entry: → [Initialisation Parameters]
Module Variable Index

Stack on Exit: →

Action

Creates a new module instance, using the module body definition at the specified address (see Module), and assigns the new module instance to the specified module variable.

3.18 Release()

Stack on Entry: → Child Number

Stack on Exit: →

Action

The specified child process and all its children are killed. Before releasing the process,

all its external interaction points are disconnected and/or detached. Any module variables of the current process that referred to the released process are made undefined.

Additional processing that occurs in the PEW is for all breakpoints that refer to the released process or its children only are cleared (not yet implemented), and the execution summary statistics for the process and its children are written to the log file.

3.19 Terminate()

Stack on Entry: → Child Number

Stack on Exit: →

Action

Terminate is a simple form of Release. The only difference is that external IPs of the terminated process that are attached at the time of termination are detached using a form of detach known as *simpledetach*. Unlike detach, a simpledetach does not result in any interactions being moved.

3.20 ModuleAssign()

Stack on Entry: → Source Module Variable Number
Destination Module Variable Identifier
Destination Module Variable Number

Stack on Exit: →

Action

Assigns the destination module variable to have the same value as the source. The identifier is used in the *PEW* for symbolic support by the process browser.

3.21 Delay(Type)

Stack on Entry: → Minimum Delay
Maximum Delay

Stack on Exit: → Boolean Value

Action

Compares the length of time that the transition has been enabled (call this *etime* with the Maximum Delay. If the etime exceeds the maximum delay, TRUE is pushed on to the stack. If the etime is less than the Minimum Delay, FALSE is pushed on to the stack. If the etime falls between the minimum and maximum delay times, a

random delay in the interval is chosen (according to the specified distribution), and if this is less than the etime TRUE is pushed, else FALSE is pushed. The *PEW* also keeps track of the shortest delay over all DELAY clauses in a specification, and if no transitions are enabled but the system has not deadlocked, the global time is updated by this shortest delay.

The Type parameter specifies the distribution type for random delays - if the type is 0 then a uniform distribution is used; if the type is negative a Poisson distribution (with –Type as the mean value) is used; otherwise an exponential distribution with Type as the mean value is used.

3.22 When(Interaction Identifier, Interaction Length

Stack on Entry: → IP Offset

Stack on Exit: → Value

Action

The identifier of the interaction at the head of the specified IP is compared with the Interaction Identifier argument. If they are identical, and the destination IP of the head interaction is indeed the specified IP (this check is necessary because of common queues) then (IP Offset + 1) is pushed on to the stack, else 0 is pushed on to the stack. The +1 is necessary to ensure a non-zero value, as IP Offsets begin at zero. The 'len' parameter is not used, and will be removed in the future.

3.23 Otherwise(First,Last)

Stack on Entry: →

Stack on Exit: → Boolean Value

Action

This instruction implements the PROVIDED OTHERWISE construct. The parameters First and Last specify the range of transitions whose PROVIDED clauses must all have failed for the PROVIDED OTHERWISE clause to succeed. As the transitions in the range must all precede the transition containing the PROVIDED OTHERWISE clause, the state of their provided clauses is known by the time this instruction executes.

3.24 Trans(Name, FromStates, From Identifier, ToState, To Identifier, Priority, Offset to Provided Clause, Offset to When Clause, Offset to Delay Clause, Offset to next Trans instruction, Length of When Interaction, Size of Variable Space, Size of Temporary Space, Offset to Transition Block)

Action

This instruction, like Module, is not really 'executed', as such, but is used to impart

information to the interpreter about a transition. The first time that the scheduler examines the containing process, the identifiers, name, and priority are all entered into the transition table entry for the transition. The scheduler will also use the Trans instruction each time to locate the clauses of the transition, and if the transition is selected for execution, to locate its body. The Variable and Temporary Space arguments are used to check for potential stack overflow before executing a transition.

3.25 EndTrans()

Action

Indicates the end of a transition block. No action is taken.

4 P-Code Optimisations

Some of the P-Code instructions are optimised forms of common P-Code instruction pairs. These are:

Instruction:	*is equivalent to:*
LocalVar(Displ)	Variable(0,Displ)
LocalValue(Displ)	Variable(0, Displ); Value(1)
GlobalVar(Displ)	Variable(1,Displ)
GlobalValue(Displ)	Variable(1,Displ); Value(1)
SimpleValue	Value(1)
SimpleAssign	Assign(1)
GlobalCall(Displ)	ProcCall(1,Displ)

Furthermore, a sequence:

Variable(Level, VDispl)
Field(FDispl)

can be replaced by the single instruction:

Variable(Level, VDispl+FDispl)

5 The Scheduler

This is the heart of the interpreter. It is an 'infinite' loop, which successively executes the following steps:

1. Calculate, recursively, for all processes in the process tree, the state of each clause of each transition of each process. Keep track of the smallest positive delay of all transitions that are only disabled due to their delay clauses failing.

2. If no enabled transition were found, update the time by the smallest delay found, and start again.

3. If enabled transitions were found, once again recurse through the process tree. If a process has no enabled transitions, recurse through its children, else bootom out the recursion process at this point (as parent transitions always take priority). If

the process offers more than one transition to be fired, use the priorities to select. If there are transitions that are enabled and have the same priorities, select randomly between them.

4. When recursing through children, if the parent is an activity or systemactivity, bottom out of the recursion the moment a single child has a transition executed. For processes and systemprocesses, continue the recursion through all of the children's peers.

5. If no transitions were found, and no delay was found in the first step, the system has deadlocked.