4º SBRC
RECIFE - 24 A 26 DE MARÇO 86

THE CONIC TOOLKIT FOR BUILDING DISTRIBUTED SYSTEMS *

M. Sloman, J. Kramer, J. Magee.

Department of Computing, Imperial College, London, England, SW7 2BZ.

Abstract. CONIC provides a set of tools for building flexible distributed systems for embedded applications such as factory automation, telecommunications, process monitoring and control. The CONIC programming language is used to program individual software modules which communicate by naming only local entryports and exitports. This gives configuration independence and allows reuse of the modules in various situations. A separate configuration language is used to specify a system by creating instances of modules and interconnecting exit and entryports. The configuration language is also used to specify changes which can be performed dynamically without shutting down the complete system. These features of a CONIC system provide the flexibility for adapting to changing requirements. This paper describes the CONIC programming and configuration languages as well as the run-time support needed for dynamic configuration. The paper also gives an overview of the Unix based tools available for building and testing software for distributed target computers. Finally we discuss experiences of using these tools and future work planned on the project.

Keywords. Distributed systems; programming languages; configuration management; dynamic change; program development environments.

INTRODUCTION

Large embedded systems are expected to have a long lifetime. They do not remain static during their operational life, but evolve as application requirements change and as new technology is incorporated. In fact the introduction of the computer system itself tends to act as a stimulus for change in the application environment, and so the services provided by the system must evolve. For example in the process control industry, computers may be introduced gradually, starting with stand alone controllers, then plant-wide automatic monitoring and eventually evolving to an integrated distributed computer control system.

In addition to **evolutionary change**, distributed systems must cater for **operational changes**. Components may have to be physically relocated to cater for modifications to the plant being controlled. After failures of parts of the system, continued, possibly degraded, operation should be possible by manual or automatic reorganisation. Distributed embedded systems should also cater for redimensioning: extension by addition of existing components or removal of superfluous ones.

The CONIC approach to building distributed systems provides the **flexibility** for the system to evolve and change to meet changing requirements and conditions. A CONIC distributed system can easily incorporate new functionality in response to evolutionary changes and allows reorganisation of existing components in response to operational changes. CONIC supports dynamic reconfiguration so that it is not necessary to shut down a complete system in order to make changes.

It has been widely recognised that in order to build large software systems, it is necessary to decompose the system into components which can be separately programmed, compiled and tested. The system is then constructed as a configuration of these software components. The separate activities of component programming and system building (configuration) have been referred to as "programming in the small" and "programming in the large" respectively. In CONIC this is reflected in separate component programming and configuration languages.

CONIC MODULE PROGRAMMING LANGUAGE

Task Modules

Modularity is a key property for providing flexibility. The Conic programming language is based on Pascal, with extensions for modularity and message passing [Kramer 84].

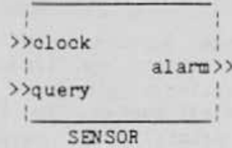
The language allows the definition of a **task module type** which is a self-contained, sequential task (process). A task module type is written and compiled independently from the particular configuration in which it will run ie. it provides configuration independence in that all references are to local objects and there is no direct naming of other modules or communication entities. This means there is no configuration information embedded in the programming language and so no recompilation is needed for configuration changes as is the case with other languages such as CSP [Hoare 78] and ADA [DOD 80].

At configuration time, **module instances** are created from module types. Module instances exchange messages and perform a particular function in the system such as controlling a device or managing a resource. Multiple instances of a module type can be created on the same or different stations in a distributed system and a station can contain many different modules instances. A major objective of CONIC is to allow the production of reuseable modules [Wegner 84]. Multiple instances of the same module may be used within a particular system; the same module type can be reused in different versions of a system and a module type may be a

* Presented at 6th IFAC Distributed Computer Control System Workshop, Monterey California, USA. May 1985.

standard component which is reused in different applications.

Fig. 1 is an example of a task module which reads a sensor whenever a signal is received from a clock. If the sensor reading is greater than a preset limit the module sends an alarm. It can also receive messages querying the current sensor reading.




```
task module sensor (limit:integer)
   type valtype = record
                    value:integer;
                    state:boolean;
                  end;
   exitport alarm : valtype;
   entryport clock :signaltype;
            query : signaltype reply valtype;
   var reading : valtype;
   function readsensor():integer
        {performs I/O to read sensor}
   end;
begin
   reading.value := 0;
   reading.state := false;
   loop
     select
        receive signal from clock
        => with reading do
             begin value := readsensor();
             if value > limit then begin
                 state := true;
                 send reading to alarm;
             end;
          end;
     or
        receive signal from query reply reading;
     end;
   end;
end.
```

Fig. 1 Example Task Module

CONIC modules have a **well defined interface** which specifies all the information required to use the module in a system. The interconnections and information exchanged by modules is specified in terms of **ports**. An **exitport** denotes the interface at which message transactions can be initiated and specifies a local name and message type in place of the destination name. In fig. 1, alarm messages are sent to the task's 'alarm' exitport. At configuration or run time, the exitport can be linked to a compatible entryport (ie. of type 'valtype') of any task which wishes to receive alarm messages. The 'sensor' task's **entryports** 'clock' and 'query' in fig. 1, denote the interface at which message transactions can be received. At configuration or run time, any task with a compatible exitport can be linked to these entryports. The programming language uses local names within the task instead of directly naming the source and destination of messages. The binding of an exitport to an entryport is part of the configuration language and cannot be performed within the programming language. There is thus no need to recompile a task module when it is reused in different situations. This provides complete configuration independence for a task module.

At creation time, instantiation parameters can be passed to a module to tailor a module type for a particular environment, for example the alarm limit value passed to the sensor task in fig. 1, or the device address passed to a device driver.

There are two classes of ports which correspond to the message transaction classes described below. **Request-reply Ports**, such as 'query' in fig. 1 are bidirectional. They specify both a request and reply message type. **Notify Ports** such as 'clock' and 'alarm' are unidirectional ie. they have no reply part. For convenience, it is possible to define families (arrays) of identical ports such as in the nurse module in fig. 7.

Ports define all the information required to use a module and so it is very simple to replace a module with a new or different version with the same operational interface.

Communication Primitives

Communication primitives are provided to **send** a message to an exitport or **receive** one from an entryport. The message types must correspond to the port types. The primitives provide the same syntax and semantics for local (within a station) and remote (inter-station) communication. Differences in performance between local and remote communication are inevitable due to network delays. This **Communication Transparency** allows modules to be allocated either to the same or different stations, which can be particularly useful during the development of embedded systems in that modules can be fully tested together in a large computer with support facilities and then later distributed into target stations.

There are two classes of message transactions:

a) A **Notify transaction** provides unidirectional, potentially multi-destination message passing. The send operation is asynchronous and does not block the sender, although the receiver may block waiting for a message. There is a (dimensionable) fixed size queue of messages associated with each entryport. Messages are held in order of arrival at the entryport. When no more buffers are available the oldest message in the queue is overwritten. The Notify can be used for time critical tasks such as within the communication system, with the queue size corresponding to a flow-control window.

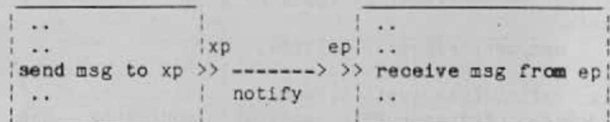


Fig. 2 The Notify Transaction

b) A **Request Reply** transaction provides bidirectional synchronous message passing. The sender is blocked until the reply is received from the receiver. A fail clause allows the sender to withdraw from the transaction on expiry of a timeout ('tval' in fig. 3) or if the transaction fails. The receiver may block waiting for a request. On receipt of a request, the receiver may perform some processing and return a reply message. In place of a normal reply, the receiver may either **forward** the request to another receiver (thereby allowing third party replies) or it may **abort** the transaction.

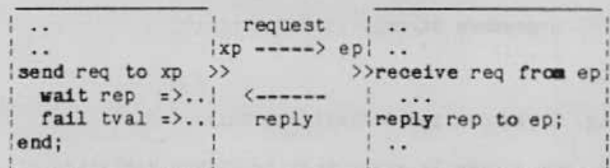


Fig. 3 Request-Reply Transaction

Standard functions are provided to determine whether an exitport is linked to an entryport, the number of messages queued at an entryport or the reason for a send-wait failing.

Any of the receive, receive-reply, receive-forward, or receive-abort primitives can be combined in a **select** statement (fig. 4). This enables a task to wait on messages from any number of potential entryports. An optional guard can precede each receive to further define conditions upon which messages should be received. A timeout can be used to limit the time spent waiting in the select statement.

```
      select
ep1       when G1
 >>       receive req1 from ep1 reply signal
      or
ep2       when G2
 >>       receive req2 from ep2
             => ....                               xp1
                forward ep2 to xp1                 >>
ep3   or
 >>       receive msg3 from ep3 => ......
      or
          when Gn timeout tval
          => {timeout action}
      end;
```

Fig. 4 Selective Receive

<u>Definitions Unit</u>

The module is the basic reuseable software component within a system. However there are many definitions which are common between different modules within a system. Definitions of constants, types, functions and procedures may be defined in separate definitions units. These can be compiled independently and can be imported into a module to define a **context**. This avoids errors introduced by having to redefine message types in communicating modules. For example the definition of message type 'valtype' in fig. 1 would, in practice, be imported from a definitions unit called 'sensortypes' by means of a declaration such as:

```
     use sensortypes : valtype;
```

The definitions unit allows the introduction of language "extensions" without modifying the compiler. For example a set of standard string definitions and manipulation procedures can be made available as a definitions unit as shown in fig. 5. This exports 2 functions 'strlength' and 'strcpy', and a type 'string', but prevents the representation of 'string' being visible outside the definitions unit.

```
define stringdefs: strlength, strcopy
                   opaque string;

     const strmax = 128;
     type string = record
                   len:integer;
                   ch :array[1..strmax] of char;
                  end;
     function strlength (s:string):integer;
      .......
     procedure strcopy (s1,s2:string);
      ......
end.
```

Fig. 5 An Outline Definitions Unit

We are currently experimenting with a variation of the definitions unit which includes data and initialisation code. This is similar to an Abstract Data Type but only a single instance can be declared when it is imported into a task module. However multiple instances of the encapsulating task module can be declared. The encapsulating task can access the data via exported procedures or directly (if the data variables are exported) but other modules must access the data via the encapsulating task's message passing interface.

<u>Input-output</u>

The programming language supports the standard Pascal I/O procedures, which are automatically transformed by the compiler into message passing operations on standard exitports. In addition, CONIC provides simple primitives to support the programming of device handlers as application tasks. Fig. 6 shows a transmitter driver for a synchronous communications line. It makes use of a set of special kernel calls imported from 'kercalls' definitions unit. The task raises its priority to 'system' to ensure it is not preempted by any other task while transmitting a message. The 'waitio' procedure suspends the task until an interrupt occurs on the vector specified as a parameter. When an interrupt occurs the scheduler is not called but rather the hardware effectively schedules the relevant device driver via the interrupt vector. Different device drivers may have different hardware priority levels, allowing nested interrupts. This is similar to the facilities provided in Modula [Wirth 77]. It is simpler and more efficient than the conversion of interrupts to messages within the underlying kernel as in ADA [DoD80] or SR [Andrews 82].

```
task module transmit (status,vector : address);

   use commstypes : msgtype;
       kercalls : priority, {system, normal etc.}
                  setpriority, waitio;
   entryport tx : msgtype reply signaltype;
   var txstat : ^set of 0..15;
       txbuff : ^char;
       msg : msgtype;

begin
   ref (txstat,status);    {converts address to}
   ref (txbuff,status+2); {pointer type}
   loop
      receive msg from tx;
         setpriority (systempr);
         for i := 1 to msg.length
         begin
            txbuff^ := msg.chars[i];
            txstat^ := [4];          {enable device}
            waitio (vector);
            txstat^ := [ ];          {disable}
         end;
       reply signal to tx;
      end;
      setpriority (normal);
   end
end.
```

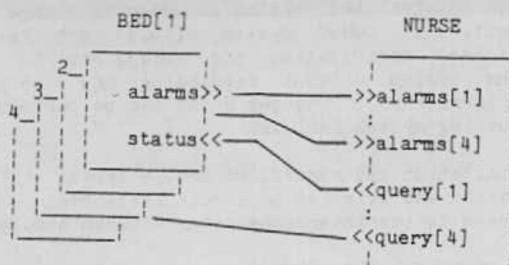
Fig. 6 Device Driver Task Module

CONIC CONFIGURATION LANGUAGE

One of the key elements in the provision of flexibility is the need to separate the programming of individual software components (task module types) from the building of a system from instances of modules. This has led to the development of the CONIC Configuration Language [Dulay 84] which is used to specify the instances of module types, the interconnection of module instances and the mapping of a logical configuration onto physical stations. The same language can be used to specify both an initial system and subsequent changes to the system.

An Example Configuration Specification

The following example describes a patient monitoring system consisting of four bed modules and a nurse module. Alarms from beds are displayed at the nurse module and the nurse can query the beds to obtain current sensor readings.




```
system ward;
   use     bedmonitor, nurseunit;
   const   nbed  = 4;
           scanrate = 100;
   create family k:[1..nbed]
           bed[k]: bedmonitor(scanrate) at node (k);
   create nurse : nurseunit at node (5);
   link    family k:[1..nbed]
           bed[k].alarms to nurse.alarms[k];
           nurse.query[k] to bed[k].status;
end.
```

Fig. 7 Ward Monitoring System

In the above example, the **use** construct specifies a context by identifying the set of module types from which the system will be constructed. The named module instances within the system are declared by the **create** construct. The actual values of the module parameters are provided at creation time. The create construct can also declare a **family** (array) of module instances of a particular type as shown for bed. The optional **at node** part of the create construct defines the station address at which the module is to be created. Logical to physical mapping will be further discussed in later.

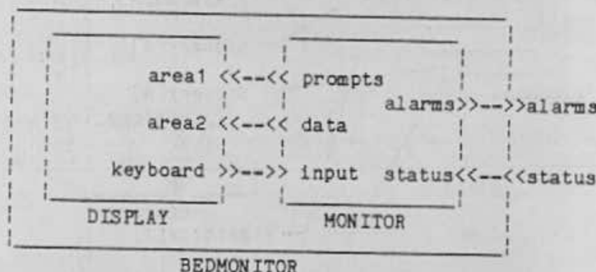
The **link** construct specifies the interconnection of module instances by binding a module exitport to a module entryport. Both type and operation compatibility are checked so an exitport can only be linked to an entryport of the same data and transaction type. Multiple exitports can be linked to a single entryport which is particularly useful for connecting clients to servers (eg. a file server). A single exitport can be linked to multiple entryports which provides multidestination message transactions. Multidestination is generally used for notify transactions, but can apply to request-reply transactions. The first reply from a multidestination request is accepted and all others are discarded. This can be used with replicated components for reliability purposes.

Structuring Configuration Specifications

The modules in a distributed system often exhibit a hierarchical relationship. For example a database subsystem makes use of file servers, which may themselves consist of directory servers, record access servers and disc drivers. This structure can be represented by nesting software components at the configuration level by means of **group modules**. A group module type is a configuration specification and identifies a collection of module types, instances of those types and their interconnection. The constituent modules may be the primitive task modules containing a single process or group modules. The group modules also have an interface defined in terms of exit- and entryports, as well as formal parameters. This structuring of the specification is essential for large systems with many module instances, otherwise the name space would become unmanageable and the configuration specification unreadable.

The structure of a group module is defined by the **use, create,** and **link** constructs described previously. The interface to the module is also defined in terms of exitports and entryports and so from the outside it is not possible to distinguish between a task and a group module. This provides configuration abstraction. In the patient monitoring system the 'bedmonitor' module is actually defined by a group as shown in fig. 8 The group interface ports are bound to the ports of component module instances using **link** statements within the group module specification. For example in fig. 8, the internal 'monitor.alarms' exitport is linked to the 'alarms' exitport at the group interface. Similarly the 'status' interface entryport is linked to internal 'monitor.status' entryport.

This linking is merely a name mapping and does not entail any run-time overheads ie. there is no copying or queuing of messages at interface ports to group modules. The interface port name is global within the group specification and must be unique, whereas ports on different module instances can have the same name as they are identified by "module_name.port_name".




```
group bedmonitor(scanrate:integer);

   use patienttypes: alarmstype,
                     patientstatustype;
   exitport  alarms:alarmstype;
   entryport status:signaltype
                    reply patientstatustype;

        { -- now define group structure}
   use  monitoring,beddisplay;
   create monitor : monitoring(scanrate);
          display : beddisplay;

   link {internal}
          monitor.prompts  to display.area1;
          monitor.data     to display.area2;
          display.keyboard to monitor.input;
        {interface}
          monitor.alarms   to alarms;
          status           to monitor.status;
end.
```

Fig. 8 Bedmonitor Group

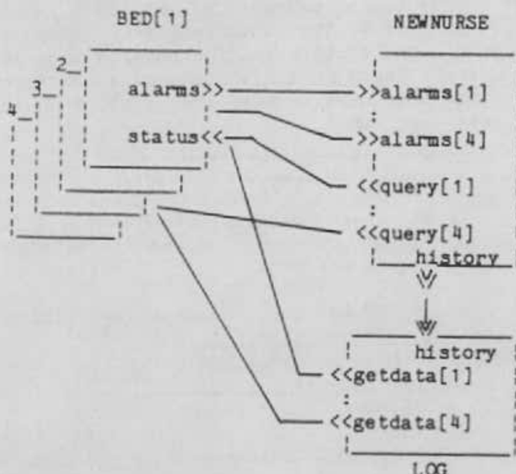
A **Segment Module** is a restricted form of group module in that the constituent module instances share an address space and so must be in a single station. The components of a segment module may share procedures and pass pointer values in messages. However there is no global data within a segment module. Any shared data must be encapsulated within a task module and references explicitly passed by messages. The only synchronisation primitives available for control of access to shared objects in a segment module are the message primitives. For example the Conic modules which implement the various layers of the communication system within a station pass pointers to

message buffers in order to reduce the overheads of copying messages between these modules.

Also a group of tasks may be required to provide parallelism within a particular function. The terminal driver is an example of a set of closely related tasks which are grouped to form a segment module (ie. input from a terminal keyboard and output to its screen). This capability for parallelism at the task level encourages simpler cooperating sequential tasks rather than multi-threaded ones. The concept of a segment module is similar to that of a Guardian in Argus [Liskov 83], but our segment modules do not automatically provide resiliency.

Change Specifications

A CONIC system is flexible in that it allows easy configuration changes. This is achieved by means of a change specification. In the following example the patient monitoring system is extended by including a 'datalogger' module and the module type 'nurseunit' is replaced with a new module type to enable the nurse to display history information from the 'log' module.




```
change ward;

    const  scanrt2 = 500;
    unlink family k:[1..nbed]
           bed[k].alarms from nurse.alarms[k];
           nurse.query[k] from bed[k].status;
    delete nurse;
    remove nurseunit;

    use    datalogger, enhancednurseunit;
    create log:datalogger(scanrt2)
           newnurse:enhancednurseunit;

    link family k:[1..nbed]
           log.getdata[k] to bed[k].status
           bed[k].alarms  to newnurse.alarms[k];
           newnurse.query[k] to bed[k].status;
    link   newnurse.history to log.history;
end.
```

Fig. 9 Change Specification

The change specification uses the configuration constructs described previously, but must also specify the inverse functions. **Unlink** - disconnects a module exitport from an entryport. A named module instance is **deleted** to remove it from the system. This can be performed only after all ports have been unlinked. The **remove** construct removes the type from the configuration specification, and is valid only after all instances have been deleted.

A change specifications is applied to a system configuration to produce a new system configuration. For a **static** system this implies stopping the system, rebuilding a new one and reloading the system into the target distributed stations. This is the approach taken by ADA and CSP.

Dynamic configuration is necessary for applications where it is too costly or unsafe to shut down a complete distributed system in order to change a component. A CONIC system allows arbitrary, unpredicted modification and extensions to an existing system without rebuilding the entire system [Kramer 85]. Changes which can be performed on a running system include:

Installation and removal of module types;
Creation and deletion of module instances;
Changes to the interconnections between modules.

These changes are performed by submitting a change specification to an **on-line configuration manager** which validates the change and produces a new system specification incorporating the changes. It also generates the necessary commands to the operating system to perform the changes.

Some incremental changes can be performed without stopping or affecting any parts of the system. For example if the 'log' module in fig. 9 were added without replacing the 'nurse' module, none of the other modules would be affected. The additional links to the 'status' entryports can be performed without stopping the 'bed' modules. However installing the 'newnurse' module requires deleting the old 'nurse' module. The 'bed' modules can continue running but any alarms sent while the change is taking place will be lost. We do not try to make changes completely transparent. Any request-reply transaction performed on a deleted module will fail. There is no automatic saving of internal state information for a module being replaced as the old state information may not be meaningful to the replacement. If the transfer of state information to a replacement is required, this must be explicitly programmed by saving the information in a file or to another task.

The change specification is kept as a history of change and it is fairly easy to apply the inverse of the change to go back to the original system, for example after testing a change on an existing system.

Logical to Physical Mapping

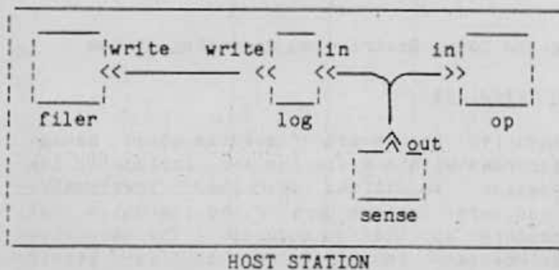
The physical topology supported by a Conic system consists of Local Area Networks (LANs) interconnected by store-and-forward gateways [Sloman 83]. A station can communicate with any other station, if necessary via a gateway. A variety of LANs can be used to suit particular application requirements eg. Ethernet [Xerox 80], Cambridge Ring [CR 82], or one of the emerging IEEE LANs [IEEE 82]. The store-and-forward gateways provide implementation flexibility at the data-link layer, allowing the interconnection of LANs of different transmission rates. The topology of interconnected subnets allows flexible extensions either of stations within a subnet or of subnets within an overall network.

The only constraint imposed by the configuration level on the interconnection of module exitports and entryports is that they are compatible in terms of message and transaction type. The logical interconnection is completely independent of the physical configuration of the hardware components on which the system is to be run. The same logical configuration can be mapped onto a single computer, a closely coupled multi-processor station or distributed stations connected via an arbitrary network. It is thus important to separate the

specification of a logical configuration from its mapping onto a physical configuration. This can be achieved in two ways. If no mapping is specified in a group module, all module instances within the group will be created at the same station as the enclosing module. This is used for small groups of modules such as the bedmonitor in Fig. 8. Alternatively module instance parameters can be used to define "logical stations" at which internal modules should be create, as shown in fig. 10. The logical group configuration of fig. 10a can be mapped onto two different physical configurations as shown in figs. 10b & 10c. The **at** construct can be used to specify co-location eg. fig. 10a specifies that the module 'op' is to be located at the the same station as 'log'.

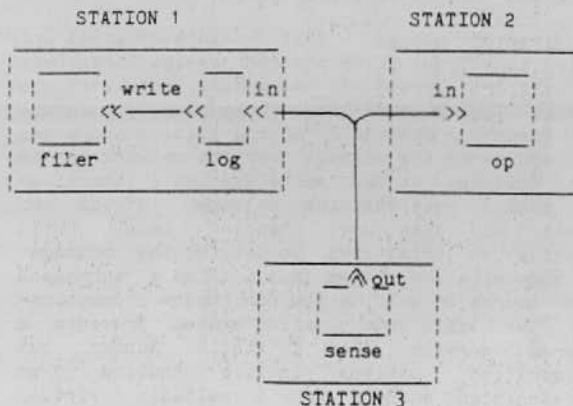
```
group module monitor (stna, stnb : module;
                      filer : filesys);

     use    sensor, operator, logger, filesys;
     create sense : sensor   at stna;
            op    : operator at stnb;
            log   : logger   at filer;
     link   log.write to filer.write;
            sense.out to op.in, log.in;
end.
```

Fig. 10a Logical Configuration




```
system test;
     const host = node (1); {host station address}
     use monitor, filesys;
     create filer : filesys at host;
            mon : monitor (host, host, filer);
end.
```

Fig. 10b Centralised Allocation




```
system final;
     use monitor, filesys;
     create filer : filesys at node (1)
            mon : monitor (node(2),node(3), filer);
end.
```

Fig. 10c Distributed Allocation

Another use of instance parameters is shown in fig. 10a. An exitport of the module 'log' is linked to an external module instance 'filer' whose name is imported by means of an module instance parameter. Without this, the group monitor would have required an interface exitport to which 'log.write' was linked. The link from the monitor's interface exitport to 'filesys.write' would have had to be performed at a higher level where filer and an instance of monitor were created. Importing module instances allows the strict hierarchy of nested modules to be bypassed and avoids the need for "floating" ports to the highest level.

RUN TIME SUPPORT

In this section we describe the run-time support software that is needed to perform dynamic changes in a distributed system. CONIC is designed for embedded applications such as process monitoring and control, factory automation etc. In such an environment, software components are not "malicious" and should be well tested before being installed into the system. This means that run-time validity checks such as access rights or message type compatibility can be avoided. Possession of an exitport of the correct type linked to an entryport gives a capability to access the service provided via the entryport. All access controls can be in terms of visibility of type and interface definitions, which can be held in a database and use the standard access control provided by the database (eg. passwords). All validity checks are then performed at configuration time, and so entail no run-time overhead. However, if an untyped language or assembler is used for programming task modules rather than the CONIC programming language then run time checks will be needed.

Configuration Manager

An on-line configuration manager is needed to support dynamic reconfiguration of a running system. The configuration manager validates the change specification and translates it into commands to the distributed operating system to execute the reconfiguration operations. The configuration manager requires information on the current state of the system (eg. is a component type already in a station or will it have to be downloaded?) and must also have access to information necessary to perform validity checks. Some of this information can be obtained by querying the system to check its current state but type information is not maintained in stations and so must be held in an online database. A single change may result in a number of commands to the system. For example creating a component instance may generate a series of actions to query resources at the target station, load the module type and finally create the instance.

The configuration manager currently being designed consists of three parts: a database describing the current system, a specification translator and a command executor (Fig. 11). The initial version of the configuration manager will be centralised but later versions will be decentralised.

The **Configuration Database** holds task module types which are the code generated by the module compiler, as well as a library of definitions units. Symbol table and port interconnection information is held in the form of symbol files. An up-to-date current system specification is held so that it can be queried at any time and a history of change is maintained in time order. The database also holds physical configuration information on the subnets (address, type, stations connected and current status) and descriptions of each station (address, memory, devices connected and software installed).

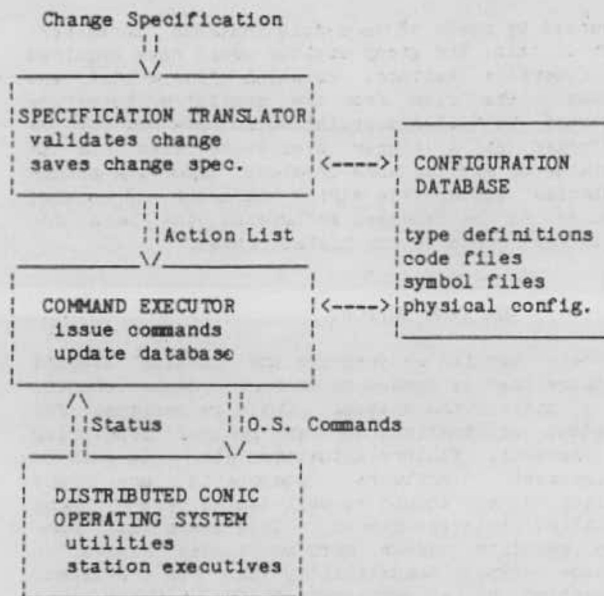


Fig. 11 Configuration Manager

The **Specification Translator** validates the change specification with respect to availability of resources (eg. memory or I/O devices) as well as type and operational compatibility for interconnections. It uses the symbol tables in the database to map the names in the specification into system addresses eg. a port address is specified by "subnet_id.station_id.module_id. port_id". The symbol tables in the database are updated to reflect the changes. The translator also produces an action list which is a sequence of simple commands to the operating system. These are passed to the command executor.

The **Command Executor** performs the operations specified in the action list on the distributed operating system by means of CONIC communication primitives. The command executor updates the state information held in the database to reflect the changes performed on the system. In order to keep the system and its specification consistent, the system is returned to its original configuration if any commands fail.

The configuration manager is not yet available although the operating system support for dynamic configuration has been implemented.

Distributed Operating System

The CONIC distributed operating system supports the dynamic configuration described above and also provides intermodule communication. It conforms to a layered structure where each layer provides services used by the layer above (fig. 12). The distributed operating system consists a set of utilities which are not replicated in every station and an executive which is in every station.

The main influences on the design of the CONIC distributed operating system [Magee 84] were that the station executive should be small and efficient so that dynamic configuration could be provided on small microprocessor systems without backing store. This led to the principle of providing minimal functionality in the executive present in every station and rather implementing as much as possible remotely by utility modules. The executive should itself be configurable so that smaller ROM stations could omit the dynamic configuration support. This was accomplished by implementing all of the station operating system components as a set of CONIC modules which can be configured using the static configuration facilities. The flexibility of the CONIC module structure has been exploited in allowing distribution of the operating system components.

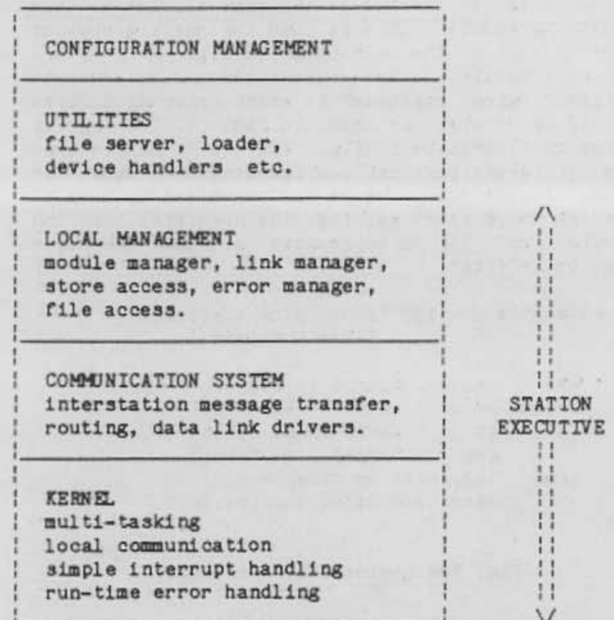


Fig. 12 Conic Distributed Operating System

Station Executive

The executive is the set of modules which manage the resources within a station and implement the communication primitives described previously. Device drivers are **not** part of the executive but are considered application modules. The executive is implemented in CONIC and so is itself configurable using the configuration language.

Station kernel - This consists of the run-time procedures accessed by task modules and a kernel task module. It provides multitasking and the primitives used by the executive's local management modules for task execution control and port linkage. It also handles run-time errors and supports the language extensions to Pascal ie. inter-task message communication within a station, timing and input/output primitives.

Communication system - This consists of a set of modules to support inter-station message transfer. An exitport linked to a remote entryport is actually linked to a local communication module which formats a message by adding station addresses etc. and sends the message over the network to the remote station. At the remote station a communication module receives the message, strips off headers and then uses standard local Conic communication primitives to deliver the message. The communication system thus acts as a surrogate local source or destination for remote communication. The basic communication system providas a datagram service over a single subnet but configuration options include routing over interconnected subnets and a reliable virtual circuit service [Sloman 83].

Local management - This is a set of Conic modules: **modulemanager** deals with the loading of task types and creating instances; the **linkmanager** handles requests to link exitports of task instances within the station to either local or remote entryports; **storeaccess** allows remote reading or writing of blocks of memory and is used for both down-line loading and remote debug; **errormanager** receives run-time error messages detected by the kernel or issued by a module and reports them to a selected

destination; **filemanager** handles the Pascal File I/O requests.

The compiler automatically generates a number of standard ports for every task module:

'Config' entryport can be used to detect that the task is in a quiescent state or to tell the task to perform tidy-up before a configuration operation is performed.

'Stdfile, stdread, stdwrite' exitports are used for standard pascal input/output and are linked to the filemanager.

'Stderror' exitport is used by the kernel (or by the task itself) to generate error messages, for example if the task fails. By default it is linked to the errormanager, but an application can provide its own error manager which takes application specific recovery action when a task fails.

Configuration Operations

The CONIC operating system provides the following dynamic configuration operations:

Load (stationid, codefile, moduletypeid)
The loader obtains the code size from the code file and sends a load request containing the moduletypeid to the target station. The station's module manager allocates memory space for code and returns the start address of the code segment. The loader forms a load image and sends load blocks to the station's storeaccess module.

Unload (stationid, moduletypeid)
The station's module manager deletes the moduletypeid and deallocates the storage for the type code. It can only be performed after all instances of the type have been deleted.

Create (stationid, moduletypeid, moduleinstanceid, parameterlist)
The station's module manager is given an identifier for the module instance and instantiation parameter values. The module manager assigns data segments, initialises control blocks etc. The module type code must have already been loaded into the station.

Delete (stationid, moduleinstanceid)
The module manager checks that the module ports are unlinked and deletes the module instance from the station.

Link (exitportid, entryportid)
The request to link an exitport to an entryport is sent to the linkmanager in the same station as the exitport. The entryportid is placed in the exitport's data structure (no information about a link is held at the entryport). A link to a remote entryport is actually made to the local communication system.

Unlink (exitportid, entryportid)
The entryport address is removed from the exitport data structure. If a request-reply transaction is in progress it will fail.

Start (stationid, moduleinstanceid)
The module manager requests the kernel to make the task module runnable.

Stop (stationid, moduleinstanceid)
The module manager in the target station requests the kernel to stop the task module.

Operations such as querying the state of tasks in a remote station are accomplished by using the store access module to read the relevant kernel data structures.

THE CONIC TOOLKIT

The CONIC toolkit for building distributed systems has been designed for a host/target development environment. A host Unix system provides the program development facilities and can be used for initial testing of a system. The system can then be installed onto a target distributed system by down-line loading, floppy disc or ROM. The on-line configuration manager performs dynamic changes.

Programing Language Compiler

The Compiler is used to compile task modules and definition units, which may import precompiled definition units. The compiler is based on the Amsterdam Compiler Kit (ACK) [Tanenbaum 83] which produces an intermediate code called EM. There are a number of back-ends to translate EM to different machine codes. The ACK Pascal compiler has been modified for CONIC and to produce a **symbol file** This contains information about a task module's interface (ie. type and address information about ports and instantiation parameters) and resource requirements (ie. code, data, stack and heap size). If the task is compiled with a debug option then the symbol file also holds information about the tasks internal global variables etc. The symbol file is machine independent, but a code file must be produced for each type of target processor.

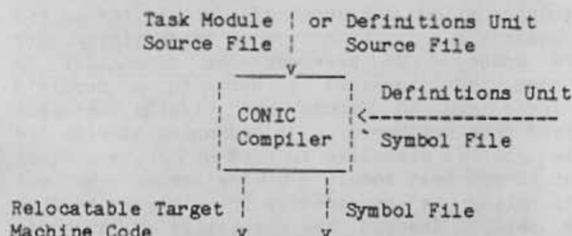


Fig. 13 CONIC Compiler

Static System Builder

The distributed operating system which supports dynamic configuration consists of a set of CONIC modules. A static system builder is needed to produce a load image of the basic software in each station in the distributed system. This operating system is itself configurable and static systems may omit the dynamic configuration support and only use the static builder.

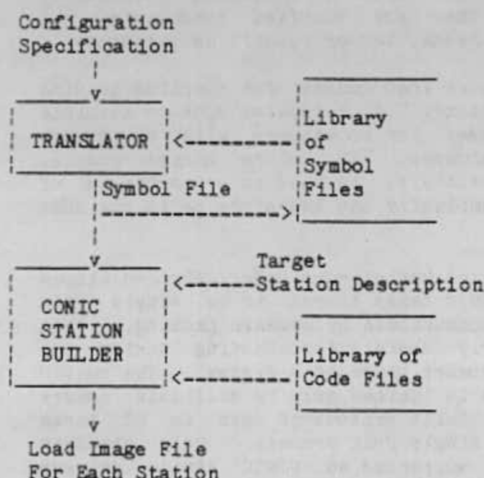


Fig. 14 The CONIC Static System Builder

The static system builder consists of a **translator** and **station builder**. The former translates system or group module configuration specifications to produce a symbol file similar to that produced by the compiler. The translator validates the configuration specification which includes checking that module instance parameters are of the correct type and that exitports are linked to entryports of the same data and transaction type. The station builder uses the descriptor file and a description of the stations in the system (processor type and memory size) to produce a load image for each station.

The builder can also process a change specification to update a system specification and produce a new load image for each station ie. the complete system is rebuilt to incorporate the change.

Debug Facilities

The station bootstrap program provides the identical interface to the store access module described under local management. If a station crashes it automatically goes into bootstrap mode and so the bootstrap program can be used to read memory blocks for analysis on the host system. If an individual task fails an error message is automatically generated by the kernel on the task's standard error exitport. A core dump can then be transferred to the host system. If the task was compiled with a debug option, the postmortem dump analyser uses its symbol table to produce state information about its variables in a readable format.

A debugger allows a remote module to be tested via its message passing interface or by examining its memory space. It provides the capability to construct test messages to send to a module's entryports and to decode and display messages received from exitports. It communicates with the remote station executive to read or write memory blocks in the test module's memory space. Minimal functionality is provided by the target station under test. Instead the complexity of a human interface can be supported at a development station or any other suitable station.

Unix Host Development Environment

All software development is performed on a host Unix system and so the compiler and station builder run under Unix. This allows access to Unix tools such as Macro Preprocessor, Make and Revision Control System.

A single command is used to compile CONIC modules and build groups or station load images. If Unix Make files are used to maintain dependency information then any modified components are automatically recompiled or rebuilt as necessary.

A loader produces load images for downline loading to target stations. It relocates code to absolute memory addresses for processors with no memory management hardware. The store access module, described previously, is used to write blocks of code. All complexity can therefore be in the host system.

A version of the kernel runs under Unix and allows a set of CONIC tasks to run as a single Unix process and communicate by message passing. This is particularly useful for testing during the program development phase of a system. The number of CONIC tasks is limited only by available memory and we have built systems of upto to 80 tasks running as a single Unix process. Only standard Unix I/O is supported so CONIC device drivers cannot be fully tested.

Currently only local CONIC communication is supported within a Unix system but we intend to allow, remote communication between CONIC tasks running under Unix and CONIC tasks running on target stations. This allows access to Unix file systems and peripherals such as printers, from remote CONIC stations.

FUTURE WORK

Some initial work has been done on incorporating fault tolerance techniques into a Conic distributed system [Loques 84]. Both hot and cold standby redundancy can be supported. The configuration facilities are used to automatically switch to a cold **standby** module after a failure is detected. These can be used for applications which can accept the comparatively short time it takes to link and start a module. No state information is preserved. Applications which require completely transparent failure recovery can include a **hot-standby** module. The active module (performing the function) transfers state information at defined points during its operation to the passive 'hot-standby' module. In the case of a failure we automatically switch to the passive module and it assumes the active role. A new hot-standby passive module can be created. The hot standby approach to fault tolerance effectively masks module failures. This seems appropriate for many real-time applications. An interesting aspect is that the configuration manager can itself be made fault tolerant using these techniques. Additional work is needed to incorporate the support for fault tolerance of transactions, such as the provision of atomicity.

CONIC is being used to implement a real-time database that supports global or shared data such as plant state information, setpoints, histories or logs as distributable data modules. These complement the CONIC task modules which perform processing in a real-time system. Data modules support replication of data for efficiency purposes, atomic transaction over data in multiple modules and different views of the data stored in the database. The database uses the CONIC configuration facilities to allow modification without rebuilding the whole database [Andriopoulos 85].

The Conic environment currently supports a single module programming language which simplifies some of the problems associated with transformation of information representation for communication between non-homogeneous computers. The port data structures do not currently hold the type information needed for such transformations. We intend to investigate the problems associated with communication between both non-homogeneous computers and different languages. The configuration flexibility provided by Conic could then be extended to building distributed systems consisting of modules implemented in other procedural languages such as Ada or C. We also intend to investigate the use of Prolog as a module programming language. This will allow an "intelligent knowledge base" to be included in a real-time system or CONIC could provide the modularity framework for building distributed expert systems.

We are investigating the provision of specifications for the behaviour of individual task modules which could then be used in composition rules to specify the composite behaviour of group modules. A sound, practical approach would provide the basis for module and system verification. It would allow analysis of a configuration specification for properties such as deadlock and whether it preserves specified constraints. Such specifications could also be used to predict the effect of configuration changes on the behaviour of a system.

CONCLUSIONS

A prototype toolkit based on a RT11 host development system has been in use for a number of years. It supports LSI 11 target computers interconnected by an Omninet serial bus or Cambridge Ring. We now have about 4 years experience of using earlier versions of the programming and configuration languages for implementing operating system utilities, device drivers, communication systems, and distributed simulations such as a conveyor belt control system. The toolkit has been used by experienced systems programmers and students for project work. The prototype software is also being used by the National Coal Board for implementing software for distributed underground monitoring and control stations, and at Sussex University for research into distributed self-tuning controllers [Gawthrop 84].

Programmers with experience of only sequential systems do have some difficulty in adjusting to designing concurrent systems. However CONIC does make this slightly easier in that the number of new concepts to be assimilated is comparatively small for those with experience in Pascal or a similar language. The experience of CONIC users has shown that it provides an extremely simple yet very flexible approach to structuring a problem as a set of communicating components. Even comparatively naive student users have found Conic easy to use for building both distributed and centralised concurrent systems. We have found the configuration independence of CONIC modules has allowed the reuse of existing modules in many different situations. This has reduced the effort needed to build new applications. We have found the nesting of group modules to be a very useful abstraction mechanism.

The Unix based development system is far more powerful than the prototype and yet is much easier to use. It has only recently been distributed outside Imperial College, so we have not yet received reports on user experience. Imperial College and other educational establishments intend to use CONIC for student programming exercises in real-time and communications courses. There has been considerable interest in the use of CONIC for a variety of control and monitoring applications both in the U.K. and in other countries.

An evaluation system is available which supports the programming of CONIC modules and building groups of modules to run on a PDP 11 or Vax Unix system. We currently only support LSI 11 target processors. We hope to support M68000 Unix hosts and M68000 targets in the near future. The on-line configuration manager is not yet available and so the system currently only supports static building. However all the run-time support for dynamic configuration has been implemented in the station executives and tested via the debugger. The prototype configuration manager will be implemented within the host development system and so will be centralised. We intend to investigate alternative strategies for distributing the configuration manager both to improve reliability and to allow faster configuration changes.

ACKNOWLEDGEMENTS

Keven Twidle and Naranker Dulay have contributed substantially to the concepts described in this paper and have been responsible for the implementation of the configuration management and compilation tools. We gratefully acknowledge the support of the SERC under grant GR/C/31440 and the National Coal Board. The views expressed are those of the authors and not necessarily those of the NCB.

REFERENCES

[Andrews 82] Andrews G. The Distributed Programming Language SR - Mechanisms, Design and Implementation. Software Practice and Experience, 12, 1982, pp. 719-753.

[Andriopoulos 85] Andriopoulos X., Sloman M. A database model for distributed real time systems. Imperial College Research Report 1985. (in preparation)

[CR 82] Cambridge Ring 82 Interface Specifications. SERC, Sep. 1982.

[DOD 80] USA Department of Defence. Reference Manual for the Ada Programming Language. Proposed Standard Document, July 1980.

[Dulay 84] Dulay N., Kramer J., Magee J., Sloman M., Twidle K. The Conic configuration language: version 1.3. Imperial College Research Report DoC 84/20, Nov. 1984.

[Gawthrop 84] Gawthrop P. Implementation of Distributed Self Tuning Controllers, EUROCON 1984, Brighton.

[Hoare 78] Hoare C. Communicating sequential processes, CACM, 21:8, Aug. 1978, 666-677.

[Kramer 83] Kramer J., Magee J., Sloman M., Lister A. CONIC: an integrated approach to distributed computer control systems. IEE Proc. Pt. E., 130:1, Jan. 1983, 1-10.

[Kramer 84] Kramer J., Magee J., Sloman M., Twidle K., Dulay N. The Conic programming language: version 2.4. Imperial College Research Report DoC 84/19, Oct. 1984.

[Kramer 85] Kramer J., Magee J. Dynamic configuration for distributed systems To be publ. in IEEE Trans. Software Eng. 1985.

[Liskov 83] Liskov B., Sheifler R. Guardians and actions: linguistic support for robust distributed programs, ACM TOPLAS, 5:3, July 1983, 381-404.

[Loques 84] Loques-Filho O., Kramer J. An Approach to fault tolerant distributed process control software, TELCON 1984, Greece.

[Magee 84] Magee J. Provision of flexibility in distributed systems. Imperial College Ph.D. Thesis, April 1984.

[Sloman 83] Sloman M., Kramer J., Magee J, Twidle K. A flexible communication system for distributed computer control. Proc. 5th IFAC Workshop on DCCS. May 1983, Pergamon Press.

[Tanenbaum 83] Tanenbaum A., van Staveren H., Keizer E., Stevenson J. A practical tool kit for making portable compilers. CACM. 26:9, Sep. 1983, 654-662.

[Xerox 80] XEROX Corporation. The ETHERNET: A local area network, data link and physical layer specifications. Version 1.0, Sep. 1980.

[Wegner 84]: Wegner P. (1984). Capital intensive software technology. IEEE Software, 1:3, July 1984, 7-46.

[Wirth 77] Wirth N. Modula: a language for modular multiprogramming. Software Practice and Experience, 7, 1977, 3-35.