CONIC: an integrated approach to distributed computer control systems

J. Kramer, B.Sc., (Eng.), Ph.D., M.B.C.S., J. Magee, M.Sc., C. Eng., M.I.E.E., M. Sloman, B.Sc., (Eng.), Ph.D., and A. Lister, M.A., Dip. Comp. Sc., M.B.C.S.

Indexing terms: *Computer applications, Process control*

Abstract: Distributed computer control systems (DCCS) have a number of potential advantages over centralised systems, especially where the application is itself physically distributed. A computer station can be placed close to the plant being controlled, and a communications network used to enable the stations to communicate to co-ordinate their actions. However, the software must be carefully designed to exploit the potential advantages of distribution. In the paper, the CONIC architecture for DCCS is described, concentrating on the software structure but also briefly describing the physical architecture designed to support a CONIC system. The software structure emphasises the distinction between the writing of individual software components and the construction and configuration of a system from a set of components. A modular structure is used to separate programming from configuration. Typed entry and exit ports clearly define a module interface which, like the plugs and sockets of hardware components, permit modules to be interconnected in different ways. On-line modification and extension of the system is supported by permitting the dynamic creation and interconnection of modules. Message-passing primitives are provided to permit modules to co-ordinate and synchronise control actions.

1 Introduction

The impact of large-scale integration and microprocessor technology has been well publicised in the computing world. The availability of cheap microcomputers has led to their increasing use in process-control applications. Microcomputers can be placed physically close to the plant being controlled and can also be used to replace hardwired control, giving added flexibility, in addition to cost reduction. Centralised computer control systems can be replaced by systems composed of distributed microcomputers.

However, new hardware and software architectures are needed to fully exploit the potential of these distributed computer systems. The CONIC project provides a unified and integrated approach to the design, implementation and management of large distributed computer control systems (DCCS). It consists of: a **network architecture**, which permits the interconnection of large numbers of computer stations; a **software methodology**, which supports the design of application systems as a set of interconnected re-usable components; a **distributed operating system** and **communication system**, which provide runtime support for application software; and a **management system**, which enables a system to be tailored from a set of components and subsequently modified and extended to meet changing application needs.

This paper concentrates on the CONIC software structure but briefly describes the proposed hardware environment for a typical industrial implementation. In particular, the paper deals with two aspects of DCCS software: the writing of application programs which perform subfunctions of the overall control task, and the construction and configuration of a complete system from a set of such application programs [1].

Section 2 of the paper gives an overview of a typical DCCS environment. We also describe our approach to the design of software for a DCCS, which emphasises the separation of the programming of software components from the configuration of such components into a complete system. Section 3 gives details of the software components (modules) and their interfaces, and describes how modules are interconnected. The next Section describes the internal structure of a module and the message-passing primitives which are available to the module programmer. Finally, we give an outline of the hardware and software environment which supports the CONIC architecture and some information on the current implementation on five LSI 11 microcomputers. A simple example is used throughout the paper to illustrate the concepts. The paper does not deal with the 'lower' operating system or communication system layers [2] of a DCCS, except where they impinge directly on the areas of consideration. However, some desirable properties of these lower layers can be inferred from the proposals made in this paper.

The work on which this paper is based is part of a research project funded by the UK National Coal Board, and so coal mining is used as a focus for requirements and examples [3]. However, the proposals are equally valid for many other process-control applications.

2 Distributed computer control systems

A DCCS is one where the control function has been partitioned into subfunctions which can be implemented by a set of physically distributed computer stations. Typically, a subfunction will be concerned with the local control of an item of plant or a machine. In order to provide a fast response time, a computer station which implements such a subfunction will be located in close physical proximity to the machine it controls.

Thus, the physical distribution of stations in a DCCS will, usually, closely correspond to the geographical layout of the system being controlled. In our coal-mining application, this means that computers may be separated by distances of up to 20 km. However, in other process-control applications, distance of up to 1 or 2 km are more usual.

To enable the overall control function to be performed, stations co-ordinate and synchronise their actions by exchanging messages via a communication network. Because of the distances between stations, this network must be implemented using serial data-transmission techniques. The communication network will always have a residual probability of failure and will introduce delays into communication between control system components. Consequently, the components must cope with communication delays and failures.

The potential advantages of distributed systems over centralised systems have been well documented elsewhere [4].

Paper 2183 E, received 11th May 1982

Dr. Kramer, Mr. Magee and Dr. Sloman are with the Department of Computing, Imperial College of Science & Technology, 180 Queensgate, London SW7 2B2, England. Mr. Lister is with the University of Queensland, St. Lucia, Queensland, Australia 4067

The advantages for process control can be summarised as:

(*a*) improved response time by locating computers close to plant

(*b*) increased availability, since the effect of physical faults such as processor failure are confined to one station

(*c*) ease of extension and modification by adding/taking away stations and communication links

(*d*) increased performance by exploiting parallelism.

The above advantages do not accrue automatically from the use of a distributed hardware architecture. DCCS software must be carefully designed to exploit the potential advantages of distributed hardware, while dealing with the problems caused by interstation communication delays and failures. It is the issue of DCCS software that this paper primarily addresses.

2.1 The CONIC approach to DCCS software

A major objective of our software architecture has been to separate the concerns of writing individual software components (programming-in-the-small) from those of constructing or configuring a system from a set of components (programming-in-the-large) [5]. This separation allows components to be programmed without knowledge of the configuration in which they will be used. Consequently, a set of standard components (e.g. 3-term controllers) could be provided and used in many different system configurations. This strict separation of programming and system building also allows a configuration to be modified without recompilation of its constituent components. In the following, we outline the characteristics of CONIC software components and identify the requirements for configuration of a control system from these components.

Software components

A software component is executed at a station and implements some subfunction of the overall control function. Components can communicate solely by exchanging messages. Consequently, the interface to a component can be defined solely in terms of the messages it can transmit and receive. This characteristic is exploited by our CONIC architecture to provide a set of message ports as a component interface. This facilitates the separation of concerns identified above.

We have rejected the possibility of dynamic migration of operational components between stations. This facility has been provided in some distributed systems to enhance performance by load sharing and to enable software components to tolerate hardware failures. However, in DCCS, as outlined before, stations are typically located with the sensors and actuators they serve. Migration of the software function makes little sense when the hardware function cannot be moved. Fault tolerance may be achieved by replication of components, or by reconfiguration after failures. The CONIC system does not automatically save the state of components and this would have to be recreated by reading hardware or by explicit checkpoints programmed into the application software.

However, although software components are not in themselves fault tolerant, they must be able to deal with communication failures and failures in other components of the system. The loose coupling of stations by serial data links in a DCCS considerably enhances the system hardware robustness, since the effect of physical faults is usually confined to one station. It is important that the software of DCCS mirrors this characteristic so that failure of one component does not cause complete system failure. This error confinement objective is similar to the 'fault isolation' objective of the HXDP executive [6]. Components interact only by message passing, so it is primarily the message-passing mechanism that is affected by this requirement. The message-passing primitives are described in Section 4 of the paper.

Configuration

Configuration is the construction of a control system from a set of components. It involves the assignment of these components to the computer stations on which they will run and their interconnection for communication.

On-line modification:

In most computer control applications, the configuration is not fixed indefinitely at installation time but must change to deal with failures and to meet changing applications needs. In the coal-mining application, the configuration must be modified and extended as new coal faces are developed and old ones become worked out. Additionally, new control functions must be added to deal with new types of machines. It is not practical to stop the entire control system to modify a small part of it, since the system performs some critical safety functions. Consequently, there is a requirement for *on-line* modification and extension to the control system in a safe and controlled way. This requirement for on-line modification is not unique to coal mining but applies to any application where the control system must be continuously available. CONIC provides for on-line creation, deletion and interconnection of software components.

Interconnection:

In computer control, components may be interconnected to form many different structures (e.g. pipelines, hierarchies, feedback loops). However, all these structures can be provided by the following three interconnection patterns provided within our architecture:

(*a*) *One-to-one* – between two specific components, e.g. a command from a controller to an actuator. The connected path may be two way to permit a response to the command

(*b*) *One-to-many* – a component may broadcast a message to multiple destination components, e.g. sensor readings

(*c*) *Any-to-one* – one at a time access by many components to a server component, e.g. an error logger. The paths connected may be two way to permit reply or response messages.

Section 3 describes how the above configuration and interconnection requirements are provided by our software architecture.

2.2 Example: control of pump for mine drainage

The diagram in Fig. 1 is the schematic of a very simplified pump installation. It is used to pump mine water, collected in a sump at shaft bottom, to the surface. The control software for this system will be developed in the rest of the paper.

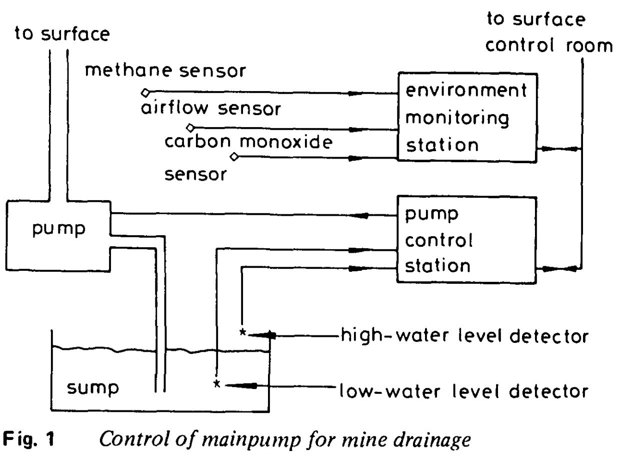


Fig. 1 *Control of mainpump for mine drainage*

Once start has been enabled by a command from the surface, the pump runs automatically, controlled by the water level as sensed by the high- and low-level detectors. Detection of high level causes the pump to run until low level is indicated.

The pump is situated underground in a coal mine, and so for safety reasons it must not be started or continue running when the percentage of methane (firedamp) in the atmosphere exceeds a set safety limit. The pump controller obtains information on methane level by communicating with a nearby environmental monitoring station. As well as methane, this station monitors carbon monoxide level and airflow velocity. The environment monitoring station provides information to the surface and other plant controllers as well as to the pump controller.

3 System configuration

System configuration is the concern of the system designer or manager. It deals with the software components which are installed on hardware stations, their interfaces and their logical interconnection. Our approach facilitates both validation of module interconnections and on-line modification and extension of an operational system.

The concepts are introduced and illustrated by extracts from the pump example.

3 1 Software components: modules

The software architecture which we adopt mirrors the underlying structure of microprocessor stations interconnected by a communications network. We propose the **module** as the software abstraction of a station. The module provides the link between programming activities and those of system configuration. A module type definition is the largest entity constructed by the applications programmer, typically to perform some local monitoring and control function (e.g. a pump controller). On the other hand, module instances form the basic building blocks from which the system is configured. The module is thus the smallest software component which can be distributed or replaced.

In order to allow multiple instances of similar modules to exist in the system, we permit programmers to define parameterised module types. These are the units of compilation. Individual modules are merely instances of a particular module type. They are created and assigned to stations at configuration time. A single module cannot be partitioned among several stations. However, in order to allow efficient use of a single station without discouraging modularity, we permit more than one module instance to be assigned to (i.e. share) a station.

For example, module types 'pumpcontroller' and 'environmonit' could be defined by a programmer for pump control and monitoring the methane, carbon monoxide and airflow, respectively. Module instances of each type – say,

pumpl: pumpcontroller

and

envl: environmonit

could then be created at particular stations as part of a complete system.

3.2 Module interface: ports

In order to co-operate and synchronise their actions, modules communicate by message passing. The interface which a module presents to the rest of the system is given by the messages which can be sent and received by the module. In our system we characterise this interface by sets of **exit** and **entry ports**, respectively [7].

An entryport may be thought of as a 'hole' through which messages can pass into the module, e.g.

RECEIVE pcommand FROM cmd

where 'cmd' is an entryport. Similarly, an exitport can be thought of as a 'hole' through which messages are sent out of a module. Thus a message is directed, not to an entryport of a receiving module, but to an exitport of the sending module; e.g.

SEND pumpcommand TO out

where 'out is an exitport. In the case of a 'request-reply' transaction (see Section 4.2), the reply is received from the same exitport used for the request, e.g.

SEND pumpcommand TO out WAIT response

and, similarly, the reply message is directed at the same entryport used for the request, e.g.

```
RECEIVE pcommand FROM cmd;
........
REPLY pstate TO cmd
```

Both entry and exit ports can thus be used for bidirectional information flow but a transaction is always initiated at an exitport.

This approach permits the programmer to design and implement module types without specifying (in the code) the name of the sending/receiving module or its exit/entry port. From the point of view of the programmer, entry and exit ports are local names for arbitrary message source(s) and destination(s), respectively. The delayed association between the exitport of one module and the entryport of another can be regarded as a deferred binding of a message to its destination from compile time (when it may not be known) to the system configuration stage (when it is known). This facilitates modification and extension of a system by reconfiguration [8]. It also allows a system to include standard or 'library' modules.

For example, Fig. 2 gives an outline of the interfaces for the 'pumpcontroller' and 'environmonit' module types.

3.3 Module interconnection: port linkage

The discussion above has resulted in the notion of a module with a set of entryports and a set of exitports. A message from one module to another is sent to an exitport of the sender and is received on the entryport of the receiver.

The association between exit and entry ports, called **linking**, provides a communication path which can be established during system installation and re-established as necessary during system modification. External to a module, a particular port is addressed as 'modulename.portname' where portname is unique within a module, and modulename is unique in the system, e.g.

LINK surface.out TO pump1.cmd

where surface and pumpl are module instances (Fig. 3).

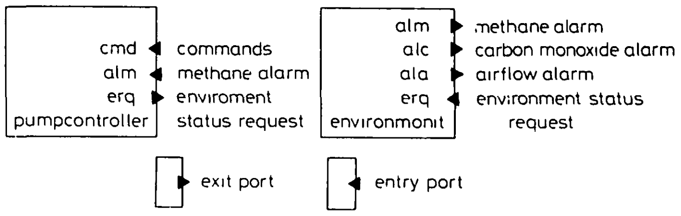


Fig. 2 *Pump control example : entryports and exitports*

surface
out
pump1
cmd

Fig. 3 *Port linkage*

Port linkage provides an explicit module interconnection structure for a system. An exitport may be linked to one or more entryports to provide a single or multi-destination communication path respectively. This satisfies the one-to-one and one-to-many interconnection structures described in the preceding text (Section 2.1). Many exitports can be linked to the same entryport to provide the any-to-one structure required of servers. Communication and the primitives provided are further discussed in Section 4.

3.4 Message and port types

Within a module, messages are processed as data objects in the same way as the simple and structured data (such as arrays, strings, or records) of conventional programming languages. Therefore messages are also considered as instances of particular data types.

In order to communicate, both the sender and receiver must be aware of the structure and representation of the message. We need to guarantee that the type of message a module expects to receive is the same as that which was sent. Such a guarantee is realised by associating a type with a port in the same way as with messages, and by allowing a linkage to be effected only if entry and exit ports are of the same type. We therefore insist that all port declarations include a type specification, e.g.

```
ENTRYPORT cmd: command
```

where, say, command is defined by enumeration as

```
command = (start, stop, status).
```

A message may therefore be sent and received only via ports of the same type as the message. This scheme gives a high degree of security with no runtime overhead. Within a module, the type consistency of messages and ports can be checked at compile time, and between module ports at linkage time. Type mismatches at run time can therefore occur only as a result of hardware corruption within a station or on a transmission line. This will be detected by error-detection mechanisms within the station or within the communication system.

Since the same port types need to be available to those modules which may potentially be linked, files of common type definitions must be defined. Those definitions needed by a particular module can then be **included** in the module definition. This approach avoids the problems of checking multiple type definitions made in different modules.

The modularity and flexibility achieved in software, by the notions of a module whose interface is a set of typed entry and exit ports, is similar to but more secure than that achieved in hardware by components whose interfaces are plugs and sockets.

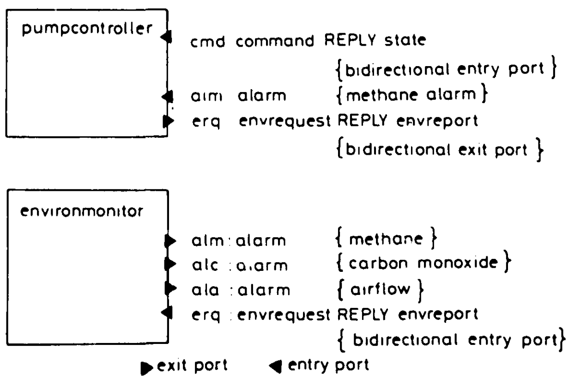

pumpcontroller
cmd command REPLY state
{bidirectional entry port}
alm alarm {methane alarm}
erq envrequest REPLY envreport
{bidirectional exit port}
environmonitor
alm : alarm {methane}
alc : alarm {carbon monoxide}
ala : alarm {airflow}
erq : envrequest REPLY envreport
{bidirectional entry port}
exit port
entry port

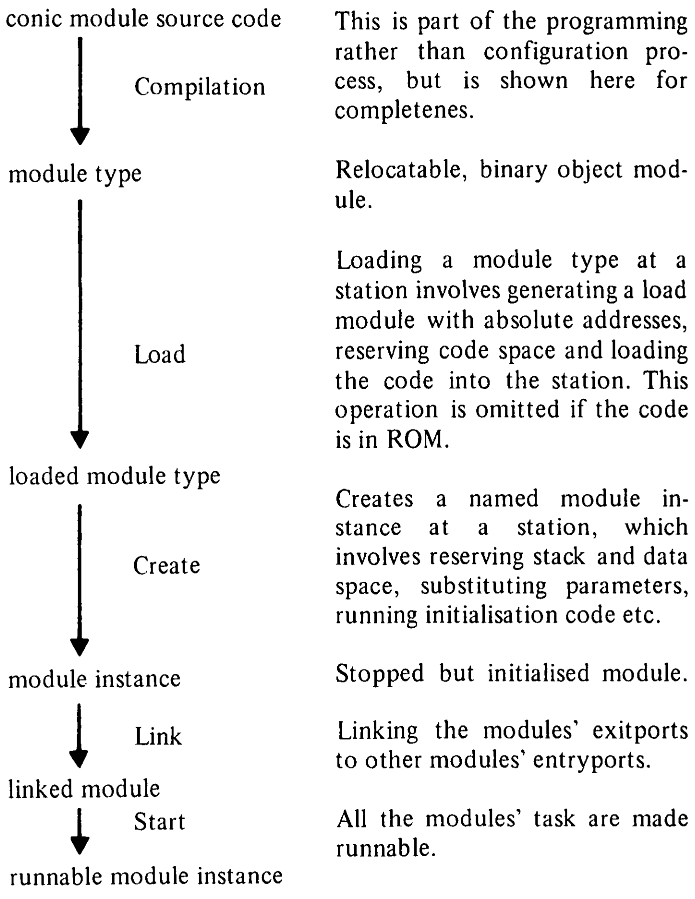
Fig. 4 *Pump control example : module interfaces*

Returning to our pump example, the module ports for the pump controller and the environment monitor are shown diagrammatically in Fig. 4. The type definitions for the ports are stored in a file of definitions called 'pumpdefns':

```
{ FILE pumpdefns; }
TYPE command  = (start, stop, status);
     state    = (running, ready, stopped, lowstop,
                 methanestop);
     sensor   = (methane, carbmonox, airflow);
     envreport = RECORD reading : REAL;
                           sn : sensor
                 END;
     envrequest= sensor;
     alarm    = signaltype;
{ ENDFILE}
```

3.5 Configuration management

Configuration management includes all the activities related to installing, removing or modifying the hardware or software components of the DCCS in order to extend or change the system or to reconfigure as a result of failures. Configuring a distributed control system from predefined module types involves the stages of loading a module type into a station, module instance creation, port linkage, and module initiation as shown in Fig. 5.



This is part of the programming rather than configuration process, but is shown here for completenes.

Relocatable, binary object module.

Loading a module type at a station involves generating a load module with absolute addresses, reserving code space and loading the code into the station. This operation is omitted if the code is in ROM.

Creates a named module instance at a station, which involves reserving stack and data space, substituting parameters, running initialisation code etc.

Stopped but initialised module.

Linking the modules' exitports to other modules' entryports.

All the modules' task are made runnable.

Fig. 5 *Stages in installing a module at a station*

The first stage is the **loading** of named module types at particular stations. This is the mapping of the logical system of modules onto the physical system of stations. The module code must be downline loaded if it is not already held locally at the station (e.g. in ROM):
e.g.

```
LOAD pumpcontroller AT station1
```

Named instances of module types are then created at the stations:
e.g.

```
  CREATE pump1: pumpcontroller AT station1
```

As described earlier, module interconnection is performed by **linking** together module exit ports to entry ports. Type checking is performed to ensure compatibility. Linking completes the mapping of the logical interconnection structure onto the physical network of stations. Finally, the system (or parts thereof) can be initiated by **starting** named modules. In case of failure, modification or extension of the system, we allow all configuration stages (including **stopping, unlinking** and **deleting** of modules) to be carried out while the rest of the system is operational.

Configuration management will make use of a database which includes type definitions, logical and physical configurations, and provides name to address translation. The type definitions of modules will be maintained for use by the loader. Message and port types are used by the compiler (for module type definition) and the linker (for checking port links). The logical configuration information will provide an up-to-date picture of the module instances and the links between exit and entry ports. The physical configuration will provide descriptions and status information of stations and subnetworks (see Section 5). Together, they provide a logical and physical picture of the current system and a means of logical name to physical address translation.

3.6 Configuration of pump control system

```
{Module creation from predefined types. Note: a device
address parameter is passed to the module pump1}
   CREATE pump1: pumpcontroller (#177562)
                             AT station2;
   CREATE surface:operatormod AT station1;
   CREATE envl:environmonitor AT station3;
{Port linkage with type validation}
   LINK surface.out TO pump1.cmd;
                             {one-one connection}
   LINK pump 1.erq TO env1.erq;
                             {one of any-one connection}
   LINK env1.alm TO pump1.alm, surface.alm;
                             {one-many connection}
{Initiation of this configuration}
   START pump1, surface, env1;
```

The relevant part of the configuration is shown diagrammatically in Fig. 6. Note that a configuration may be modified using the available configuration commands: start/stop, link/unlink, create/ delete.

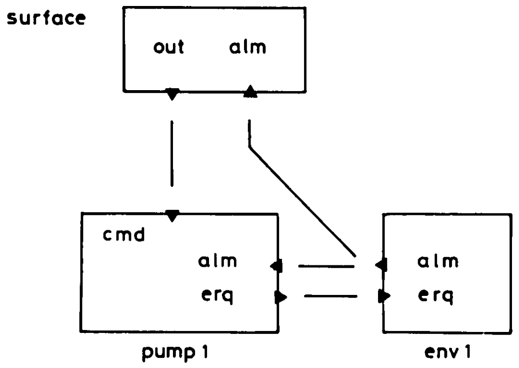


Fig. 6 *Pump control example : linked exit and entry ports*

4 Application programming

The previous Section identified the 'module' as the unit of application programming. This Section describes the internal structure of a module and the message-passing primitives which are available to the module programmer.

4.1 Module structure

A module-type definition consists of the following elements:

```
  MODULE ⟨identifier⟩ (⟨parameters⟩)
      INCLUDE ⟨file identifier list⟩
      ⟨entry and exit port declarations⟩
      ⟨type definitions⟩
      ⟨variable declarations⟩
      ⟨procedure declarations⟩
      ⟨task declarations⟩
      ⟨link statements⟩
  BEGIN
     ⟨initialisation statements⟩
  END.
```

Module parameters can be used to tailor a standard module type for a specific use when an instance is created, e.g. to specify a device address.

Ports
The exit and entry port declaration part specifies the module interface as described in the preceding Section. The type definitions for these interface ports are declared in external definitions files. The contents of these definitions files are made available for use inside the module by the INCLUDE statement.

Types, variables and procedures
Those declared at module level have global scope and can be shared by the tasks within a module but are not visible outside the module boundary. Consequently, even when installed in the same station, modules cannot share data or procedures. Tasks within a module can, however, synchronise access to shared data by message passing as described below.

Tasks
A module may include one or more tasks. Tasks are the active entities within a module. They correspond to the 'processes' of Concurrent Pascal [9] and Modula-1 [10]. Tasks are instantiated when a module is created at a station, initiated when the module is started and terminate when the module is deleted. Consequently, the number of tasks within a module is defined at compile time. We have chosen this static model of tasking, as opposed to the more dynamic model incorporated in Ada [12], primarily for simplicity, and also because flexibility to create and destroy software components is already provided by modules at the system configuration level. A task contains the following elements:

```
  TASK ⟨identifier⟩
          ⟨entry and exit port declarations⟩
          ⟨local type, variable, procedure declarations⟩
  BEGIN
          ⟨statements⟩
  END
```

The task-level port declarations specify the exit and entry ports which are owned by a task. Those ports which are visible outside the module boundary must also be declared at module level. In this way, task ports are 'exported' to form the module interface. Ports which are declared at task level but do not appear at module level are used for intertask communication within a module. These 'internal' ports may be

Table 1: Relationship between port declarations, connection patterns and message primitives

Port declarations	Connections	Message primitives
EXITPORT pump:command REPLY state	one-to-one	SEND start TO pump WAIT response;
ENTRYPORT cmd:command REPLY state	one-to-one any-to-one	RECEIVE cmdmsg FROM cmd REPLY pumpstate;
EXITPORT alm:alarm	one-to-one one-to-many	SEND signal TO alm;
ENTRYPORT palm:alarm	one-to-one any-to-one	RECEIVE almmsg FROM palm;

of types which are defined at module level. They are connected by LINK statements within a module. Thus, a task uses the same message-passing mechanisms to communicate both with tasks in the same module and with tasks in other modules.

Tasks within a module may also communicate by shared access to the data declared at module level. Since a module instance can never be split over multiple stations, shared data can legitimately be used to improve efficiency. This avoids the overheads of message copying involved in message passing, and is especially useful for large tables, buffers etc. Synchronised access (e.g. for mutual exclusion) to this data can be accomplished using message communication. The message primitives available to a task are outlined in the following text.

4.2 *Message primitives*

Our system provides two sets of primitives for inter-task communication by message passing. The reasons for choosing these primitives and a detailed description has been included in a previous paper [12]; hence the treatment below is brief. The same primitives are used for both local and remote communication to allow modules to be located in the same or a different station. Table 1 indicates the port declaration and permissable interconnection patterns for each message primitive.

Send-wait & receive-reply

This set of primitives provides the 'command-response' and 'query-status' message transactions commonly found in computer control systems. The transaction consists of two messages: a request from a source task to a destination task which asks the destination to perform some service and, subsequently, a reply message from destination to source containing service completion information. This message exchange is treated as a single transaction from source exitport to the destination entryport. The execution of the source task is suspended between sending the request message and receiving the reply message. An example of the send primitive is shown below:

```
SEND start TO pump
   WAIT response     =⟩ –– successful send action
   TIMEOUT period    =⟩ –– timeout action
   FAIL              =⟩ –– unlinked exitport action
END
```

where start is a value of type 'command' and response a variable of type 'state'. The exitport 'pump' would be declared as:

```
EXITPORT pump : command REPLY state;
```

The optional TIMEOUT clause in the send-wait primitive allows a source task to limit the time it is suspended waiting for a reply message. It prevents the source task from being suspended indefinitely, following a communication failure or destination task failure. The send-wait is similar to the Ada timed entry call, except that, in Ada, the timeout is on the acceptance of the call by the destination task (i.e. receipt of the request message), whereas our timeout is on completion of the call at the source task (i.e. receipt of the reply message). We believe our timeout mechanism leads to simpler implementation and more robust task behaviour in distributed systems.

The optional FAIL clause allows the source task to detect that its exitport is not linked to an entryport (due to a configuration change or a communication failure). The task could perhaps choose to delay and retry periodically, or to send an error message to some error logger. If no fail clause is included the system will report the error and abort the task.

If neither timeout nor fail clauses are used, a simplified form of send-wait may be used:

```
SEND start TO pump WAIT response
```

The receive-reply primitive causes the destination task to be suspended until a message is available at the designated entryport. The destination task may then perfom some processing on the message before replying. In order to associate the reply with the message received, the reply must specify the entryport on which the original message was received. e.g.

```
RECEIVEcmdmsg FROM cmd;
      –– process message
REPLY pumpstate TO cmd
```

In those cases where no processing is necessary, the receive-reply primitive can be used in a simplified form:

```
RECEIVE cmdmsg FROM cmd REPLY pumpstate
```

In both cases, the entryport 'cmd' would be declared as:

```
ENTRYPORT cmd:command REPLY state;
```

An exitport with a REPLY part may be linked to only one destination 'REPLY-type' entryport. There is no requirement for multiple responses to a request, and also the semantics for multidestination request-reply connections in the face of failures tend to be very complex. In order to provide the any-to-one or server interconnection pattern described in Section 2, one or more 'REPLY-type' exitports may be linked to an entryport with REPLY part. Messages sent to an entryport from more than one source are responded to in FIFO order as they are received.

As shown below, the receive-reply primitive may be incorporated in an Ada-like select statement to enable a task to wait on messages from a number of potential sources. An optional guard can precede each receive in order to further define the conditions upon which messages should be received. If the guard is false then the corresponding receive is not available for selection.

```
SELECT
    WHEN command guard
      RECEIVE cmdmsg FROM cmd
                                =>  .........
                                    .........
                                    REPLY pumpstate TO cmd
                                    .........
  OR
    WHEN device guard
      RECEIVE waterlevel FROM level REPLY signal
                                =>  .........
                                    .........
  OR
      RECEIVE .............
  ELSE TIMEOUT Period
                                =>  .........
                                    .........
   END;
```

As a simple example of the use of receive-reply primitives, a bounded buffer module is given below:

```
MODULE boundedbuffer;
    ENTRYPORT putchar : char REPLY signaltype;
              getchar : signaltype REPLY char;
                        {signaltype is a null reply}
    CONST maxsize = 100;

  TASK buffer;
    ENTRYPORT putchar : char REPLY signaltype;
              getchar : signaltype REPLY char;
    VAR inp, outp, contents : integer;
       buffer : ARRAY [1 . . maxsize] OF char;
    BEGIN
      inp : = 1; outp : = 0; contents : = 0;
      LOOP
        SELECT
          WHEN (contents<maxsize)  {buffer not full}
          RECEIVE buffer [inp] FROM putchar
          REPLY signal =>
            inp : = (inp MOD maxsize) + 1;
            contents : = contents + 1;
        OR
          WHEN (contents>0)         {buffer not empty}
          RECEIVE signal FROM getchar
            REPLY buffer [outp] =>
            outp : = (outp MOD maxsize) + 1;
            contents : = contents-1;
        END
      END
    END;
  BEGIN    END.
```

Send & receive
The second set of communication primitives provide a unidirectional, potentially multidestination message passing service which meets the requirement for the transfer of alarm and status information. The send operation is asynchronous in that it does not cause execution of the sending (source) task to be suspended and so can be used by tasks performing time-critical functions. Consequently, unlike the send-wait operation, a source task may execute many send operations before the destination task executes a receive operation.

The send operation would seem to imply the need for multiple-message buffering, whereas the send-wait operation requires, at most, one buffer, as only one message can be outstanding at a time. However, to avoid the complexity of dynamic buffer management and the problems invoked when no buffers are available, we have chosen to statically allocate a fixed, dimensionable queue of buffers for each receive entryport, e.g.

```
ENTRYPORT in : char QUEUE 80;
```

When no more buffers are available, the oldest message in the queue is overwritten. This strategy is simpler than causing an exception or trying to block the sender, both of which are difficult in a distributed system. The default allocation is a single buffer. This buffer, the 'unibuffer' [12], is updated by successive send operations and emptied by receive operations. If the source task sends messages faster than they are being received by the destination task, the unibuffer will be overwritten (e.g. updates to an operator display). We believe this general overwrite strategy is reasonable in a computer control environment in order to provide the most up-to-date or latest message(s), while still allowing for a known and controlled discard of messages at times of burst-message traffic. If it is important that information is not lost, either the request-reply primitives should be used, or a higher level protocol can be constructed using the buffering primitives (e.g. file transfer protocols). An example of the asynchronous send operation is shown below:

```
SEND status TO out
```

where status is a value of type 'statustype' and the exitport 'out' would be declared as:

```
EXITPORT out : statustype;
```

Exitports with no reply part may be linked to one or more entryports to give the one-to-many interconnection pattern of Section 2.2. A message sent to a multidestination exitport is transferred to all the entryports to which it is linked. The receive primitive is syntactically the same as the receive-reply primitive, except that the REPLY part is omitted.

The CONIC program for the pumpcontroller module type is given in Appendix 9.

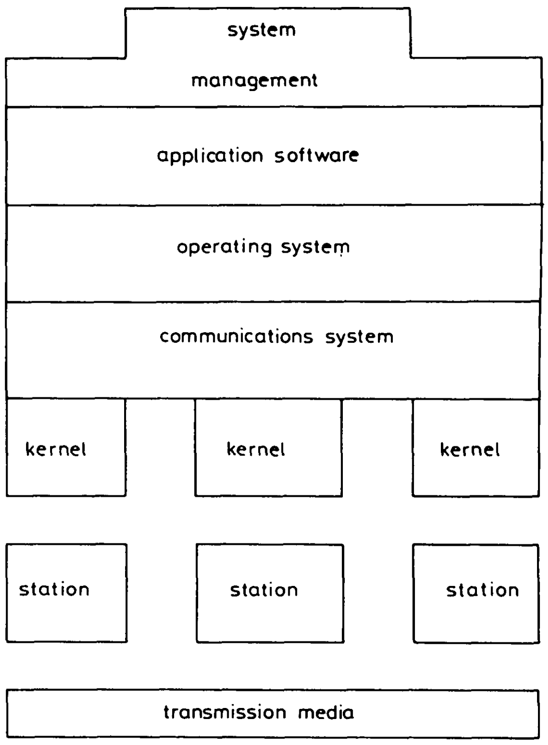


Fig. 7 *CONIC system architecture*

5 The CONIC environment

This Section gives a brief description of the hardware and software entities which provide the support environment for installing, modifying and running CONIC application programs. Fig. 7 identifies the entities which constitute a CONIC system.

5.1 *Hardware*

Stations

A computer station will typically consist of two closely coupled processors – one performing application functions and the other performing communication functions. The reasons for this separation are that the functions are independent and both may be time critical. The CONIC architecture also allows for a single processor station for simpler functions. In all cases the station must implement the standard interface to the communication system.

Network topology

A control system is usually hierarchical, in that the overall system is functionally or geographicaly structured into subsystems which may themselves consist of complex machines with independent controllers. This structure is reflected in the physical configuration of a CONIC system. The overall network consists of subnetworks which can correspond to a subsystem, a machine or a geographical area of the plant. The subnetworks are broadcast and are interconnected by store and forward gateways. The architecture is independent of the subnet technology and can use whatever suitable VLSI circuits emerge for the office automation or local network market, e.g. ring, ETHERNET, IEEE serial highway [13–15]. Fig. 8 shows a simple network of interconnected rings and serial highways. The overall network architecture consists of a mesh of subnets to allow multiple paths between any two subnets [16].

5.2 *Kernel*

Each station has a kernel which implements multi-tasking and intertask communication, within a station, via message passing. In addition, it provides the mechanisms required by the station operating system to manage station resources and support dynamic module creation and linking of ports.

5.3 *Communication system*

The communication system supports interstation communication and provides two main services. The first is a reliable connection-based service for transferring variable length messages between stations. This service performs automatic correction of transmission errors and reroutes around failures. The application programs using this service need not concern themselves with retransmissions etc. The second service is a simple datagram type of service with lower overheads but no guarantee of delivery. This is mainly used for system management but is also available for application use.

The communication system routes messages from one subnet to another using tables. These tables are built up and maintained using a completely distributed algorithm which automatically recovers from failures. The system also supports multidestination addressing (i.e. a single message can be sent to multiple modules) and broadcasting (to all stations on a subnet).

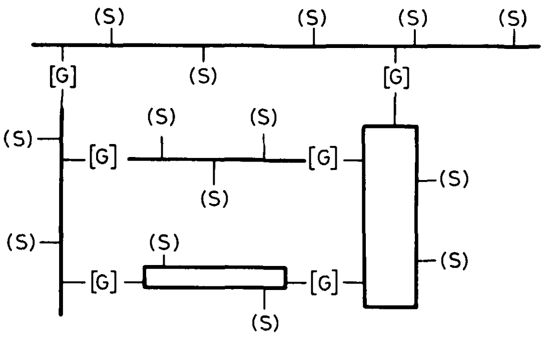


Fig. 8 *Interconnected subnets*
[G], gateway; (S), station

5.4 *Operating system*

The station operating system provides a set of services which support the following functions:

(*a*) down-line loading of module code into station store
(*b*) creation/deletion of module instances
(*c*) start/stop of module execution
(*d*) linking/unlinking of ports resident in the station
(*e*) detection/reporting of module program errors and station hardware failures.

The operating system is implemented as a set of CONIC modules and its services are made available via entryports. Operating system services can thus be invoked by either local or remote users through message passing. The operating system modules in each station co-operate by message passing to form a distributed, network-wide operating system.

5.5 *System management*

The management system for CONIC will provide facilities for system installation and management, including the installation of hardware and software, changing the configuration of the components of the system, controlling their status and initiating maintenance and diagnostic operations. Facilities for gathering information such as performance statistics, error reports and status reports will also be provided. The management system will maintain an up-to-date picture of the actual system in a database which will allow selective retrieval of the information. Finally, based on information gathered, the management system will provide decision making facilities such as reconfiguration in the face of failures.

In CONIC, the same mechanisms may be used to manage the application system, operating system and communication system, as all are constructed using the same techniques (modules with interconnected ports). However, different user interfaces to the management system will be required, e.g. for the system designer who builds and configures the initial system, for the site engineer who performs routine modifications, and for the maintenance engineer who examines fault information and runs diagnostics.

6 Conclusion

CONIC provides an integrated set of tools for distributed computer control systems. In particular, the modular software concepts for program construction and system building have been found to be especially powerful and versatile. Modules separate the programming of a functional unit from its configuration with other units to form a control system. The use of typed exit and entry ports to clearly define a module interface promotes the reuse of modules in many different systems and permits the reconfiguration of a single system without recompilation of its constituent units. The ability to dynamically create and interconnect modules allows on-line modification and extension of a system.

CONIC will be used to construct the application control and monitoring software as well as other application facilities, such as operator interface, diagnostics, logging etc. Furthermore, we believe that the real-time facilities required for programming and building application software does not differ significantly from those for writing the system software (operating system, communications, utilities etc.). Hence, wherever possible, we have used the CONIC concepts for implementation of the system software. In fact, even when a system is not distributed, we have found that its design as a set of CONIC modules facilitates testing and system integration.

The module concept and message primitives have been added to the programming language Pascal. This 'extended Pascal' is translated by a preprocessor into standard Pascal. The extra functionality of tasking and message communication is incorporated as procedure calls to the kernel.

We have implemented the kernel, communications system and operating system described in Section 5 on a system composed of 5 LSI-11 computers. The LSI-11s are connected by asynchronous links to form a communications ring. In the future, the asynchronous links will be replaced by a suitable local area network, such as a Cambridge Ring [13] or Ethernet [14].

The kernel has been implemented in Pascal with a small number of assembly code inserts. It requires 3 K bytes of store for its code and data. The time for a request-reply cycle (sending a request message to a task in the same station which immediately replies) on the LSI 11/02 is given by:

$$2.46 + 0.013 * (Rb + Pb)\,\text{ms}$$

where Rb is the number of bytes in the request message and Pb is the number of bytes in the reply message (e.g. 4.25 ms for 10 byte request and 128 byte reply messages).

The communications system does not as yet handle multiple subnets or provide virtual circuit connections. It supports a datagram service on a single subnet. It is implemented as a set of CONIC modules. Its store requirement is 7 K bytes for code, data and buffer space. The processing time required by the communication system to get a message from an application module and transmit it out of the station is about 7 ms. To receive an incoming message and pass it to an application module takes about 10 ms.

The station operating system has also been implemented as a set of CONIC modules. It requires 7 K bytes of store.

The store size and runtime of both the kernel and communications system could be optimised by hand coding them in assembler. However, in the prototype system, which is subject to change, this would be wasted effort.

A number of utilities have been implemented in CONIC to run on the system. These include a terminal driver, error logger, a file server and an interactive debugging aid. Since the interface to a module is via message passing, the debugging aid can be used on local or remote modules. Additionally, a distributed conveyor control simulation has been written in CONIC and runs on the system.

We are currently implementing the management system, which enables an operator to interactively configure and subsequently modify the control system he requires. The management function is being implemented as a set of modules which run on the above system.

7 Acknowledgment

Acknowledgment is made to the National Coal Board for a grant in aid of these studies, but the views expressed are those of the authors and not necessarily those of the Board.

8 References

1 LISTER, A., MAGEE, J., SLOMAN, M., and KRAMER, J.: 'Distributed process control systems: programming and configuration'. Imperial College, Research Report DOC 80/12, May 1980

2 PRINCE, S., and SLOMAN, M.: 'Communication requirements of a distributed computer control system', *IEE Proc. E, Comput. & Digital Tech.*, 1981, **128**, (1), pp. 21–34

3 SLOMAN, S., KRAMER, J., MAGEE, J., and SAADAT, S.: 'Present and future coal mining application requirements for distributed computer control systems'. Imperial College, Research Report DOC 80/15, July 1980

4 FARBER, G.: 'Principles and applications of decentralised process control computer systems'. IFAC 1978, Helsinki, Vol. 1 pp. 385–392

5 DEREMER, F., and KRON, H.: 'Programming-in-the-large versus programming-in-the-small'. Proceedings of conference on reliable software, 1975, pp. 114–121

6 BOEBERT, W., FRANTA, W., JENSEN, E., and KAIN, R.: 'Decentralised executive control in distributed computer systems'. Proceedings of int. computer software and applications conference, Nov. 1978, pp. 254–258

7 KRAMER, J., and CUNNINGHAM, R.: 'Towards a notation for the functional design of distributed processing systems'. Proceedings of int. conference on parallel processing, Aug. 1978, pp. 69–76

8 KAIN, R., and FRANTA, W.: 'Interprocess communication schemes supporting system reconfiguration'. Proceedings of computer software and applications conference, Oct. 1980, pp. 365–371

9 BRINCH-HANSEN, P.: 'The architecture of concurrent programs' (Prentice-Hall, 1977)

10 WIRTH, N.: 'Modula: a language for modular multiprogramming', *Software–Pract. & Exper.*, 1977, 7, pp. 3–35

11 USA DEPARTMENT OF DEFENCE: 'Reference manual for the ADA programming language: proposed standard document', July 1980

12 KRAMER, J., MAGEE, J., and SLOMAN, M.: 'Intertask communication primitives for distributed computer control systems'. Proceedings of 2nd int. conference on distributed computing systems, Paris, Apr. 1981, pp. 404–411

13 WILKES, M., and WHEELER, D.: 'The Cambridge digital communication ring'. Proceedings of the local area communications network symposium, Boston, May 1979, pp. 47–61

14 IEEE: 'IEEE 802 local network standard: draft B', Oct. 1981

15 XEROX CORPORATION: 'The Ethernet: a local area network – data link and physical layer specifications'. Version 1.0, Sept. 1980

16 SLOMAN, M., and PRINCE, S.: 'Local network architecture for process control'. *In* WEST, A., and JANSON, P. (Eds.), 'Local networks for computer communications' (North-Holland), pp. 407–428

9 Appendix: Control of pump for mine dranage

Programming: the pumpcontroller

For the sake of brevity, only the pumpcontroller is described in detail. Its internal structure is shown in Fig. 9.

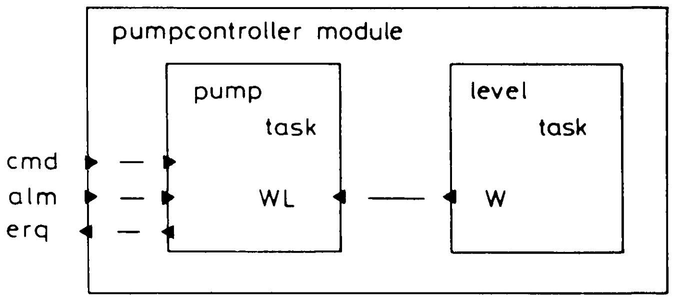


Fig. 9 *Internal structure of pumpcontroller*

The pumpcontroller consists of the included type definitions, pumpdefns, and the two tasks, pump and level. Level is a simple task which periodically scans the water level sensors and sends the level reading to the pump task via an internal link from port W to WL. Task pump performs the required pump control. A program for the pumpcontroller is given below:

```
MODULE pumpcontroller (sensoraddr:integer);
      {* Module Interface *}
   INCLUDE 'pumpdefns';                    {see Section 3.4 for definition}
   ENTRYPORT cmd : command REPLY state;
                 alm : alarm QUEUE 1;
   EXITPORT   erq : envrequest REPLY envreport;

      {* Module Body *}

   TYPE waterlevel = (lowlevel, highlevel, normallevel);

TASK level;
        EXITPORT W : waterlevel;
           CONST period = 10;
           VAR wlevel : waterlevel;
        BEGIN
           LOOP
           --  scan waterlevel sensors at module parameter address sensoraddr
               and put result in wlevel.
                SEND wlevel TO W;
                DELAY period;
           END;
        END;

 TASK   pump;
        ENTRYPORT cmd : command REPLY state;
                    alm : alarm QUEUE 1;       {Note: QUEUE 1 can be omitted as}
                    WL : waterlevel QUEUE 1;        {it is the default.}
        EXITPORT erq : envrequest REPLY envreport;

        CONST  safetylimit = 1.25;
                  period = 100;
        VAR pstate : state; palarm : alarm;
              plevel : waterlevel; pcommand : command;

         FUNCTION startcheck : state;             {* request and check methane level * }
              VAR request : envrequest;
                    report : envreport;
              BEGIN
                    request : = methane;
                    SEND request TO erq
                       WAIT   report =) IF report.reading(safetylimit
                                                THEN  startcheck : = ready
                                                ELSE   startcheck : = methanestop;
                       TIMEOUT period=)ERROR ("Environment request failure");
                                              startcheck : = methanestop
                       FAIL              =)ERROR ("Environment link failure");
                                              startcheck : = methanestop
                             {* ERROR is a call to a standard system error logger * }
                       END

              END;
```

```
  BEGIN
      pstate : = stopped;
      LOOP
         SELECT                                    {* process a command *}
            RECEIVE pcommand FROM cmd =)
                        CASE pcommand OF
                           stop : BEGIN
                              IF pstate = running THEN -- stoppump;
                              pstate : = stopped;
                              END;
                           start : IF pstate () running THEN pstate : = ready;
                         status : {null};
                        END;
                        REPLY pstate TO cmd
        OR                                         {* process an alarm * }
           RECEIVE palarm FROM alm =)
                        IF pstate = running THEN --stoppump;
                       •pstate : = methanestop;
        OR                                         { * process waterlevel * }
           RECEIVE plevel FROM WL =)
                      CASE plevel OF
                            highlevel : IF (pstate = ready) OR (pstate = lowstop)
                                        THEN BEGIN
                                     pstate : = startcheck;
                                     IF pstate = ready
                                              THEN BEGIN -- startpump;
                                                        pstate : = running;
                                                    END
                                     END;
                            lowlevel : IF pstate = running
                                           THEN BEGIN -- stoppump;
                                                      pstate : = lowstop
                                                END;
                            normallevel : {null};
                      END;
        END;
      END;
  END;

LINK W TO WL; { * Internal links within the module * }

BEGIN
END.
```

Jeff Kramer graduated from the University of Natal, South Africa, with a degree in electrical engineering, in 1970. He was awarded an M.Sc. in 1972, and Ph.D in 1979, both degrees in computer science from Imperial College, London. After working as a programmer and a research assistant, he became a lecturer in the Department of Computing at Imperial College in 1976. His research interests include specification and design of distributed systems, and tools for the production of verified software. Together with Dr. Sloman, he is principal investigator of the CONIC project funded by the National Coal Board. Dr. Kramer is a founder member of EWICS TC 11 on Application Oriented Specifications.

Morris Sloman graduated from the University of Cape Town with a degree in electrical engineering in 1970 and later obtained a Ph.D. in computer science at the University of Essex. After working for GEC Computers Ltd., he joined the Department of Computing at Imperial College as a lecturer in 1976. His research interests include distributed computer architecture, computer communications and operating systems. Together with Dr. Kramer, he is principal investigator of the CONIC Project funded by the National Coal Board. Dr. Sloman is co-ordinator of a COST 11 funded project on Local Area Networks involving ten research institutions in Europe, and is a member of EWICS TC 5 Group on Communications.

Jeff Magee graduated from Queens. University, Belfast with a degree in electrical engineering in 1973, and was awarded an M.Sc. in computing science from Imperial College in 1978. He spent six years with the British Post Office working on the development of System X. Since 1979, he has been a Research Assistant at Imperial College, working on distributed systems for real-time applications. Mr. Magee is a member of the EWICS TC 8 group on Real-time Operating Systems.

Andrew Lister graduated from Cambridge University, and afterwards lectured in computer science at the Universities of Lancaster and Essex before moving to the University of Queensland, where he is currently head of the Department of Computer Science. His research interests lie in operating systems and distributed computing, particularly in the area of interprocess communication. He is author of a textbook on operating systems and co-author of an introductory text on computer science.