CMU-CS-81-122

Schemes for Communication

Mathai Joseph*

3 June 1981

Table of Contents

Abstract

This report describes features of a language for distributed and parallel programming which has been designed to provide flexibility in the transfer of information and control between the individual components of a program. The language allows synchronous and asynchronous message-passing, multiple-source input and broadcast output, and enables particular features of a distributed architecture to be efficiently accommodated without modification to the language. The module serves as the unit of encapsulation and a single communication takes place between an output *port* in one module and a set of input *ports* in other modules: each port has a *control rule* which specifies the protocol for sending or receiving messages, and is associated with a particular *communication scheme* which implements the communication operations. Modules are assumed to execute independently of each other except when they communicate by sending messages: the lifetime of a module is therefore limited only by its ability to send and receive messages. The use of the distinctive features of the language, such as broadcast mode output, is illustrated with several examples.

*Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa 15213. *On leave from the Tata Institute of Fundamental Research, Bombay, India.*

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Projects Agency or the US Government.

1 Introduction

Most large programs are built of smaller components which interact by the transfer of information or control. Traditional sequential programming languages allowed these components to be described as *procedures* and combined the operation of transferrring information between them with the exchange of control, in a *procedure call.* Quasi-parallel programming languages (eg, Concurrent Pascal[1], or Modula[2]) extended the kinds of components to include *processes* which communicate with each other by executing mutually exclusive procedure calls to shared *monitors.* In both these cases, there was the tacit assumption that the transfer of information could take place by access to shared memory and, often, also that individual processes would share a single processor. In contrast to this, the parallel programming languages such as CSP[3] and DP[4] start out with the assumption that individual processes will execute on separate processors and that the transfer of information will be accomplished by sending *messages*, usually requiring some form of communication between processors.

The forms of explicit communication between program components have, so far, been determined by the nature of the components themselves: for example, whether information is transferred by a synchronous procedure call or by the asynchronous transfer of a message is usually dependent on whether the called component is a procedure or a process. However, the distinctions go further than this: though there are several conventions even for message passing (eg, ranging from the 'no-wait' send of Plits[5], to the 'remote procedure call' of DP), each language has also made a choice of the particular kind of communication it will provide.

In this paper, we will describe a language in which programs are built of *modules* and where the 'behaviour' of a module - eg, whether procedure-like or process-like - is determined solely by the kinds of communication in which it participates. Two modules may thus communicate with each other in several ways, according to the needs of the information they are exchanging: rather than attempt to artificially simulate asynchronous (or message passing) communication between two modules which are compelled to use synchronous (or procedure call) communication, or to introduce protected tokens as a means of enforcing synchronicity between processes which otherwise communicate asynchronously, the language permits the sender and the receiver of information to specify the mode in which the transfer is to occur. This is done by defining *ports* through which communication is to take place, by explicitly making connections between ports, and by associating a control protocol with every port.

Ports are distinguished by the functions they serve, ie input or output, and by the types of messages they transmit. An output port is termed **oport** and may be connected to any input port, or **iport**, which accepts a message which is 'compatible' with the one it will transmit. More generally, a single port may be connected to

a set of ports: a message sent through an **oport** is therefore sent to each **iport** to which it is connected, and an **iport** may correspondingly receive a message from any one of its communicating oports. Since ports are identified by names which are 'local' to modules, communication paths can be specified without using system-wide names.

An underlying goal for the language is efficient implementation on a variety of architectures, ranging from those with shared memory to distributed systems which only share communication links. This is kept within reach by ensuring that the only language features which are particularly dependent on the architecture are those of the communication constructs. Here, the assumptions made are minimal and many different, and efficient, implementations can easily be envisaged for different architectures. We will show that, in fact, ports are essentially no different from other data types programmed in the language, making it possible for a user to define new *communication schemes*, distinct from, say, the standard one supplied by an implementation of the language. Moreover, it is emphasised that different implementations *will* probably be simultaneously present and that the specification of a module will include the choice of the *communication scheme* to be used for its implementation. Separating the design of a communication scheme from the use of communication constructs has the convenient side-effect that issues such as the presence, absence and extent of buffering remain questions for the method of implementation of the communication *scheme* and not for the modules which use this scheme.

The success with which a language such as this meets its goals has, at this stage, to be judged largely by appeal to examples and experience; one intention of this presentation is to illustrate that there seems every possibility that this goal can be met. At this point in time it is difficult to cite much practical experience in the use of *any* such language, since distributed systems are still largely experimental (notable exceptions are, of course, the Xerox Ethernet-based Alto computers), and there is not even a wide and well-accepted set of problems whose solutions would unambiguously demonstrate the features of the language. It appears to be particularly important now that language designs suitable for distributed systems be demonstrated using varied examples. We will therefore choose examples which both illustrate how this language compares with others and which demonstrate the purpose of its design. In keeping with this, we will restrict the description primarily to those features of the language which will be illustrated by examples and which are significantly different from those in other languages.

2 Language Overview

We will first describe each language feature with examples and then illustrate parts of the syntax in a BNF-like notation. Boldface will be used for reserved words, and in the syntax productions square brackets [...] will denote zero or more occurrences of the enclosed clause, while braces { ... } will denote that the enclosed

clause must appear at least once. Special symbols which appear in the language are placed within double quotation marks. Types will be defined only to the extent they are required for the examples, and it will be assumed that expressions are composed in the usual way. The simple (ie, pre-defined) types in the language are **boolean**, **char**, **integer** and **real**; the communication types are **oport** and **iport**, and the standard aggregate types are **array** and **set**, which must be homogeneously composed of elements of the same type, and **record** which may have dissimilar fields.

Programs are built of *modules*: a module is composed of a *specification part* which defines the ways in which the module communicates with other modules, and an *implementation part* which contains the instructions for the operations performed by the module.

2.1 Module Specifications

A *module type* is defined with the header **module type**, a set of specification statements, and a terminator **end** <module name>. An instance of a module may either be declared with the header **module specs** or by instantiation as a variable in some other module.

A module specification must name all the modules and the *attributes*[1] of other modules which are needed for the implementation of the module. These attributes consist of constants, types and ports and they are denoted by qualifying the name of an attribute (eg, Input) with the name of the module in which it appears (eg, CardReader.Input); an attribute name appearing by itself denotes an instantiation parameter. These external attributes are specified in a **needs** statement:

```
needs iport Print.Out;
    module Transmitter;

needs type MaxM, MaxMO, Stat, Message;

needs const Buff.N ;
    iport CardReader.Input; LinePrinter.Output;
    oport Timer.Alarm;
```

A module specification must also contain definitions of all the attributes provided *by* the module, in a **provides** statement. In this case, every port definition must have additional information: the *type* of each message parameter it will transmit, and the *control rule* for that port. The control rule specifies the sequence of actions for communication with a port:

For example,

```
provides oport Put(char) -> ( ); Over(boolean);
    iport Get(char) => ( ); TimeOut( ); Next(integer) -> (char);
```

declares an **oport** Put which transmits a parameter of type **char** and waits for a null message ( ) to be returned, and an **oport** Over which transmits a **boolean** parameter; similarly, Get, Timeout and Next are iports, the first taking a character of type **char** and returning a null message, and the last taking an **integer** parameter and returning a character. Other attributes, such as types and constants, can be provided:

```
provides type EntryType; const N : integer;

provides iport FromLeft [1..100] (integer);
    oport ToRight [1..100] (integer);
```

where, in the last cases, an array of ports is defined. The control rules indicated by -> and => have the following meaning:

iport rule

->	a message sent to this **iport will** result in a message being sent in response
=>	a message sent to this **iport** *must* be followed by the sender waiting for a response

oport rule

->	the sender will wait for a message to be sent in reply

It may be seen that there are three possible control rules for iports and two for oports, since in the simplest cases there is no response on receiving a message and no waiting after sending a message (for convenience, we will sometimes denote these cases by "-").

The third type of specification statement is the **connect** statement which is used to specify how the ports which have been named in **needs** and **provides** statements are to be connected to each other. The **connect** statement has the form:

```
connect (Put, LinePrinter.Output);

connect (S.FromLeft[2..100], S.ToRight[1..99]);

connect (TimeOut, Timer.Alarm);
```

which connects the first named port (or ports) with the list of ports which follows. Note that:

connect (A, B) ≡ connect (B, A)
connect (A, B, C) ≡ connect (A, B); connect (A, C)
connect (A[1 .. 10], B[3 .. 13]) ≡ connect (A[1], B[3]) .. connect (A[10], B[13])

For a connection to be legal,

1. The kinds of ports being connected must be complementary, ie an **oport** connected to iports or an **iport** connected to oports,

2. The message types transmitted by the ports must be compatible, and

3. The control rules for the ports must match according to the following requirements:

oport	iport	Compatible
-	-	Y
-	->	Y
->	->	Y
->	=>	Y

All other connections are illegal.

<ModuleSpecs> ::= **module** <ModuleKind> <Ident> ";" SpecBody **end** Ident "."
<ModuleKind> ::= **specs** | **type**
<SpecBody> ::= [NeedStmt] [ProvideStmt] [ConnectStmt]
<NeedStmt> ::= **needs** {<AttribType> {<NameList> ";" }}
<ProvideStmt> ::= **provides** {<AttribDef> ";" }
<ConnectStmt> ::= **connect** { "(" <NameList> ")" ";" }
<AttribType> ::= <PortAttrib> | **module** | **type** | **const**
<AttribDef> ::= <PortAttrib> <PortSpecs> [":" <PortSpecs>] | **const** <IdentList> ":" <Ident>
| **type** <IdentList> "=" <SimpleTypeDef> | **module** <ModuleSpec> ":" <Ident>
<PortAttrib> ::= **iport** | **oport**
<PortSpecs> ::= <Designator> <ParList> <ControlRule>
<NameList> ::= <QualIdent> ["," <QualIdent>]
<QualIdent> ::= <Ident> "." <Designator> | <Designator>
<Designator> ::= <Ident> | <Ident> <ArrayRange>
<ModuleSpec> ::= <ArrayRange) **of** <IdentList> | <IdentList>
<ControlRule> ::= "-> <ParList>" | "=> <ParList>" |
<ParList> ::= "(" <IdentList> ")"
<IdentList> ::= <Ident> ["," <Ident>] |

2.2 Module Implementation

A module specification defines attributes of a module to allow them to be used in that module, and in others, for defining types and constants and for making connections between ports. The actual operations of a module are defined in its *implementation*. A module implementation always has a header with the same identifier as its specifications,and a body consisting of declarations and a main block.

2.2.1 Declarations

All variables used in a module must be declared before use. Those constants and types which are used in a module and which are not provided by some other module (ie, which do not appear in a needs statement in the specification for the module), must also be declared before use. A declaration associates one identifier, or several, with a definition:

```
declare Max = const integer (:= 25);

declare M, Mp, P : integer;

declare Buff = [1 .. N] of char;
```

(The kind of declaration is determined by the symbol(s) used to separate the identifier(s) from the definition: a constant is declared using the separator "= **const**" and by specifying its type and its value; a type is defined using the separator "=" and a variable with ":" .)

Constants must be of simple type. Variables may be assigned initial values, using initialisation clauses. Arrays are defined by specifying range and component type, and enumerations and records somewhat as in Pascal[6]; sets are defined by specifying the component type, which must be pre-defined.

All the ports provided by a module must be declared in port declarations. A double colon "::" is used as a separator and a ";" or a **reply/response** clause as a terminator in a port declaration. An **oport** is declared with the keyword **send** followed by a message (ie, the set of types of its parameters, and possibly empty); the control rule for the **oport** is then specified by including a **response** clause. An **oport** with no response clause corresponds to the control rule "-"; a response clause is equivalent to the rule "->". The response clause for an **oport** may either be a message, or the name of an **iport**. An **iport** is declared with the keyword **receive**, a parameter set, and a control rule. For a responding **iport**, however, there is a (possibly empty) set of statements between the **receive** and the **reply**. An **iport** with no reply part corresponds to the rule "-"; the presence of a reply part allows either of the control rules "->" or "=>" to be followed and the choice between them is made in the specifications.

```
declare NextChar :: send (C1:char);
    Assemble :: receive (C:char);

declare Query :: send (C1:char) response (B);

declare IsaMember :: receive (J:integer)
            < ... statements ... >
        reply (B);
```

declare GetChar :: send ( ) response Assemble;

declare ToEach :: [1..17] of send (I:integer);

Note:

1. The set of parameters used in a port declaration may consist of pre-defined variables, constants (for output messages), or formal names (qualified by their types).

2. When a port is invoked in a port call (as described below), actual parameters of matching type must be supplied for each formal parameter in a port declaration, and in corresponding order.

3. A parameter which is a module is passed *by reference*, so that both the sender and receiver can now communicate with the same module; *all* other parameters are passed *by value.*

4. The receipt of a message is treated as the assignment of the values of the parameters of the message to local variables.

<ModuleImpl> ::= **module implementation** <Ident> ";" <DeclarationPart> <MainBlock> **end** <Ident> "."
<DeclarationPart> ::= **declare** { <Declaration> ";"}
<Declaration> ::= <ConstDecl> | <TypeDecl> | <VarDecl> | <PortDecl>
<ConstDecl> ::= <IdentList> "=" **const** <SimpleType> <ValueClause>
<TypeDecl> ::= <IdentList> "=" <TypeDef>
<VarDecl> ::= <IdentList> ":" <TypeSpec>
<PortDecl> ::= <IdentList> "::" <PortDef>
<TypeSpec> ::= <TypeDef> | <TypeDef> <ValueClause>
<ValueClause> ::= "(" <AssignClause> ")"
<AssignClause> ::= ":=" <Expr> ["," <Expr>]
<PortDef> ::= <ArrayRange> **of** <PortStmt> | <PortStmt>
<PortStmt> ::= <IportDef> | <OportDef>
<IportDef> ::= **send** <ParameterSet> <Irule>
<Irule> ::= **response** <Ident> | **response** <ParameterSet> |
<OportDef> ::= **receive** <ParameterSet> <Orule>
<Orule> ::= <StatementSet> **reply** "(" <IdentList> ")" |
<ParameterSet> ::= "(" [<ParEl> ["," <ParEl>]] ")"
<ParEl> ::= <Ident> ":" <Ident> | <Ident>

2.2.2 Basic Statements

The main block of a module consists of a (possibly empty) set of statements. If present, statements are preceded by **begin** and separated from each other by ";". There are two kinds of statements, basic statements and communication statements, and we will describe them separately.

<MainBlock> ::= **begin** <StatementSet> |
<StatementSet> ::= <Statement> [";" <Statement>] |
<Statement> ::= <BasicStmt> | <CommStmt>
<BasicStmt> ::= <AssignStmt> | <IfStmt> | <LoopStmt>

The basic statements are the assignment statement, the conditional if statement, and the general **loop** statement. These have forms which are very similar to those of equivalent statements in other languages.

The assignment statement consists of a list of identifiers of one or more different target variables, and an assignment clause with as many expressions separated by commas as there are target variables; starting from the left, each target variable is assigned the value of the corresponding expression.

```
Net := Salary + Perquisites;

Net, Deduct := Net - TaxDeduct, Deduct + TaxDeduct;
```

The **if** statement closely resembles its counterpart in Modula^2, and has a set of boolean expressions which are evaluated in order; the set of statements following the first expression which evaluates to **true** is executed. If none of the expressions is **true**, the **else** part is executed, if it is present.

```
if A > B then A := A - B else B := B - A end if;

if Free then Left := Left - 1
    elseif Queued then Waiting := Waiting + 1
    else Rejected := Rejected + 1 end if;
```

The **loop** statement is used to conditionally or unconditionally repeat a set of statements. The conditional clauses allow the set of statements to be repeated a number of times (equal to the value of an integer expression), or a number of times while the value of a variable is assigned increasing or decreasing values (**for** ...), or as long as a boolean expression has the value **true** (**when** ...).

```
loop (1 .. 20)
    Decr := Decr + Decr/Base;
    Base := Base/10;
end loop;

loop (for I := 2 .. 60 step 2)
    A[I-1] := A[I];
end loop;

loop (while I > 0) I := I - 5 end loop;
```

An **exit** statement unconditionally terminates the execution of a loop and transfers control to the next statement. If **loop** statements are nested and labelled, the **exit** statement can be used with a label designator to transfer control to the statement following the labelled loop. An **exit** statement may appear only in a **loop** statement.

<AssignStmt> ::= <IdentList> <AssignClause>
<IfStmt> ::= if <Expr> then <StatementSet> [elseif <Expr> then <StatementSet>] <ElseClause> end if
<ElseClause> ::= else <StatementSet> |
<LoopStmt> ::= loop <LoopCond> <StatementSet> end loop
<LoopCond> ::= <Range> | <ForClause> | <WhenClause> |
<ForClause> ::= "(" for <Ident> ":=" <Range> <Steps> ")"
<WhenClause> ::= "(" when <Expr> ")"
<UntilClause> ::= "(" until <Expr> ")"
<Steps> ::= step <Expr> |
<ExitStmt> ::= exit | exit <LabelDesig>

2.2.3 Communication Statements

There are five communication statements, each specifying some action(s) to be performed with the ports declared in a module.

<CommStmt> ::= <PortCall> | <ForwardStmt> | <AwaitStmt> | <EnableStmt> | <DisableStmt>

The simplest communication statement is the port call, which causes the statements of the specified port to be executed. For example, if NextChar and Query are **oports**, they can be called as:

NextChar ('F');

if Query(':') **then** IsaLabel := **true else** IsaLabel := **false**

A port call specifies the name of the port and provides actual parameters for each formal parameter in the port declaration. The value returned by a responding **oport** (ie, one with a **response** clause, obeying the control rule "->", as in Query) may also be used in an expression, as for a traditional function call.

A port call to an **oport** will result in a message being sent to the **iport**(s) to which it is connected. If an **oport** is not connected to any **iport**, the action of the **send** has no effect (ie, it is a 'null' statement). A port call to an **iport** will result in the execution of the module being resumed only when a message is received at that port; a port call to an **iport** which is not connected to any **oport** will cause the execution of the module to be permanently suspended (and then terminated). Hence, a call to a responding **oport** which is not connected to any port will also cause the execution of the module to be suspended (as otherwise, the module will remain waiting for a response message which will never come).

A **forward** statement is used to transfer control out of an **iport** *without* executing the normal **reply** part; the **forward** statement names an **oport** through which a message is to be sent and causes the execution of the current **iport** to be terminated. The **iport** which receives a forwarded message will find it indistinguishable from a message which has been sent normally: the reply port for the message received at an **iport** becomes the

reply port for a message forwarded from the **iport** (hence, if module A send a message to module B, with a reply port P in A, a reply will be received at P either from B or from some other port which receives the forwarded message from B).

```
declare Store :: receive ( ... )
        if Full then forward A ( ... )
        else ...
    reply ( ... );
    A :: send ( ... );
```

The **await** statement specifies a (non-null) set of **iports** and causes the execution of the module to be resumed after a message is received at *any* one of these **iports**. If any of the specified **iports** is not connected, execution of the module is suspended.

```
await {NextQuery, Close};

if Empty then await {NextIn}
   elseif Full then await {NextOut}
   else await {NextIn,NextOut};
```

The **enable** statement specifies a set of **iports** at which messages may be received *whenever* the module is awaiting a message at **any iport** (ie, due to a port call, an **await** statement, or when waiting for a response to a **send**). Unlike the **await** statement, the action of the **enable** statement is distributed over all the succeeding communication statements. The **enable** statement permits messages to be received from *all* the 'enabled' **iports** which have messages ready *before* a message is ready at an awaited **iport**. For example, the statements

```
enable {Error};
await {Result};
```

are equivalent to

```
loop await {Error, Result};
   if Lastport = Result then exit end if;
end loop
```

where Lastport is a standard **oport** which returns the name of the last **iport** at which a message was received. A corresponding **disable** statement can be used to remove **iports** from the enabled set.

2.2.4 Comments

1. The port call is the simplest of the communication statements and is used for unconditionally sending or receiving a message.

2. The **forward** statement can be used so that a module can receive a message and then either perform the function associated with the **iport**, or **forward** a message to some other module for further action (eg, for handling errors).

3. The **await** statement is *non-deterministic*, in that if messages are ready at more than one of the awaited ports, *any one* of the messages may be received.

4. The use of an **if** statement with **await** statements provides a simpler form of guarded command[7] than the alternative statement in CSP as the boolean expressions of the **if** statement are evaluated in order and the statements following the first true expression are then executed: non-determinism is therefore restricted to the choice of which ready message to accept.

5. From the point of view of practical programming, there is usually no difficulty in transforming a non-deterministic alternative statement (which can readily be derived from the post condition using the methodology suggested by Dijkstra[8]) into a deterministic **if** statement with non-deterministic **await** statements, and the resulting programming construct is considerably easier to compile efficiently. In terms of CSP, this amounts to restricting non-determinism to the input commands, rather than the boolean expressions, in guards.[1]

6. The **enable** statement allows a simple form of priority to be introduced in the handling of input messages, and is useful both in simplifying program structure (by allowing all 'enabled' ports to be specified at one place so that the subsequent statements are simpler) and for handling exceptional conditions (so that, if these occur, the associated error messages will be received before all other messages).

7. However, it should be noted that exceptions are not defined in the language: an exception can be treated as just another kind of communication - 'raising' an exception then amounts to sending a message through an **oport**, and this message can be received if a module makes a connection to this **oport**. Naturally, most implementations will predefine the names of exception generating modules and the associated **oports**, and exceptions may need to be propogated using a different communication 'scheme' (as we shall see later).

<PortCall> ::= <PortDesig>
<ForwardStmt> ::= **forward** <PortDesig>
<AwaitStmt> ::= **await** <Expr>
<EnableStmt> ::= **enable** <Expr>
<DisableStmt> ::= **disable** <Expr>
<PortDesig> ::= <Ident> | <Ident> <ParList>

---

[1] I am grateful to Fred Schneider for pointing out a significant example where this restriction becomes cumbersome; however, I do not know of any other cases where this is so. In most cases, only one boolean guard of an alternative statement evaluates to **true** at any time and this situation can be simply and correctly expressed by a deterministic **if** statement.

3 Examples

Problem: A module, Clock, sends a (null) message through an **oport** Tick after every time unit has elapsed. Design a module which can be used by other modules for timing multiples of the basic time unit.

Solution: Let Timer be a type module with an **iport**, Count, which can be connected to Clock.Tick (ie, to the **oport** Tick in the module Clock) to count time units. An iport, SetTime, will receive an integer specifying the interval to be timed, and an **oport** Alarm will send a (null) message when the interval has elapsed.

```
module type Timer;
    needs oport Clock.Tick;
    provides iport Count ( ) -> ( ); SetTime (integer);
        oport Alarm ( );
    connect (Count, Clock.Tick);
end Timer.

module implementation Timer;
    declare Interval : integer ( := - 1);
        Count :: receive ( )
                    if Interval >= 0 then Interval := Interval - 1 end if;
                 reply ( );
        SetTime :: receive (Interval);
        Alarm :: send ( );
    begin
        loop await {Count, SetTime} ;
            if Interval = 0 then Alarm end if;
        end loop;
end Timer.
```

All instances of Timer will be connected to the **oport** Clock.Tick. A module can set an interval of time after which it wishes to receive an alarm by sending a non-negative integer to the **iport** SetTime. Timer stores this number in the variable Interval and decrements it at every 'tick' of the clock; when the interval reaches zero, it sends a message through Alarm.

Problem: A minor road crosses a major road at a junction with traffic lights which are normally green for traffic on the major road. Pressure pads on the approaches of the minor road to the junction send a signal whenever they are depressed. When the *first* vehicle crosses the pressure pads while the lights are red for traffic on the minor road, the lights must change after 20 time units and remain green for 15 time units. (Note: there is never much traffic on the minor road).

Solution: Let us call the major road NS and the minor road EW. The traffic lights have two states: green for NS and green for EW. The pressure pads cause a change of state only when the first vehicle crosses them while the lights are red for EW. Let Pads be a module with **oports** EW1 and EW2 which send the signals from

the pressure pads, and NSGreen and EWGreen be iports in a module Lights. An instance of the module Timer will be used to measure the waiting intervals.

```
module specs Signals;
    needs module Timer; oport Pads.EW1, Pads.EW2;
        iport Lights.NSGreen, Lights.EWGreen;
    provides module T:Timer; oport EWtoGreen ( ); NStoGreen ( ); Set(integer);
        iport PadSignal ( ); Timesup ( );
    connect (PadSignal, Pads.EW1, Pads.EW2); (Set, T.SetTime); (Timesup, T.Alarm);
        (EWtoGreen, Lights.EWGreen); (NStoGreen, Lights.NSGreen);
end Signals.

module implementation Signals;
    declare T : Timer;
        EWtoGreen,NStoGreen :: send ( ); Set :: send (I : integer);
        Timesup :: receive ( ); PadSignal :: receive ( );
    begin
        loop
            NStoGreen; await {PadSignal};
            Set (20); await {Timesup};
            EWtoGreen; Set (15);
            loop
                await {PadSignal, Timesup};
                if LastPort = Timesup then exit end if;
            end loop;
        end loop;
    end Signals.
```

The main block has a loop to set the initial condition (NStoGreen) and then to wait until a pad is depressed. When this occurs, the lights are switched after 20 units and the time interval is reset to 15 units. The second loop reads and ignores any further pad signals until the time interval has passed (LastPort is a standard **oport** which returns the name of the last port at which a message was received). The solution can be easily extended to switch the lights either when a vehicle crosses the pressure pads, or after a fixed time interval, whichever occurs sooner.

4 Module Lifetimes and Termination

A module is *created* when it is instantiated; except for **type** modules, modules are instantiated implicitly when the program of which they are part is created. A **type** module must be instantiated in some other module to be created. When a module is created, its declarations are performed, all (local) module instances which may be declared in that module are created, and the connections described in its specifications are made. The module is then said to be *initialised*, as are all its local module instances. An initialised module is ready to execute the statements of its main block.

A module continues execution until

1. It has completed execution of the last statement in its main block (a module with an empty main block is treated as if it has executed a single 'null' statement), or

2. It attempts to receive a message at a **iport** which is unconnected to any **oport**.

When either of these conditions occurs, a module suspends execution and becomes *comatose* and all connections are broken from and to the ports of *that module* (note that this does *not* automatically cause its local modules to become comatose). A module remains comatose as long as it has any local modules which are either executing or comatose and is terminated only when all its local modules have been terminated. Thus a module cannot be terminated while its own variables are still in execution; the variables of a comatose module will themselves become comatose if they attempt to communicate with the module and can then be terminated if they have no executing variables of their own.

A module therefore has four stages in its existence: *created, initialised, executing,* and *comatose.* The stages of this existence may be dependent on the stages of existence of other modules. For example, we have already seen that a comatose module cannot be deleted until all its local modules have been terminated. On the other hand, an executing module may become comatose because some other module with which it is communicating has ceased execution. From the point of view of an implementation, whenever one module is terminated a check must be made to see if the condition for terminating any other module has now been met.

4.0.1 Example

Problem: (cf. Hoare[3]) Two square matrices or order N are to be multiplied. One matrix is already stored with its individual elements in the elements of a square array of modules. The other matrix is to be read in columns, multiplied with the stored matrix, and the product matrix is to be output in rows. "After an initial delay, the results are to be produced at the same rate as the input is consumed"[3].

Solution: Let West be a vector of modules each of which has an **oport**, Next, which delivers a real number corresponding to an element of the column of the array to be multiplied. Let North be a module which has an **oport**, Zero, which sends zeroes. The elements of the array of multipliers, Cn, are modules of type Node which receive a real number at the **iport** FromW, and a partial sum from an **iport** FromN, and send a new partial sum through an **oport** ToS. A vector of modules, South, each with an **iport**, Result, receives the rows of the product matrix. It is assumed that West and South are pre-defined.

```
module type Node;
    provides iport FromW(real); FromN(real);
        oport ToS(real);
end Node.

module implementation Node;
    declare R,R1,R2:real;       % R is the stored element\
        FromW, FromN :: receive (El:real);
        ToS :: send (Res:real);
    begin
        loop (1..N)
            FromW (R1); FromN (R2); ToS(R*R1 + R2);
        end loop
    end Node;

module specs Multiplier;
    needs oport West.Next[1..N] (real); North.Zero (real);
        iport South.Result[1..N] (real);.
    provides module Cn : [1..N, 1..N] of Node;
    connect (West.Next[1..N], Cn.FromW[1..N,1] .. [1..N,N]);
        (North.Zero, Cn.FromN[1,1] .. [1,N]);
        (Cn.ToS[1,1..N] .. [N-1,1..N],Cn.FromN[2,1..N] .. [N,1..N]);
        (Cn.ToS[N,1..N], South.Result[1..N]);
end Multiplier.

module implementation Multiplier;
    declare Cn : [1..N,1..N] of Node;
end Multiplier.
```

This solution requires N(N + 2) + 1 modules, compared to the N(N + 4) processes required for a similar solution in CSP; this reduction is made possible because of the ability to connect one port (eg, North.Zero, or West[1].Next) to a set of ports (ie, Cn[1,1] .. Cn[1,N], and Cn.FromW[1,1] .. Cn.FromW[1,N]). The indices used in the **connect** statements clearly indicate the types of connections: ie, between columns and rows (West to the rows of Cn), and rows and rows (each row of Cn to the succeeding row). Multiplier is an example of a module with no main block. It becomes comatose as soon as it is initialised. However, the array of local modules, Cn, will execute the basic loop N times. The module, Multiplier, will be terminated when each module in the array, Cn, is terminated.

5 Module Creation

The modules we have described so far have been created as parts of programs, or as variables in other modules. In addition to this, modules may be *dynamically* created using the **make** statement. A dynamically created module must be referred to by a name which is associated with a module type by a special declaration.

```
declare NewMod, NewNode : *ModType;
```

A "*" in a declaration (or a provides statement) denotes a dynamically created module. NewMod and NewNode will be referred to as dynamic variables: note that this declaration does *not* create a module.

A set of modules may be dynamically created by the **make** statement:

make {NewMod, NewNode};

which results in the names NewMod and NewNode being associated with dynamically created modules of type ModType. A dynamic module is declared in a **provides** statement:

provides NewNode : *Node;

and its ports can be connected as usual:

connect (Out, NewNode.Inp);

This connection will be made between the port Out and the port Inp *whenever* a dynamic module is associated with NewNode. (Since the receipt of a message is equivalent to the assignment of the values of the message to local variables, a dynamic connection can also be made for modules passed as parameters in messages).

```
declare NewNode : *ModType;
    Out :: send ('String');
    Get :: receive (Mod : ModType);
begin
    make {NewNode};          %NewNode is now associated with a new module\
    Out;                                                                  [L1]
    Get (NewNode);           %NewNode is now associated with a module\
    Out;                     %passed as a parameter\                      [L2]
end
```

At each of the statements L1 and L2, the variable NewNode refers to a different instance of ModType. Hence, two different dynamic connections are made between the port Out, and the port Inp in each instance of ModType, and the two port calls to Out will send the message ('String') to two different modules.

A corresponding **break** statement can be used to disconnect a set of dynamically created modules:

break {NewNode, NewMod};

A dynamic variable can only be associated with one module at a time: therefore, if a dynamic variable was associated with a module, its connections must first be broken before the variable can be associated with a new module.

A dynamically created module is treated like a module created by a declaration; however, a dynamically connected module can be disconnected in two ways, by a **break** statement or when its creating module becomes comatose. Note that the termination rule for modules still allows the problem of dangling references

to be avoided. The legality of port connections can be checked (at compile time) as a dynamic variable can only be associated with modules of one type.

Problem: (cf Hoare[3], attributed to D. Gries) Construct a parallel version of the Sieve of Erastothenes; each element of the sieve will first receive, and then print, a prime number from its nearest neighbour on the left. It will then receive a stream of integers; every integer which is a multiple of the prime must be suppressed, and all other numbers must be sent to its nearest neighbour on the right. Print all the prime numbers between 2 and 10000.

Solution: Let the elements be modules of type Sieve; each module has an **iport** FromLeft which reads in integers sent by the neighbouring module on the left. The first number, which is also a prime, is output through the **oport** ToPrint and all succeeding numbers which are multiples of this number are suppressed. When the module first encounters a number which is not a multiple of its prime, it creates a new instance of Sieve and sends the number through its **oport** ToRight, which is dynamically connected to the **iport** FromLeft of the new module. The starting sequence of numbers and the first instance of Sieve are created in a module Primes; all subsequent instances of Sieve are dynamically created and there will be as many such instances as there are prime numbers between, in this case, 3 and 10000.

```
module type Sieve;
    needs iport Print.Out (integer);
    provides module Next : *Sieve; iport FromLeft (integer);
        oport ToRight (integer); ToPrint (integer);
    connect (ToRight, Next.FromLeft); (ToPrint, Print.Out);
end Sieve.

module implementation Sieve;
    declare M,Mp,P : integer; Next : *Sieve; First : boolean;
        FromLeft :: receive (N :integer);
        ToRight :: send (M); ToPrint :: send (P);
    begin
        FromLeft (P); ToPrint; Mp := P; First := true;
        loop
            FromLeft (M);
            loop (while M > Mp) Mp := Mp + P end loop;
            if M < Mp then
                if First then make {Next}; First := false end if;
                ToRight;
            end if;
        end loop;
    end Sieve.
```

```
module specs Primes;
    needs type Sieve; iport Print.Out;
    provides iport S.FromLeft(integer);
        oport P1(integer); ToP(integer);
    connect (ToP, Print.Out); (P1, S.FromLeft);
end Primes.

module implementation Primes;
    declare S : Sieve; R : integer;
        P1 :: send (R); ToP :: send (2);
    begin
        ToP;
        loop (for R := 3..10000 step 2) P1 end loop;
    end Primes.
```

The module Primes will send out a stream of odd numbers from 3 to 9999 and then become comatose; its connections will be broken, and the first instance of Sieve, S, will therefore become comatose when it attempts to read the next number through its port, FromLeft. This will be repeated for all the dynamic instances of Sieve; the last of these will not itself have created any instance of Sieve and will be terminated as soon as it becomes comatose. Once this is done, the previous module can be terminated, and so on. The dynamic instances of Sieve will all be terminated in the order opposite to that of their creation and then the program Primes can itself be terminated.

6 Building a Binary Tree

Problem: A company keeps information about each of its employees in the form of a record with the fields Ident, Name, Age and Sex. The records are to be kept sorted by Ident, which is a unique integer for each employee. The operations to be done are Insert, to add a record; Search, to find the entry corresponding to a name; Delete, to remove the entry for a name; and, List, to print out all entries in sorted order.

Solution: Let the employee data be stored in the form of a binary tree, with each node having one record of type EmplRec, and variables Left and Right as the subtrees which extend from the node.

```
EmplRec = record
        Ident : integer; Nm : Name;
        Age : integer; S : Sex
        end EmplRec;
```

All nodes can be of identical type, TreeNode, with the operations performed by the **iports** Insert, Delete, Search and List.

1. *Insert*: A new record can only be inserted in a node which is empty (ie, one whose EmplRec has zero in the Ident field). According to whether the Ident for the new entry is less or greater than the Ident for a non-empty node, the new entry will be on the left or right subtree for the node. A

non-empty node which has no descendants makes a *pair* of new nodes and forwards the entry to the appropriate one.

2. *Delete*: The record to be deleted is identified by name and may be anywhere in the tree: if it is at a leaf (ie, a node whose references to subtrees are both nil), the node can be merely be made empty; if, however, it is elsewhere in the tree, the record which is at the root of a subtree originating at the required node should replace the record to be deleted. This must be done iteratively down the subtree until, finally, a leaf node is made empty. A node which has two empty descendants can disconnect them both (ie, using the **break** statement). The Delete operation makes use of an auxiliary operation, ReadLeft (or ReadRight), to find which subtree should be moved up and to detect empty descendant nodes.

3. *Search*: If a node contains the required record, the search is over (and the reply is **true**); otherwise, (a) if the node has no descendants the reply is **false**, or (b) the search must continue down both subtrees and the reply is the logical **or** of the replies from the subtrees.

4. *List*: To print the entries with Ident in ascending order, print the left subtree (if there is one), the node (if not empty), and then the right subtree (if there is one). Repeat for all nodes.

```
module type TreeNode;
    needs type EmplRec; iport Print.Entry;
    provides module Left, Right : *TreeNode; iport Insert (EmplRec); Delete (Name);
            Search (Name) => (boolean); List ( ) => ( ); Read ( ) => (EmplRec);
        oport InsertLeft, InsertRight (EmplRec); DeleteLower (Name);
            DeleteLeft, DeleteRight (Name); ReadLeft, ReadRight ( ) -> (EmplRec);
            NextLevelSearch (Name) -> (boolean); Display (EmplRec);
            ListLeft, ListRight ( ) -> ( );
    connect (InsertLeft, Left.Insert); (InsertRight, Right.Insert);
        (DeleteLeft, Left.Delete); (DeleteRight, Right.Delete);
        (ReadLeft, Left.Read); (ReadRight, Right.Read);
        (DeleteLower, Left.Delete, Right.Delete); (Display, Print.Entry);
        (NextLevelSearch, Left.Search, Right.Search);
        (ListLeft, Left.List); (ListRight, List.Right);
end TreeNode.

module implementation TreeNode;
    declare Rec : EmplRec; Left, Right : *TreeNode; Sons, Found : boolean (:= false);
        Insert :: receive (R : EmplRec)
                if Rec.Ident = 0 then Rec := R;
                    else if ¬Sons then make {Left, Right}; Sons := true end if;
                        if R.Ident < Rec.Ident then forward InsertLeft (R)
                            else forward InsertRight (R) end if;
                end if;
            reply ( );
```

```
    Delete :: receive (N : Name)
            if Rec.Nm ≠ 0 then
                if Rec.Nm ≠ N then forward DeleteLower (N)
                    elseif ¬Sons then Rec.Ident := 0
                    else Rec := ReadLeft;
                        if Rec.Ident ≠ 0 then DeleteLeft (Rec.Nm)
                            else Rec := ReadRight;
                                if Rec.Ident ≠ 0 then DeleteRight (Rec.Nm)
                                    else break {Left, Right}; Sons := false end if
                        end if
                end if
            end if
        reply ( );
    Search :: receive (N : Name)
            if Rec.Nm = N then Found := true
                elseif Sons then NextLevelSearch (N); Found := B;
                    await {Result}; Found := Found or B
                else Found := false
            end if
        reply (Found);
    List :: receive ( )
            if Sons then ListLeft; Display; ListRight
                elseif Rec.Nm ≠ 0 then Display end if
        reply ( );
    Display :: send (Rec);
    ListLeft, ListRight :: send (.) response ( );
    Read :: receive ( ) reply (Rec);
    InsertLeft, InsertRight :: send (ER : EmplRec);
    ReadLeft, ReadRight :: send ( ) response (ER : EmplRec);
    DeleteLower, DeleteLeft, DeleteRight :: send (N : Name);
    NextLevelSearch :: send (N : Name) response Result;
    Result :: receive (B);
begin
    loop
        await {Insert, Delete, Search, List, Read}
    end loop
end TreeNode.
```

Inserting a new record takes log n operations, assuming the tree is balanced and has n nodes, and may involve the creation of two nodes. This is no better, and no worse, than for a sequential program. However, a number of new records can be inserted in 'pipe-lined' fashion and each subsequent insertion takes only 1 step (the Insert operation has no waiting, so the caller is free to continue execution). Deleting a record is not as simple: in the worst case (deleting the root of the tree), a single deletion may trigger off (log n) - 1 other deletions, each of which requires one or two Read operations. However, note that for the Delete operation, as for the Search, finding the required element is done in parallel (each node forwarding the request to its descendants, with a 'fan-out' of 2); the Delete operation may also be pipelined but with reduced parallelism as

the subtree originating in the node to be deleted will be 'blocked' until the entire Delete operation is over. Insert and Delete operations may even be pipelined and interleaved in any order and, subject to the restriction for parallelism for the Delete operation, they can be executed as fast as 1 operation per step.The Search operation takes a time of 2 log n steps, as the request has to travel down the tree and the responses (Result) have to be sent back up, each node combining the responses of its two descendants. These results compare reasonably well with those given by Bentley and Kung for their tree-structured parallel computer[9]. The List operation is relatively simple: since records are sorted by Ident when they are inserted, the sorted list of records can be printed by traversing the nodes in order; naturally, this takes n steps. (For both Search and List, the value of n must include the empty nodes).

7 Reliable Communication on Unreliable Lines

Problem: Two programs communicate with each other by sending messages across unreliable lines. Communication may be in one direction, or both, and the propogation time on the lines is such that the transmission of N messages can be initiated in the time it takes one message to go to the other end. A standard operation Checksum is available to verify if a message has been mutilated in transmission. Devise a means for messages to be sent from one program to the other without errors or change of order.

Solution: Since communication may be one way or both ways, transmission and receipt of messages must be kept separate to ensure that these operations can be performed independently. Let Transmitter be the module which accepts messages from a program for transmission across the lines, and Receiver be the module which receives messages from the lines. Every message which is received must be checked (using Checksum); if it has been corrupted, a message must be sent to the other end asking for a re-transmission. Since N messages can be sent in the time it takes one message to go from one end to the other, the acknowledgement for a message will take a time equal to the transmission time for 2N messages. Let us assume that messages are much longer than their acknowledgements, so that it is worthwhile sending acknowledgements 'piggy-back' along with outgoing messages.

Transmitter will need to keep a buffer of at least 2N messages if the line is to be efficiently used as a message can only be deleted after it has been acknowledged. The buffer requires three pointers: NextFill, NextOut, and NextAck, to indicate the positions of the next empty slot, the next message to be transmitted, and the next message to be acknowledged, respectively. A variable, MCount, records the number of unacknowledged messages still in the buffer. Transmitter will require an **iport**, FromProg, to receive messages for output from the program and another, FromRec, to receive acknowledgements from its Receiver. It will have an **oport**, ToLine, for sending messages to the line. However, since communication can be in one direction or both, and since acknowledgements are sent 'piggy-back' with outgoing messages, it is

possible that acknowledgements will get held up at one end if there are no outgoing messages. To avoid this, Transmitter must also send a 'null' message if any acknowledgement is pending without an outgoing message in the buffer.

We will assume that the constants N and Max (= 2N), and the types Status = (Ack, Nack), Message and MaxMO = 0 .. 2N are defined elsewhere.

```
module type Transmitter;
    needs constant N, Max; type Status, Message, MaxMO;
    provides iport FromProg(Message) => ( ); FromRec(MaxMO, Status, MaxMO, Status);
        oport ToLine (MaxMO, Status, MaxMO, Message) -> ( );
end Transmitter;

module implementation Transmitter;
    declare Buffer : [1 .. Max] of Message;
        NextOut, NextFill, NextAck : 1 .. N (:= 1, 1, 1);
        RecStat : Status; MCount, RecNum : MaxMO(:= 0, 0);
        FromProg :: receive (M : Message)
                Buffer[NextFill] := M; MCount := MCount + 1;
                NextFill := (NextFill mod Max) + 1;
            reply ( );
        FromRec :: receive (AckNum : MaxMO; AckStat : Status; RecNum; RecStat);
                if AckNum ≠ 0 then
                    if AckStat = Ack then
                        NextAck := (NextAck mod Max)+1;
                        MCount := MCount - 1;
                    else NextOut := AckNum end if;
                end if;
            reply ( );
        ToLine :: send (RecNum; RecStat; Num2 : MaxMO; M2 : Message) response ( );
    begin
        loop
            if MCount = Max then
                await {FromRec}
            else await {FromProg, FromRec} end if;
            if NextOut ≠ NextFill then
                ToLine (NextOut,Buffer[NextOut]);
                NextOut := (NextOut mod Max) + 1;
            else ToLine (0, null) end if;
        end loop;
    end Transmitter;
```

Transmitter has the iport FromProg to receive messages from the program provided MCount < Max. The iport FromRec is to be connected to the Receiver module for that program, which will supply the number, AckNum, and the status, AckStatus, of the last *acknowledgement* received from the other program and the

number and status of the last *message* received (these are stored directly in RecNum and RecStatus, for transmission to the other program through the **oport** ToLine). We use the convention that a message number is set to zero when the status is to be ignored. A message (and an acknowledgement, if there is one) is sent to the line whenever there is one pending in the buffer (ie, NextOut ≠ NextFill); if just an acknowledgement is pending, it is sent with a null message. Note that if the acknowledgement (AckStat) from the other program indicates that a message (AckNum) was correctly received, MCount is decremented and NextAck updated; otherwise, all messages from that one onwards are to be retransmitted (by setting NextOut to the value of AckNum).

Receiver has **iports** ToProg, to supply the next message to the program, and FromLine, to receive input from the line; messages ready for the program are stored in a buffer of size Max. A variable RCount records the number of correct messsages in the buffer. Input from the line may consist of acknowledgements and/or messages; acknowledgements to messages sent by the Transmitter of that program are AckNum, the message number, and AckStat, the status, and are just to be sent to Transmitter. An incoming message (M) from the other program has to be checked by passing it to the **oport** Checksum. If a message has been received without error, it can be added to the buffer; however, if there is an error, no further messages should be stored in the buffer until that message has been correctly re-transmitted (Num2 is the number of the next message and it must be the same as the buffer pointer NextIn if the message is the next to be stored, or zero if there is no message (ie, a null message with a real acknowledgement)).

```
module type Receiver;
    needs constant N, Max; type Status, Message, MaxMO;
    provides iport FromLine (MaxMO, Status, MaxMO, Message) -> ( );
            ToProg ( ) => (Message);
        oport ToTrans (MaxMO, Status, MaxMO, Status);
end Receiver.

module implementation Receiver;
    declare Buffer : [1 .. Max] of Message;
        NextIn, NextToProg : 1 .. Max (:= 1, 1);
        AckStat, RecStat : Status;
        AckNum, MCount : 0 .. 2N (:= 0, 0); M : Message;
        ToTrans:: send (AckNum; AckStat; Num1 : MaxMO; RecStat);
        CheckSum :: send (M) response (RecStat);
        ToProg :: receive ( )
                M := Buffer[NextToProg]; MCount := MCount - 1;
                NextToProg := (NextToProg mod Max) + 1;
            reply (M);
        FromLine :: receive (AckNum; AckStat; Num2 : MaxMO; M);
                if Num2 ≠ 0 then CheckSum end if;
                if (RecStat = Ack) and (Num2 = NextIn) then
```

```
                Buffer[NextIn] := M; MCount := MCount + 1;
                NextIn := (NextIn mod Max)+1;
                ToTrans(Num2);
            else ToTrans( 0 ) end if;
        reply ( );
begin
    loop
        if MCount = 0 then await {FromLine}
            elseif MCount = Max then await {ToProg}
            else await {FromLine, ToProg}
    end loop;
end Receiver.
```

Finally, the two module types can be instantiated in the modules representing the programs which need to communicate across the lines. Let Trans1 and Rec1 be the instances in program 1, and Trans2 and Rec2 the instances in program 2. The **oport** ToTrans in Rec1 (or Rec2) must be connected to the **iport** FromRec in Trans1 (or Trans2) so that acknowledgements can be sent for each message which is received. Program 1 will send its messages to the **iport** FromProg in Trans1: this is a blocking send, as the control rule in the specifications for this port is "=>". Program 1 can only continue when the message has been accepted, and that will be when there is place in the buffer. Program 1 can receive incoming messages from the **iport** ToProg in Receiver: this is also a blocking send, so the program will wait until a message has been received. Note that the line is transparent to the Transmitter and Receiver: in fact, the **oport** ToLine in Trans1 (or Trans2) can be directly connected to the **iport** FromLine in Rec1 (or Rec2) and the properties of the lines can be represented in a communication scheme which is used for these ports.

8 Ports as Modules

> "We shall call a group of definitions and programs which cover a notion such as this an *implementation scheme*, or simply a *scheme*[10]."

We have so far considered ports as pre-defined types which are created by declarations in modules. It is fairly easy to see that the declaration of a port accomplishes more that a simple declaration as it associates a control rule and, for **iports**, possibly a set of statements with the port. The syntax of a port declaration provides a framework in which the operations of the port are performed.

We can also view ports as data types which have a limited set of operations which can be performed on them. Some of these operations are explicitly stated whenever ports are used : **connect**, **send** and **receive** (note that **reply**, **response** and **forward** are basically syntactic variations of the last two operations); another has been referred to in connection with module lifetimes and the **break** statement, and is **disconnect** (which is performed on all the ports of a module when it becomes comatose) . Another operation is necessary for the

await statement, to indicate if a port has a message ready. Since there is no inherent reason why these operations cannot be programmed in the language, we can even consider a port as a module (with a set of **oports** and **iports** for each of the operations **send**, **receive**, **connect** and **disconnect**) which is *instantiated* at each port declaration.

To illustrate how ports *might* be implemented, let us consider a version which would be suitable for distributed systems in which each processor has a uniform method for communicating with any other processor. Since, in general, an **oport** may be connected to a set of **iports**, an output message may need to be sent to several different destinations any of which may delay accepting it; we would also like communication to be as asynchronous as the control rule for a port will allow. Both these requirements point to the need for maintaining buffers for ports. We will therefore first consider the design of a suitable buffer module.

The module Buffer will contain an array of Storage to hold messages: this will allow a sending module or a receiving module to execute as much in parallel with the transmission and receipt of messages as possible (the size of the array can be a parameter). The obviously necessary operations are Save (to put a new message into the array), Read (to non-destructively read a message from the array) and Clear (to delete a message in the array); an additional operation Restore allows a partially transmitted message (ie, one which has been sent to some but not all of its intended destinations) to be stored until it can be sent to its remaining destinations. Two operations, Empty and Full, allow the state of the Buffer to be tested.

The definitions of Buffer and Port are mutually dependent, as we will see that the destinations of messages stored in a buffer are ports, and that ports require buffers for storing messages. (From the point of view of compilation, this would require the specifications for Buffer and Port, the implementation of Port, and the implementation of Buffer to be compiled in that order). The **oport** Next, used in the implementation of Buffer, takes a set of 1 .. N and returns the value of one of its elements, according to an algorithm we will not consider here.

```
module type Buffer;
    needs module Port; type BufEl; const N;
    provides iport Save(BufEl) => ( ); Read ( ) => (BufEl);
            Restore(BufEl) -> ( ); Clear ( ) -> ( );
            IsEmpty, IsFull ( ) -> (boolean);
end Buffer.
```

```
module implementation Buffer;
    declare Vacant, Filled : set of 1 .. N (:= {1, .. N}, { });
        NextEl : 1 .. N; Storage : [1 .. N] of BufEl;
        Save :: receive (M : BufEl)
                NextEl := Next(Vacant); Storage[NextEl] := M;
                Vacant := Vacant - {NextEl}; Filled := Filled + {NextEl};
            reply ( );
        Read :: receive ( ) reply (Storage[Next(Filled)]);
        Restore :: receive (M : BufEl)
                Storage[Next(Filled)] := M;
            reply ( );
        Clear :: receive ( )
                Vacant := Vacant + {Next(Filled)};
                Filled := Filled - {Next(Filled)};
            reply ( );
        IsEmpty :: receive ( ) reply (Filled = { });
        IsFull :: receive ( ) reply (Vacant = { });
        PSet : set of iport (:= {Clear, IsEmpty, IsFull});
    begin
        loop
            if Vacant = { } then await (PSet + {Read})
                elseif Filled = { } then await (PSet + {Save})
                else await (PSet + {Read, Save}) end if;
        end loop
end Buffer.
```

Vacant and Filled are sets of 1 .. N representing the elements of Storage: initially, Storage is empty (so Filled = { }) and for each entry stored in the array, one element of Vacant is added to Filled. The main block of Buffer resembles the operation of a bounded buffer: if Vacant = { } then no further entries can be accepted, if Filled = { } then no entries can be read, otherwise entries can be stored and removed. PSet contains the names of the **iports** at which messages will always be accepted.

We can now define the module Port: we will assume that PortType is a compile-time parameter of type ($Iport, $Oport). Port will require **iports** for the basic operations $Connect, Disconnect, $Send, and $Receive, and we will define a single module which can serve as both kinds of port.

The $Connect operation is to be used to connect two ports together (since all multi-way connections can be reduced to sets of two-way connections); as either of two connected modules may become comatose, a record of all connections must be kept in both modules (or, to be more precise, in the ports of both modules). The $Connect operation therefore has two parts: to add a new port to the set, PS, of ports connected to a port, and to send the name of the port to the port it is to be connected to. The second part requires the use of a distinct operation, AddLink. Similarly, corresponding to the Disconnect operation, another distinct operation, RemLink, is used to remove a port from the set of connections of the port it is connected to. Both these operations make use of a standard operation, This, which returns the name of the current module.

The $Send operation must eventually cause the general send to be performed: from one oport to the set of iports it is connected to. To do this with a high degree of parallelism, the message is first stored in the Buffer, along with the set of ports it is to be sent to (thus, unless the Buffer is full, the sending module is then free to continue execution). The main block of Port performs the actual transmission of the message, by reading it from Buffer and sending it with the name of the transmitting Port (using the Transfer **oport**) to as many of its destination ports as are not full; when a message has been sent to all of its intended destinations, it can be deleted (by the operation Clear, in Buffer), but if any of the destinations is full, the message must be sent back to Buffer (Buf.Restore).

With the use of a Buffer in Port, the $Receive operation becomes very simple: it will just wait until there is a message available in the Buffer. If more than one message is ready in the Buffer, one of them is chosen according to the value returned by the **oport** Next (this function could therefore serve to arbitrate between messages and/or senders of different priorities). Note that it is possible that the Buffer is full when the $Receive operation is invoked, so that some senders may potentially have been unable to transfer messages into the Buffer; WasFull is a boolean indicating this condition and if it is set, a message is sent to all connected **oports** through the **oport** Signal (in this simple implementation, this could result in unnecessary communication, but we will ignore that here).

```
module type Port;
    needs module Buffer; type MessageType; const BufSize, PortType;
    provides type StorageEl; module Buf : Buffer; Dest : *Port;
        iport $Connect (Port) -> ( ); Disconnect (Port) -> ( );
            AddLink (Port) -> ( ); RemLink (Port) -> ( );
            $Send (MessageType, Port) => ( ); TryAgain ( );
            $Receive (MessageType, Port) => ( );
        oport SaveInBuf (StorageEl) -> ( );
            MakeLink (Port); BreakLink(Port); PortFull ( ) -> (boolean);
            Transfer(MessageType, Port) -> ( ); Signal ( );
    connect (BufEmpty, Buf.IsEmpty): (BufFull, Buf.IsFull); (BufRead, Buf.Read);
        (BufRestore, Buf.Restore); (SaveInBuf,Buf.Save); (BufClear,Buf.Clear);
        (MakeLink, Dest.AddLink); (BreakLink, Dest.RemLink);
        (Transfer, Dest.Buf.Save); (PortFull, Dest.Buf.IsFull);
        (TryAgain, Dest.Signal);
end Port.
```

```
module implementation Port;
    declare StorageEl = record
                M1 : MessageType;
                MPR : Port
                MPS : set of Port;
                end StorageEl;
        Buf : Buffer(StorageEl, BufSize); Dest : *Port;
        PS, NPS1, NPS2 : set of Port (:= { });
        StEl : StorageEl; B, WasFull : boolean ( := false, false);
        $Connect :: receive (Dest);
                PS := PS + {Dest}; MakeLink;
            reply ( );
        Disconnect :: receive (Dest);
                PS := PS - {Dest}; BreakLink;
            reply ( );
        MakeLink, BreakLink :: send (This);
        AddLink :: receive (To : Port)
                PS := PS + {To};
            reply ( );
        RemLink :: receive (From : Port);
                PS := PS - {From};
            reply ( );
        $Send :: receive (M : MessageType; R : Port);
                StEl.M1 := M; StEl.MPR := R; StEl.MPS := PS;
                SaveInBuf (StEl);
            reply ( );
        $Receive :: receive ( )
                if PS = { } then Abort
                    elseif BufFull then WasFull: = true end if;
                BufRead (StEl); BufClear;
            reply (StEl.M1, StEl.MPR);
        Transfer :: send (CopyEl : StorageEl) response ( );
        SaveInBuf :: send (NewEl : StorageEl) response ( );
        BufEmpty, BufFull :: send ( ) response (B);
        BufClear :: send ( );
        BufRead :: send ( ) response (NextEl : StorageEl);
        BufRestore :: send (OldEl : StorageEl);
        PortFull :: send ( ) response (B);
        Signal :: send ( );
        TryAgain :: receive ( );
```

```
begin
    loop
        if PortType = $Oport then
            await {$Connect, Disconnect, AddLink, RemLink, $Send, TryAgain};
            if not BufEmpty then
                BufRead (StEl); NPS1 := StEl.MPS; NPS2 := NPS1;
                loop (while NPS2 ≠ { })
                    Dest := Next(NPS2);
                    if not PortFull then
                    StEl.MPS := {This};
                        Transfer (StEl); NPS1 := NPS1 - {Dest} end if;
                    NPS2 := NPS2 - {Dest};
                end loop;
                if NPS1 = { } then BufClear
                else StEl.MPS := NPS1; BufRestore (StEl) end if;
            end if;
        else await {$Connect, Disconnect, AddLink, RemLink, $Receive};
            if WasFull then
            NPS1 := PS; WasFull := false;
            loop (while NPS1 ≠ { })
                Dest := Next(NPS1);
                Signal; NPS1 := NPS1 - {Dest};
            end loop
        end if
    end loop
end Port.
```

8.1 Comments

Operations using ports then amount to little more than operations on modules which implement ports, exactly as operations on, say, TreeNode are operations defined in the module TreeNode; the difference arises only because the operations on ports are not directly invoked by port calls but by statements in the language. This creates the possibility, given some means, of associating different implementations of ports with modules, depending on particular requirements. For example, it is usually necessary to implement communication with peripheral devices differently from normal inter-module communication even when, conceptually, they can be treated equivalently ('special' handling of this kind can be seen in Modula[2], with device modules and device processes, and in Ada[11], when interrupts are to be associated with entry calls on tasks).

This would itself serve a useful purpose, as it permits different communication needs to be met without burdening the language with new features and constructs. But we can go beyond this. Many of the languages recently proposed for distributed or parallel processing appear to have been designed with an underlying architecture in mind as they make no provision for specifying how different components are physically

distributed or how communication takes place if the architecture is not uniform. Because of this, CSP would be difficult to implement on a system which did not allow uniform interconnection between processes (either on a single processor, or on a distributed system with one process per processor and full interconnection), as the usual requirement for shared memory has been replaced by the need for a shared intercommunication structure. Similarly, DP has been designed so that the single process on each processor of a distributed system would interleave its own processing with serving requests for the execution of remote procedure calls from other processes. (The 'guardians' of Clu[12] can also be seen to have a similar structure if we substitute a guardian for the DP process and a Clu process for each DP procedure). Dependence on a particular architecture can be a major restriction as it seems likely that there will be many more differences in architectures than in the features of programming languages, so each such language will have to rely on the ingenuity of its implementors in making visible, or invisible (if that is preferred), the interconnections of the hardware and the way processes are mapped onto processors. The use of different implementations of ports is one way for languages for parallel or distributed computing to be both independent of the particular architectures they are implemented on and capable of being used efficiently on a variety of architectures.

We can refer to each implementation of ports as a *communication scheme* and visualise a port declaration as an instantiation of a port type. It is usually of no consequence to the instructions of a module how ports are going to be implemented, so the choice of communication scheme is made in the module specifications. For example, if Ready is an **oport** which sends a boolean message when a line printer is ready to accept another line for printing through its **iport** Buff, and Print is an **iport** to which users can send lines for printing, we can specify a line-printer device driver as:

```
module specs LPDriver;
    needs type Line; iport LP.Buff; oport LP.Ready;
    provides iport Print(Line) -> ( ) using InterProcess;
            Next( ) using InterruptLevel;
        oport Output(Line) using InterruptLevel;
    connect (Next, LP.Ready); (Output, LP.Buff);
end LPDriver.
```

The **using** clause specifies the particular communication scheme to be used for the implementation of the ports *and* each port to which they are to be connected (two ports can be connected only if their message types match, their control rules are compatible and if they use the same communication scheme). We have assumed here that the line printer is represented as a module LP which uses the communication scheme InterruptLevel, and that other modules communicate with each other using the scheme InterProcess. When there is no **using** clause, it is assumed that some standard default communication scheme will be used. "Using" a communication scheme means that the operations of that port will be provided by the module named in the **using** clause.

There are several occasions where the use of different communication schemes would be obvious:

- We have already mentioned the handling of devices: in general, the interface with the hardware of a machine (including all forms of device operations and interrupts) would require a scheme which is quite distinct from all others;

- In a system with multiple processors and memories, one scheme could be used for modules which can share memory for their execution (in this case, messages can often be passed without making copies in buffers), and another scheme would be necessary for modules which execute on different processor-memory systems; for a distributed system, providing one scheme for local communication and another for inter-node communication would allow more efficient message transfer.

- Some systems have a hierarchical configuration: for example, on the Cm*[13] system, processors are grouped in clusters which are then interconnected; this would suggest the use of one scheme for modules executing on the same processor, another for modules on different processors in the same cluster, and a third for modules in different clusters.

It should be noted that the statements in the implementation of a module do not need to be altered when there is a change of communication scheme; all that is required is to recompile a module whenever a different scheme is to be used. The choice of scheme is therefore static but so also, in most cases, is the architecture.

We have emphasised the use of different communication schemes to take account of the features of the architecture: this is a fairly straightforward requirement (though not one considered in most languages). But even on the same architecture, it is clear that several different schemes may be necessary if the same language is to be used for the implementation of the operating system and user programs. This is because the implementation of the mechanisms for synchronisation and access to storage have usually to differ between the operating system and user programs. In the past, this requirement has either been hidden, or different compilers have been used for operating system and user programming, or low level languages have been used so that synchronisation has been provided by explicit programming.

9 Exception Handling

In a programming language designed with particular regard for communication, an exception should obviously be treated as another form of communication: between a module detecting an exception and one which has to take some action for recovery. But there are special requirements for adequately handling exceptions, and limitations to what can be accomplished in a distributed system. We will restrict ourselves to dealing with exceptions which invalidate other communications, or which require the execution of a module to be terminated; we will not consider simple exceptions which can be characterised as distinguished values in the range of a function (such as a function returning the value zero for illegal arguments), as these can obviously be handled by normal programming techniques. To simplify the discussion, we will often talk of exceptions which can occur when a module A sends a message to a module B and is awaiting its response.

1. Let us describe exceptions which occur during communication between A and B as *communication exceptions*; such exceptions can have several causes:

   - The message sent by A to B may be outside the domain of the function to be evaluated by B (as in the case, eg, of a simple square root function which only takes positive real numbers as arguments). An exception is then send to indicate this fact (and that the value which will be returned by the function is undefined). In this case, A must be prepared to receive two different messages.

   - Module B may fail to respond to A's request, either due to software or hardware errors on a processor; for example, this may be detected by a timeout at the **iport** at which A is waiting for the reply.

   - There may be a failure of the communication medium between A and B, also detected by a timeout at A's **iport**.

   In the last two cases, there can be no communication from B itself and A could be waiting indefinitely unless some alternative action is taken.

2. There is another important set of cases where exceptions may be detected: this is where a module fails to perform correctly and has to be terminated. We will refer to such exceptions as *module exceptions*; they are distinguished by the fact that recovery is not possible (eg, due to stack overflow) and, exception handling, when it is possible at all, is limited to 'cleaning-up'operations.

Handling communication exceptions requires the detecting module or scheme to send an exception message through a pre-defined **oport**, and the receiving module to declare and connect an **iport** to this **oport**. This is analogous to providing a handler for an exception condition. The action that can be taken by this mechanism can be quite elaborate:

1. The announcement of the exception may merely be informative, and it may be entirely up to the programmer of a module to decide what action, if any, he wishes to take; this can be done by having the exception **oport** follow the control rule '-' .

2. Alternatively, the exception **oport** may follow the control rule '->', so that it can only be connected to an **iport** which will send a reply; this will allow a receiving module to do some recovery, and for the exception module to examine the effects of this action before allowing the module to continue.

3. The exception **oport** may also send a message to a monitoring module, and allow the module with the error only limited time for recovery.

Module exceptions are more difficult to handle as the module with the error may not be waiting for any message (and therefore will not receive an exception message even if it is sent). One way of overcoming this problem is to use 'watch-dog' timers. Another is for the scheme implementing the creation and deletion of modules (see the next section) to handle such errors.

Whenever exceptions can be handled as messages received by an exception **iport**, there is the advantage

that all the variables of the module are available for inspection and modification (ie, the exception 'handler' executes in the same scope as the other instructions of the module).

10 Implementing Modules

Having illustrated how ports can be built as modules, allowing a variety of schemes to be used for the implementation of the communication primitives, it is interesting to consider if the same technique can be applied to the implementation of *modules*. The reason for this is that we have so far assumed that all modules are created and terminated in identical manner, and that module names are unique over an entire system. Both these assumptions are questionable: in a complex system, there is likely to be need for several different implementations of modules in the same way that there is need for different implementations of ports. In a hierarchically composed system, for example, there may be one implementation of modules for the operating system, another for users, and a user may have need for yet another implementation for the individual modules of his program; each level of the system may provide different implementations of modules in order to control the creation, scheduling, and termination of modules at the next level. Such a hierarchy may also be aptly representative of multi-level communication protocols where several levels may exist between communication at the level of a user program and the eventual transfer of data across a physical medium.

We can envisage the implementation of modules being done by operations in a pre-defined module:

```
module specs NewModules;
    ...
    provides iport CreateModule ( ) -> (ModuleName); DeleteModule ( ) -> (ModuleName;
            IsaModule (ModuleName) -> (boolean);
end NewModules.
```

and for the specifications of a module to (perhaps optionally) indicate which implementation scheme is to be used:

```
module specs UserModule using NewModules;
    ...
end UserModule.
```

Apart from the practical advantages of this technique, there is also the fact that it allows a common specificational technique to be used for studying the properties of modules schemes, communication schemes, and 'user' modules which use these schemes. The use of algebraic specifications for a simple communication data type has already been described by Cunha and Maibaum[14] and a similar but more elaborate definition for these schemes is being considered separately.

11 Conclusions

Software designed for the distributed environment has often been built around 'service' programs such as file servers, printer servers, etc. In such designs, the extent of parallelism is determined by the extent to which the active agents in the system (ie, the processes) are able to manipulate the passive objects (eg, files) in parallel. In constrast to this, the language we have described here is intended to be used in a way which exposes the maximum parallelism *in the problem*: it is then possible to reduce this to the extent necessary for adaptation to the available hardware, using appropriately designed communication and module *implementation schemes*. The distinction between these two approaches is important and likely to become even more so as distributed and parallel systems of hundreds of processors become widely available.

The low cost of small, powerful processors and fast memory, and the developments in communication technology are likely to make it possible to build interconnected computer systems of many kinds, eg those with shared memory, with a common bus, or special purpose systems for applications like process control. It would be unrealistic either to expect each such system to be programmed in a special language or for the important characteristics of the system to be hidden in the special implementation of a general language. One purpose of using communication schemes is to be able to separate these two concerns: to be able to implement the logic of a module independently of the choice of communication scheme to be used.

Languages like CSP and DP have already demonstrated the attractive possibilities of using parallel programming features as the basis for language design. Here, we extend parallelism to include 'broadcast' mode output and multiple caller input, also allowing the particular synchronisation discipline to be specified as a control rule. This is done in the context of a fairly realistic design which can readily be implemented: in fact, the feasibility of implementation has been an important consideration in determining how ambitious the language should be, and has been one reason for making the features relatively simple.

These simplifications have not resulted in any significant compromise either in the power of the language or in the compactness with which algorithms can be expressed. On the other hand, they have led to a very simple and efficient rule for the lifetimes and termination of modules, which assumes great importance in an actual implementation. A conspicuous omission from this report has been any mention of a program verification or proof methodology which would serve as an aid for the design and validation of programs. This issue is being considered separately and the relatively simple nature of the language appears to make this task tractable.

12 Acknowledgements

This report describes some of the major language issues in the design of communication schemes. I would like to thank Michael McKeag for his interest in an earlier version of the language, and several members of the Computer Science Department at Carnegie-Mellon University for comments which can only have improved the paper. Richard Snodgrass made a number of detailed and perceptive suggestions on earlier versions of this report which have undoubtedly made the presentation more precise and understandable, and I am grateful to Nico Habermann for discussions on some of the features described here. None of them is responsible for any errors which remain.

The basic idea behind communication schemes originated from the experience gained while implementing and using the CCNPascal language[15] but its present form has been considerably influenced by the work on CSP[3] and DP[4].

References

1. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, June 1975, pp. 199-207.

2. N. Wirth, "Modula: a Language for Modular Multiprogramming," *Softw. Pract. & Exp.*, Vol. 7, No. 1, January 1977, pp. 37-66.

3. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

4. P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Comm. ACM*, Vol. 21, No. 11, November 1978, pp. 934-941.

5. J.A. Feldman, "High Level Programming for Distributed Computing," *Comm. ACM*, Vol. 22, No. 6, June 1979, pp. 353-368.

6. N. Wirth, "The Programming Language Pascal," *Acta Informatica*, Vol. 1, No. 1, 1971, pp. 35-63.

7. E. W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *Comm. ACM*, Vol. 18, No. 8, August 1975, pp. 453-457.

8. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

9. J.L. Bentley, H.T. Kung, "Two Papers on a Tree-Structured Parallel Computer," Tech. report, Carnegie-Mellon University, September 1979.

10. J.G. Mitchell, B. Wegbreit, *Schemes: A High-level Data Structuring Concept*, Prentice-Hall, 1978, pp. 150-184.

11. J.H. Ichbiah et al, "Rationale for the Design of the Ada Programming Language," *SIGPLAN Notices*, Vol. 14, No. 6, June 1979, , Part B

12. B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, "Abstraction Mechanisms in CLU," *Comm. ACM*, Vol. 20, No. 8, August 1977, pp. 564-576.

13. R.J. Swan, *The Switching Structure and Addressing Architecture of an Extensible Multiprocessor : Cm**, PhD dissertation, Carnegie-Mellon University, 1978.

14. P.R.F. Cunha,T.S.E. Maibaum, "A Communication Data Type for Message Oriented Programming," *IV Intl. Symp. on Prog.*, Springer-Verlag, 1980, pp. 79-91, Lecture Notes in Computer Science.

15. M. Joseph, V.R. Prasad, K.T. Narayana, I.V. Ramakrishna, S. Desai, "Language and Structure in an Operating System," in *Operating Systems - Theory and Practice*, D. Lanciaux, ed., North Holland, Amsterdam, 1979.

I. Appendix

I.1 Dining Philosophers

Problem: (due to E.W. Dijkstra) Five philosphers spend their time between pondering, going to a room to eat, and returning to think. The food at the table is spaghetti, and this requires two forks for eating. There are five forks placed on a round table, between the philosophers' places. Each philosopher must pick up the fork on his left, the fork on his right, and then eat. Write a program which will allow the philosophers to eat without deadlock (ie, all five philosophers taking one fork each and waiting for a neighbour to release the other fork).

Solution: The program given by Hoare[3] uses one process for the room, each philosopher, and each fork, and we will give a similar solution.

```
module type Room;
    provides iport In ( ) => ( ); Out ( ) -> ( );
end Room.

module implementation Room;
    declare Count : integer ( := 0);
        In :: receive ( ) Count := Count + 1; reply ( );
        Out :: receive ( ) Count := Count - 1; reply ( );
    begin
        loop
            if Count < 4 then await {In, Out} else await {Out} end if;
        end loop
    end Room.

module type Fork;
    provides iport Pick ( ) => ( ); Down ( );
end Fork;

module implementation Fork;
    declare Pick :: receive ( ) reply ( );
        Down :: receive ( );
    begin
        loop Pick; Down end loop;
    end Fork.

module type Sage;
    provides oport Enter, Take1, Take2 ( ) -> ( );
        iport Leave, Drop1, Drop2 ( );
end Sage.
```

```
module implementation Sage:
    declare Enter, Take1, Take2 :: send ( ) response ( );
        Drop1, Drop2, Leave :: send ( );
    begin
        loop Ponder; Enter; Take1; Take2;
            Consume; Drop1; Drop2; Leave;
        end loop
    end Sage.

module specs Dinner;
    needs type Room, Fork, Sage;
    provides module Rm : Room; Fk : [1 .. 5] of Fork;
        Phil : [1 .. 5] of Sage;
    connect (Rm.In, Phil.Enter[1]..[5]); (Rm.Out, Phil.Leave[1]..[5]);
        (Fk.Pick[1..5], Phil.Take1[1..5], Phil.Take2[2..(<2)]);
        (Fk.Down[1..5], Phil.Drop1[1..5], Phil.Drop2[2..(<2)]);
end Dinner.

%Note: < n  = pred n mod array size\

    module implementation Dinner;
        declare Rm : Room; Fk : [1 .. 5] of Fork; Phil : [1 .. 5] of Sage;
        end Dinner.
```

Synchronisation is essential for two actions, entering the room and picking up a fork, and this is ensured by using the " => " control rule for the associated ports. Dropping a fork and leaving the room are both actions which can be performed asynchronously and should require no waiting. Deadlock is prevented by not letting more than 4 philosophers into the room at a time. Note that each philosopher's ports Take1 and Take2 are connected to the appropriate forks according to their position at the table.

Problem: A file may be read simultaneously by a number of readers but may only be written by one writer at a time, and then only if there are no readers (this is the Readers and Writers problem). Assume the file has two operations, Read and Write, to access and modify the file respectively..

Solution (cf Brinch Hansen[4]): Let State be an integer variable recording the kind of access being made to the file. Each file will perform its own control over access: the operations Read and Write must be preceded by the operations StartRead and StartWrite, and followed by the operations EndRead and EndWrite, respectively.

```
module specs RWFile;
    provides iport StartRead ( ) => ( ); Read ( ) => ( ... ); EndRead ( ) -> ( );
        StartWrite ( ) => ( ); Write ( ... ) -> ( ); EndWrite ( ) -> ( );
end RWFile;
```

```
module implementation RWFile;
    declare State : integer (:= 1)
        Data : ... ;
        StartRead :: receive ( )
                State := State + 1
            reply ( );
        Read :: receive ( )
                < Get Data >
            reply ( ... );
        EndRead :: receive ( )
                State := State - 1
            reply ( );
        StartWrite :: receive ( )
                State := 0;
            reply ( );
        Write :: receive ( ... )
                <Update Data>
            reply ( );
        EndWrite :: receive ( )
                State := 1
            reply ( );
    begin
        loop
            if State = 1 then await {StartRead, StartWrite}
                elseif State > 1 then await {StartRead, Read, EndRead}
                elseif State = 0 then await {Write, EndWrite}
        end loop
    end RWFile.
```

A writer can start only if there are no readers or writers already (State = 1); a reader can start if there is no active writer (State >= 1).

Problem: (cf. Brinch Hansen[4]) A vending machine has two operations: Insert, which takes in a coin, and Serve, which returns an item if there are any left and the inserted coins cover the cost (if either of these conditions is false, any inserted money is returned).

```
module specs VendingMachine;
    provides iport Insert (integer) -> ( ); Serve ( ) => (integer, integer);
end VendingMachine.

module implementation VendingMachine;
    declare Stock, Paid, Cash, Change : integer (:= 50, 0, 0, 0);;
        Insert :: receive (Coin : integer)
                Paid := Paid + Coin;
            reply ( );
```

```
    Serve :: receive ( )
            if Stock > 0 and Paid >= Price then
                Change, Item := Paid - Cash, 1;
                Stock, Cash := Stock - 1; Cash + Price
            else Change, Paid, Item := Paid, 0, 0 ;
        reply (Change, Item);
begin
    loop
        await {Insert, Serve}
    end loop
end VendingMachine.
```