Dynamic Configuration for Distributed Systems

JEFF KRAMER AND JEFF MAGEE

Abstract—Dynamic system configuration is the ability to modify and extend a system while it is running. The facility is a requirement in large distributed systems where it may not be possible or economic to stop the entire system to allow modification to part of its hardware or software. It is also useful during production of the system to aid incremental integration of component parts, and during operation to aid system evolution. The paper introduces a model of the configuration process which permits dynamic incremental modification and extension. Using this model we determine the properties required by languages and their execution environments to support dynamic configuration. CONIC, the distributed system which has been developed at Imperial College with the specific objective of supporting dynamic configuration, is described to illustrate the feasibility of the model.

Index Terms—Configuration, configuration process, configuration specification, distributed systems, flexibility, reusability, system evolution.

I. System Evolution

THERE IS a very real need for large embedded computer systems to accommodate **evolutionary** change, particularly those systems with a long expected lifetime. They need to evolve as human needs change, technology changes, and the application environment changes. 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 will need to evolve [1]. These changes may require modification of a function already provided by the system, or extension by the introduction of new functions. It may involve a change in implementation due to a technology change, improved implementation techniques, or to provide redundancy in the system.

In general, evolutionary changes are difficult to accommodate as they cannot be predicted at the time the system is designed. Consequently, systems should be sufficiently flexible to permit **arbitrary incremental** changes.

It has been widely recognized that in order to build large software systems it is necessary to decompose the system into components each of 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 configuration have been referred to as "programming-in-the-small" and "programming-in-the-large," respectively [2]. We concentrate on the latter configuration process as a means of satisfying the flexibility requirements.

Distributed systems potentially offer a flexible environment for such modification and extension. Computer (hardware) stations can be added as and when required, and connected to a network to permit intercommunication with other stations in the system [3], [4]. Similar configuration flexibility is required for the software components of the system. Our experience is that those systems built to accommodate possible physical distribution exhibit much more flexibility due to the enforced modularity and component independence than those designed explicitly for centralized hardware. We therefore propose to cater for distributed systems with centralized implementation as a limiting case.

When can changes or extensions be introduced into a system? Recent work on programming environments, such as the Ada Program Support Environment, has concentrated on configuration management during the software development process. While this eases the problems of software evolution during the development of a product it does not handle the problems of changing a system's software without rebuilding the system. In fact, the traditional edit/compile/link/load approach means that changes must be followed by recompiling and/or rebuilding the entire system. Incremental changes to a system are not well supported.

Also, in a large number of cases, it is not feasible either for economic or safety reasons to stop or take off-line an entire computer system in order to change part of it. Hence, in order to provide for the greatest flexibility, we wish to support **dynamic configuration**. This is the ability to modify and extend a system while it is running.

This paper discusses the issues associated with the need to support and control the dynamic configuration of software in systems which may be distributed. We concentrate on the properties required of the support environment rather than those required of the application. The need for validation of the configuration changes in order to maintain system consistency is also discussed.

Section II of the paper introduces a model of the dynamic configuration process. Section III describes the properties required by programming languages and their supporting systems to allow dynamic configuration. Section IV describes the configuration features of CONIC [5], a language and support system developed at Imperial College which satisfies most of the required properties for dynamic configuration. The software generation and configuration tools contained in the CONIC system are described in Section V. A simple hospital patient monitoring system is used to illustrate some of the

Manuscript received March 16, 1984; revised September 25, 1984. This work was supported by the Science and Engineering Research Council of Great Britain under Grant GR/C/31440, and by the National Coal Board of Britain.

The authors are with the Department of Computing, Imperial College of Science and Technology, London SW7, England.

0098-5589/85/0400-0424$01.00

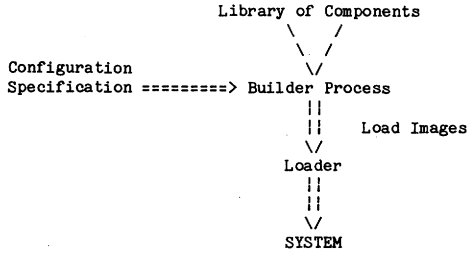


Fig. 1. Static configuration process.

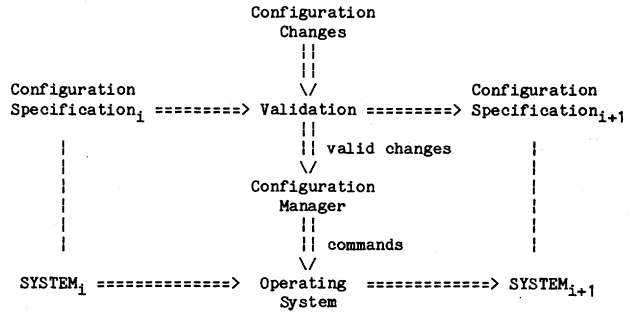


Fig. 2. Dynamic configuration process.

CONIC facilities. Finally some of the remaining problems and the need for further work are discussed.

II. Configuration Model

This section introduces the terminology and basic concepts associated with system configuration. We present a model of the configuration process which permits dynamic incremental modification and extension. This model is used in the next section to determine the properties required by languages and their execution environments in order to support dynamic configuration.

A. Configuration Specifications

As mentioned, systems are constructed from a set of component parts. In the same way as a program source text written in a programming language describes the structure and behavior of a software component, so a **configuration specification** written in a **configuration language** describes the behavior and structure of a system composed of a set of components. For instance, a complete configuration specification for a distributed system will described the types of software components from which the system is to be constructed, the instances of these components which exist in the system, how these instances are interconnected (which instance communicates with which other instances), and where they are located on the distributed system hardware. This last aspect on location can of course be omitted from specifications for nondistributed systems. In this paper we concentrate on logical (software) configurations and the configuration process.

B. Static Configuration

The **static configuration** approach to building a system from its configuration specification is illustrated in Fig. 1.

The builder process is directed by the Configuration Specification to produce a load image of groups of components for each of the computer stations in the distributed system. This builder process is equivalent to the linker in traditional sequential programming systems.

This approach to system building provides a **complete static** system configuration. It is complete in the sense that all components of the system are configured at the same time. If a modified system is required, then the complete system must be stopped and rebuilt according to a **new** configuration specification. The new configuration specification will be an edited version of the old specification incorporating the changes. Thus this approach to system building is exactly equivalent to "edit/compile/link-load" process used for traditional sequential programming. Incremental changes to the system configuration are **not** supported and consequently it is not possible to change systems on-line.

This approach has been taken by most of the recent proposals for programming (distributed) systems, such as Ada [6], DP [7], SR [8], STARMOD [9], CSP [10], and many others. In general these languages are restricted to static configuration because they do not enforce sufficient separation between component programming and configuration. For instance, all the above mentioned languages directly specify the intended destination of an interprocess communication. The interconnections are thus embedded at the programming level and hence do not facilitate configuration changes.

Some languages support dynamic interconnections by name passing. For instance, in SR a message can contain the name of a process entry to which a subsequent message can be addressed. However, the provision of different unpredicted interconnections, and hence not provided by name passing, will still require reprogramming, compiling, and building of the system. Use of the term static is perhaps a little more difficult to justify in language systems which permit dynamic creation of software components, such as Ada. However, the components which can be created dynamically in Ada are of existing component types. Modification or extension of the system by the modification of existing components or the introduction of arbitrary new component types will also require reprogramming and building the system. At best only those stations which are affected by the new configuration specification need to be stopped and reloaded.

C. Dynamic Configuration

An **incremental dynamic** configuration process provides for arbitrary unpredicted modification and extension without rebuilding the entire system. Where possible the incremental changes to the system should be made without stopping the unaffected parts of the system. The dynamic configuration process can be modeled as shown in Fig. 2.

The configuration specification describes the logical and physical structure of the system. Changes to the system are submitted in the form of change specifications such as the introduction of new components, modification of existing components, and provision of different interconnection patterns. Changes must be validated to ensure that the changes are compatible with the existing system. The result is a new specification. A configuration manager generates the required operating system commands such as loading the required code,

deleting and creating components, and removing and setting up connections. The result is a new system described by the new specification. Of course, such a dynamic configuration process must be supported by both the language used to program the components and the underlying (distributed) operating system. The required properties are identified in the next section.

Some existing programming systems for single computers provide a separate configuration language and some degree of dynamic configuration, notably Mesa [11] and MASCOT [12]. In the distributed programming area, CONIC [5], PRONET/NETSLA [13], and PCL [14] have a separate configuration language and allow some form of dynamic configuration. The distributed programming language and system ARGUS [15] permits a large degree of dynamic configuration; however, the configuration statements are embedded in the text of component programs (Guardians). We believe this approach, in addition to restricting flexibility makes it difficult to validate configuration changes. As mentioned before, the most common mistake is that there is insufficient separation between programming and configuration. Features of the above and other systems are used to illustrate the required properties for dynamic configuration discussed in the next section.

III. Required Properties for Dynamic Configuration

The properties required by the different components of the configuration model are categorized into those **essential** for allowing dynamic configuration and those **desirable** for making dynamic configuration either simpler to implement or more efficient.

A. Programming Language

Programming language refers to the language used to program the software components which form the system building blocks.

Essential Properties:

Modularity—The language must provide software **modules** which can be written and compiled independently from the configuration in which they will run, i.e., **context independence**. The statements inside a module should refer only to local objects since reference to global objects such as shared data restricts the ease with which modules can be added or removed. In a distributed system global references also restrict the way modules can be allocated to physical processors.

Interconnection—Similarly, the direct naming of other modules or communication entities restricts the logical configuration flexibility since change would involve modifying these names in the program text and thus require recompilation. A module must communicate with the outside world solely through its interface. Module interconnections should not be embedded in the module code but left to the separate configuration specification. **Interconnection independence** is the key property in separating module programming from configuration [16].

Interfacing—All the information passing into and out of a module must be by an interface which specifies both the type of the information and the mechanism by which it is to be transferred. This **well-defined interface** should be the only logical information required by the configuration level to use the module. Interconnection between modules is specified and checked at the configuration level using the interface information.

Intercommunication Primitives—In distributed systems, the intermodule communication primitives provided by the programming language must have the same syntax and semantics for local (same station) and remote (interstation) communication. The property of **communication transparency** is required whichever communication primitives are adopted. Since modules may be allocated either to the same or different stations, they must have the same behavior for these different allocations to allow full configuration flexibility. By behavior we mean logical behavior since it is unavoidable that the performance or time behavior will be different due to the increased latency of remote communication.

Desirable Properties:

Resource Requirements—It is desirable to know the maximum physical resource requirements of a module when it is compiled. This allows the configuration manager to check that a station can provide these requirements and that the module will not fail once it has started because, for example, it cannot acquire enough store. For some real-time applications this property of **known resource requirements** is essential for reliable deterministic system behavior.

We regard the above properties as collectively contributing to the degree of **configuration independence** exhibited by the components provided by the programming language. Most languages with modules exhibit some but not all of the properties. For example, the program modules of SR (resources), ARGUS (guardians), PRONET (processes), and CONIC (modules), exhibit the property of **context independence** while those of Ada (packages) do not, since Ada packages may access shared data and procedures. PRONET processes and CONIC modules achieve **interconnection independence** through the use of input and output ports from which messages are received and sent respectively. As mentioned in Section II, languages such as Ada, STARMOD, CSP, and SR which use direct naming do not exhibit this property. Most modern programming languages with module constructs provide well defined interfaces using a mechanism similar to the import and export lists of MESA. The property of **known resource requirements** is difficult to provide without abandoning useful programming techniques such as recursion and dynamic variables (heaps). A compromise used in CONIC provides these facilities but specifies an upper bound for the store required by a module. Exceeding this bound causes the module to fail.

B. Configuration and Change Specifications

The configuration language used to describe logical structure of a system is outlined below. We assume that change specifications used to describe modifications to a system are also described in this configuration language.

Essential Properties:

Context Definition–The configuration language must specify the set of module types from which the system is constructed.

Instantiation–It must also specify the instances of module types which are to be created in the system.

Interconnection–Finally, the configuration language must describe the way module instances are interconnected.

Change Specification–To allow the configuration language to express changes to a configuration it must also be able to specify the **inverse** functions to those above. It must be able to specify the disconnection of module instances, the deletion of module instances, and the removal of module types from the context of the system. Changes should appear as updates to the configuration specifications.

Separation of Function–Full configuration flexibility is only possible if each of the above configuration functions can be specified separately. For example, combining module instantiation with module interconnection means that it would not be possible to reconnect a module without deleting it and subsequently reinstantiating it with new connections.

Specification Modularity–In a large system with a large number of module instances, the name space would become unmanageable and the specification unreadable unless it were structured in some way. Structuring of the specification is analogous to the need for structure in system design and construction. It aids both the specification process and comprehension of specifications.

Desirable Properties:

Declarative Configuration Language–It is desirable that the configuration specification be descriptive (declarative) rather than operational (in the sense of a configuration "program" which is executed). In general declarative specifications are more amenable to analysis and manipulation, e.g., equivalence checks, logical consistency checks, and transformations. Also, a specification should describe the current configuration of the system. Specification/system consistency is further discussed in Section III-D. Operational specifications introduce a time dependency which complicates this relationship especially for large systems (i.e., the current configuration would be given by the current state of the configuration program). Additionally, introduction of arbitrary unpredicted changes can be presented as declarative specification updates. An executing configuration program which formed part of the behavior of the system would present implementation difficulties if on-line changes were allowed.

All the systems which provide separate configuration languages support context definition, instantiation and interconnection. However, C/Mesa, the Mesa configuration language, combines instantiation with interconnection. In C/Mesa, instances of modules are connected together by parameterizing them with interface records, the definitions of which are imported or exported by module programs. A similar interconnection method is used in MASCOT, where activities (processes) are parameterized with intercommunication data areas (IDA's). Consequently, both Mesa and MASCOT do not exhibit the **separation of function** property. NETSLA, the PRONET configuration language, and PCL provide separate interconnection by using link statements which connect input ports to output ports. NETSLA is an executable configuration language which performs configuration changes in response to messages sent and events signalled by component processes. In addition to the disadvantages outlined above this incurs a significant run time overhead [17]. Although declarative, PCL attempts to capture both the static and dynamic configuration structure. Component processes can execute instantiation statements. This complication renders PCL descriptions less amenable to analysis.

Finally, a recent report [18] describes a configuration language (LL) based on Milner's calculus for communicating processes CCS [19]. Like CCS, LL provides an algebraic specification of the configuration which is amenable to manipulation and simplification. In CONIC there is an explicit **link** construct for interconnection as opposed to the name equivalence method of CCS. Both approaches have similar expressive power and one can be transformed into the other.

C. Operating System

The distributed operating system is responsible for modifying the running system in response to commands from the Configuration Manager.

Essential Properties:

Module Management–The operating system must provide the ability to load/delete the code for module types into station(s). It must support the creation/deletion of instances of module types. Additionally, it must allow the configuration manager to control the execution of modules (start/stop) and to query the state of the system.

Connection Management–The operating system must provide facilities to establish and delete connections between modules, both for modules located on the same station and, in distributed systems, for remotely located modules.

Communication Support–For both statically and dynamically configured systems, the operating system must support intermodule communication.

Desirable Properties:

Real-Time Modification–To fully exploit the potential of dynamic configuration management, the time taken by management operations should be such that they can be used for on-line real-time reconfiguration of the system in response to failures.

Overhead–The management facilities provided by the operating system should incur a minimal run-time overhead on the normal operation of the system. Additionally, the store required by these management facilities in each station should be kept to a minimum. For applications such as distributed process control, the operating system should be able to support module management on very small microprocessor based stations.

Logical Interconnection–Where stations in a distributed system are not fully interconnected by the physical communication network the operating system should provide full logical communication interconnection (i.e., each pair of stations can directly exchange information). This full logical interconnection considerably simplifies logical to physical mapping (i.e., module allocation) since the configuration manager can

allocate modules to stations with the knowledge that they can always be interconnected.

Flexible OS Configuration—The operating system should itself be capable of flexible configuration to enable it to be used in different systems and to allow it to be changed in the same way as the application system it supports. For example, all the module management elements of the operating system may not be required on very small ROM based systems which are always statically configured. It thus seems sensible that the operating system be constructed using the same module structure as the application system.

A number of distributed operating systems described in the literature have some or all of these properties: Medusa [20], Amoeba [21], Accent [22], etc. The major constraint on the operating system is that the facilities it provides must be integrated with the programming language and configuration language for an efficient and safe system. Failure to achieve efficient implementation can negate the benefits of dynamic configuration. For example, the "evolutionary kernel" for MASCOT which supports dynamic configuration is little used because of its run-time overhead. Also, dynamic configuration can be a dangerous activity if changes are not validated and controlled. Hence the need for the validation and management activities described below.

D. The Validation Process

Essential Properties:

Interconnection—The checks that can be performed to validate both configurations and configuration changes depend primarily on the interfaces provided by the software modules. The minimum check, which must be supported by both the module programming language and the configuration language, is the ability to ensure type compatibility between communicating modules.

Specification/System Consistency—It is essential to ensure that the configuration of the actual system is given by and satisfies the current specification. The system can be considered as an implementation of the specification and verified as in conventional program verification. Even change specifications can be verified using the verification approach first described for data representations [23]. An abstraction function A can be defined to map from the system to its specification, i.e.

$$A(\text{system}_i) = \text{specification}_i.$$

Proof of the correctness of the change commands for some change specification can then be demonstrated by ensuring that

$$A(\text{commands}(\text{system}_i)) = \text{changes}(\text{specification}_i)$$

where

$$\text{commands}(\text{system}_i) = \text{system}_{i+1}$$

and

$$\text{changes}(\text{specification}_i) = \text{specification}_{i+1}.$$

The configuration manager can be thought of as performing the inverse mapping to that of the abstraction function A.

Desirable Properties:

Allocation—The logical-to-physical mapping should be checked to ensure that the resources required by modules can be provided by the stations to which they are allocated. As discussed in Section III modules should have **known resource requirements** to allow rigorous validation.

Behavior—Ideally, it should be possible to perform some semantic checks on the behavior of the configuration based on information provided on the behavior of each module. Semantic validation could enable derivation of the behavior of the projected system by composition of the individual module behaviors and provide checks such as deadlock detection. This is further discussed in the conclusions.

The rigorous type checking of interfaces is relatively easy to provide. Both C/Mesa and NETSLA enable these checks to be performed at configuration time. However, there is no current easy answer to providing semantic checks on configurations.

E. Configuration Management

The configuration manager translates the (valid) changes expressed for the configuration specification into executable commands to the operating system to change the current system. This translation may require knowledge of the state of the system. The required information may be obtained from a database maintained by the configuration manager, and/or from the system itself. For instance, the manager may query the system to decide on the actions required (e.g., is a component type already resident at a particular station or will it have to be down-line loaded?). Also, a single change may result in a number of commands to the system (e.g., create component instance ⇒ query; load; instantiate).

Essential Properties:

Specification/System Consistency—As above.

Desirable Properties:

Change Strategy—In Section III-B we discussed the desirability of using a declarative language to specify both configurations and changes. Consequently, a change specification written in such a declarative language would say nothing about how the change should be performed. This leaves the configuration manager with the freedom to decide (perhaps with human operator help) the strategy used in performing the change. The goal of the change strategy would be to perform the change with minimum disruption to the running system. For example, it could minimize the "down-time" of replacing a module by loading and creating the new module instance before deleting the old one.

Allocation Strategy—Specification of the logical to physical mapping in a distributed system may be complete in that it specifies the exact station to which each module should be allocated. However, if allocation information is less complete the configuration manager can exploit this freedom and allocate modules to stations based on resource information. The configuration manager could thus optimize the use of the distributed system hardware resources. This freedom can be further exploited in failure situations where, if sufficient redundancy exists in the system, the configuration manager could reallocate modules from failed stations to working ones. It

should be noted that this reconfiguration could be performed without changing the configuration specification, i.e., the configuration specification is still satisfied.

In the above sections we have been concerned with the properties required from language systems and their support environments in order to describe and permit dynamic configuration. The systems referenced provide some but not all of the properties required. In the next section we describe CONIC, a system which attempts to exhibit all of the required properties.

IV. Configuration in CONIC

In order to illustrate the identified properties, a patient monitoring system example [24] is developed and described in CONIC. The CONIC system [5] has been developed at Imperial College specifically with the objective of supporting the construction of dynamically reconfigurable distributed systems. CONIC provides a Configuration language [25] for describing systems consisting of interconnected modules, a Programming language [26] for programming module types, and a distributed operating system [27] to support and manage the execution of CONIC systems.

Although CONIC is used as the means for demonstrating how the described facilities can be provided, it is not our intention to advocate CONIC as the only approach, but rather to advocate provision of the identified properties in programming and configuration languages and their environments.

A. The CONIC Programming Language

The programming language is based on Pascal to which message passing primitives have been added. The message primitives provide both unidirectional asynchronous and bidirectional request-response message transactions. **Modules** have **well-defined interfaces** defined in terms of local typed **exitports** to which messages can be sent and **entryports** from which messages can be received. Instances of these modules in a running system may only communicate by message passing which provide the required **communication transparency**. Messages are defined by standard Pascal type declarations. Examples of module **type** definitions are shown below in outline. These module types form part of the patient monitoring system which is used as an example throughout the section.

```
module bedmonitor (scanrate:integer);
   use patienttypes: alarmstype, patientstatustype;
   exitport  alarms:alarmstype;
   entryport status:signaltype reply patientstatustype;
   {
      – The module scans sensors attached to a patient
        every "scanrate" seconds. When the sensor read-
        ings are outside ranges set at a bed-side terminal
        it displays an alarm at the bedside terminal and
        sends alarm messages to "alarms." Data on sen-
        sor readings and ranges are sent to "status" in re-
        sponse to a "signal" request.
   }
end.
```

```
module nurseunit;
   use patienttypes: alarmstype, patientstatustype, maxbed;
   entryport alarms[1 · · maxbed] :alarmstype;
   exitport  query[1 · · maxbed] :signaltype reply patientstatustype;
   {
        – The module displays alarms received on "alarms
        [i]" on a terminal and in response to input at the
        terminal displays the status of a patient by re-
        questing it on "query[i]."
   }
end.
```

CONIC modules can be compiled separately since they contain no references to outside objects (except type definitions), i.e., they satisfy **context and interconnection transparency**. The parameters of a module are resolved when an instance of the module is created at a station. The types (and constants) modules use for communication are contained in type definition units. In these, types are declared by standard Pascal type definitions, e.g.,

```
define patienttypes:maxbed, sensortype, alarmstype, patientstatustype;
      const maxbed    = 16;
      type  sensortype = (bloodpressure, skinresistance,
                          temperature, pulse);
            alarmtype  = (outofrange, sensorfault, noalarm);
            alarmstype = array[sensortype] of alarmtype;
            ..................
end.
```

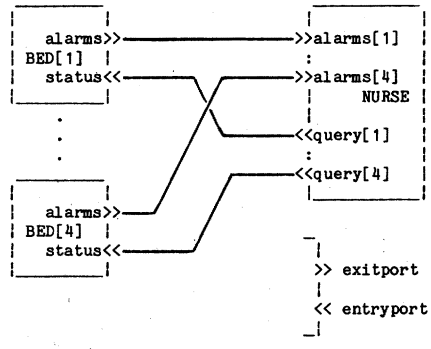


Fig. 3. Ward monitoring system.

Module types written in the CONIC programming language are **configuration independent** as they satisfy all the required properties mentioned in Section III-A.

B. System Configuration Specification in CONIC

Systems in CONIC consist of interconnected sets of module **instances.** Modules are interconnected by linking their **exitports** to the **entryports** of other module instances. The system specification below describes the patient monitoring system of a ward consisting of four beds and a nurse station. Alarms from bed stations are displayed at the nurse station and the nurse station can query the status of any bed (Fig. 3).

```
system ward;

    use bedmonitor; nurseunit;

    const nbed = 4;

    create family k:[1 · · nbed]
            bed[k]: bedmonitor(100);

    create nurse  : nurseunit;

    link family k:[1 · · nbed]
            bed[k].alarms to nurse.alarms[k];
            nurse.query[k] to bed[k].status;

end.
```

A system configuration specification identifies the module types from which the system will be constructed, declares the instances of these types which will exist in the system and describes the interconnection of instances. The three functions of context definition, instantiation, and interconnection are provided by the **use**, **create**, and **link** constructs respectively (separation of function, Section III-B).

In the "ward" system in Fig. 3, the module types "bedmonitor" and "nurseunit" are brought into context in the **use** construct. Named instances "bed" and "nurse" are then **created** using the available module types. If a module type is parameterized (e.g., "bedmonitor"), the actual parameters are specified. The create construct can also be used to create a **family** (array) of module instances as shown for "bed." Finally, the instances are connected together using the **link** construct. This binds entryports to exitports. The constraint on this binding is that an exitport must have the same type as the entryport to which it is linked. For example, the exitports "query[k]" of "nurse" and the entryport "status" provided by each of the family of "bed" must have identical type definitions, i.e., "signaltype **reply** patientstatustype."

Although all the links in the example are from one exitport to one entryport, an entryport in CONIC may have more than one exitport connected to it, i.e., the entryport may provide a service entry to a number of clients. Furthermore, exitports which have no **reply** part (indicating that they can only be used for unidirectional message transactions) can be connected to more than one entryport, i.e., they can be multidestination.

C. Configuration Change Specifications in CONIC

Changes to a system may be made merely by editing the text of its configuration specification and rebuilding the system from this edited specification. However this has two disadvantages; first, there is no record or history of changes made to the system and second, it is difficult to dynamically reconfigure a system since the edited specification must be compared to the original specification to determine the changes the configuration manager must make to the running system. For these reasons changes are specified separately in CONIC, although the change specifications may be applied to the system specification to give a new system specification text if required. The following example outlines the change specification to add a data logging facility to the patient monitoring system.

```
module datalogger(recordfreq:integer);
    use  patienttypes:patientstatustype, maxbed;
         loggertypes:patientidtype, replaydatatype;
    exitport  getdata[1 · · maxbed]: signaltype
                              reply patientstatustype;
    entryport history : patientidtype
                              reply replaydatatype;
    {
      –Reads patient status at "recordfreq" intervals from
      "getdata[i]" and records the information on a floppy
      disk. Information on a particular patient is sent in reply
      to a request from "history"
    }
end.

change  ward;

    use  datalogger;

    create log:datalogger(500);

    link family k:[1 · · nbed]
            log.getdata[k] to bed[k].status;

end.
```

This change is an **extension** to the existing "ward" system; it does not modify the connections of existing "bed" and

"nurse" modules. If performed dynamically, the change does not involve interrupting the execution of these modules. The example below demonstrates a change involving **modification** to the existing system. It replaces the module type "nurse-unit" to enable the "nurse" to display history information from the "log" module.

```
module enhancednurseunit;
   use patienttypes: alarmstype, patientstatustype, maxbed;
       loggertypes: patientidtype, replaydatatype;
   entryport alarms[1 · · maxbed] :alarmstype;
   exitport  query[1 · · maxbed] :signaltype
                     reply patientstatustype;
             history : patientidtype
                     reply replaydatatype;
   . . . . . .
end.

change ward;

   unlink family k: [1 · · nbed]
         bed[k].alarms from nurse.alarms[k];
         nurse.query[k] from bed[k].status;

   delete nurse;

   remove nurseunit;

   use  enhancednurseunit;

   create newnurse:enhancednurseunit;

   link family k:[1 · · nbed]
         bed[k].alarms to newnurse.alarms[k];
         newnurse.query[k] to bed[k].status;

   link newnurse.history to log.history;

end.
```

Remove, **delete**, and **unlink** perform the inverse functions to **use**, **create**, and **link** in configuration specifications. **Remove** removes knowledge of a module type from the system configuration specification. It means that further changes to the system cannot declare instances of the removed module type unless it is again included in the system context by a **use**. **Delete** removes the named module instance from the system and similarly **unlink** removes exitport to entryport links. A change will only be valid if all the links to a module instance are "unlinked" when the instance is "deleted" and all the instances of a module type are "deleted" when the type is "removed." The system resulting from applying both the above changes is described by the following specification (Fig. 4).

```
system ward;

   use bedmonitor; enhancednurseunit; datalogger;
```

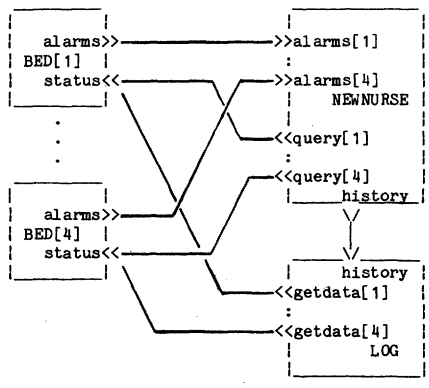


Fig. 4. Modified ward monitoring system.

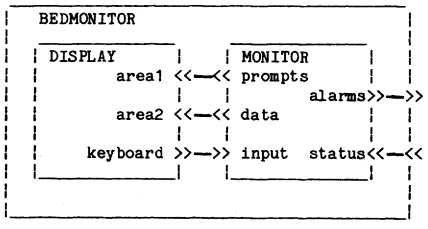


Fig. 5. Bedmonitor group.

```
   const  nbed = 4;

   create family k:[1 · · nbed]
         bed[k]: bedmonitor(100);

   create newnurse: enhancednurseunit;
         log      : datalogger(500);

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

   link newnurse.history to log.history;

end.
```

D. Structuring Configuration Specifications in CONIC

In CONIC, **specification modularity** is provided by the **group** facility. A group describes a set of interconnected instances of modules (or groups) and has an interface defined in terms of typed exitports and entryports. The interface to a group has exactly the same form as a **module** interface and consequently the types referred to in the system specification can refer to modules or groups. For example, in the simulation of the patient monitoring system we have implemented, the "bed-monitor" type is actually defined by a group as shown below and diagrammatically in Fig. 5.

```
group module bedmonitor(scanrate:integer);
   use patienttypes: alarmstype, patientstatustype;
   exitport  alarms:alarmstype;
   entryport status:signaltype reply patientstatustype;

         {—now define group structure}
```

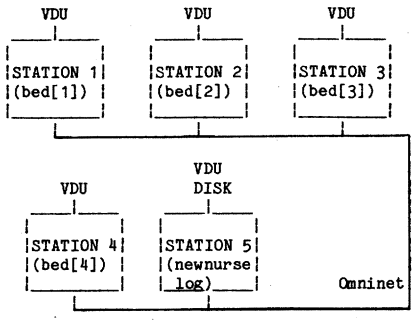


Fig. 6. Ward monitoring system hardware.

```
use monitoring; beddisplay;

create monitor: monitoring(scanrate);
       display  : beddisplay;

link   monitor.alarms    to alarms;
       status            to monitor.status;
       monitor.prompts   to display.area1;
       monitor.data      to display.area2;
       display.keyboard  to monitor.input;

end.
```

The structure of a group is defined using the **use**, **create**, and **link** constructs previously discussed. In the same way that entryports are bound to exitports, group interface ports are bound to the ports of module instances declared inside the group by **linking**.

E. Allocation in CONIC

The allocation of modules to physical stations is expressed by extending the syntax of **create** to additionally name either a module instance with which the new instance should be located or the physical address at which the module should be located. For example, the allocation of the patient monitoring system to physical stations as depicted in Fig. 6 is expressed by the configuration specification below. It should be noted that the configuration specification has been extended to show the basic operating system which must be resident in each station. This basic operating system is itself a group of CONIC modules.

```
system ward;

    use    bedmonitor; enhancednurseunit; datalogger;
           basicOS; diskOS;

    const  nbed = 4;

    create family j: [1 · · nbed]
           station[j] :basicOS at node (j);

    create diskstation:diskOS at node(nbed + 1);

    create family k: [1 · · nbed]
           bed[k] : bedmonitor(100) at station[k];
```

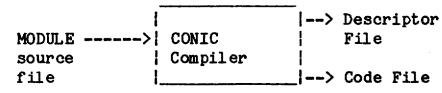


Fig. 7. The CONIC compiler.

```
    create newnurse: enhancednurseunit at diskstation;
           log      : datalogger(500) at newnurse;

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

       link newnurse.history to log.history;

end.
```

The modules within a group are by default allocated to the same physical location. A physically distributed group can be expressed by parameterizing it with the instances with which its constituent modules should be located. Thus, the designer of a group can express the potential distribution of a group even though its user may decide to instantiate it at one location. The **at** construct, as described above, allows the designer to specify which module instances should be colocated without specifying the exact physical location. This freedom can be exploited by the configuration manager as discussed in Section III-E.

This section has outlined a language for describing system configuration; the next section describes the tools necessary to build systems from specifications expressed in this language.

V. The CONIC Configuration Tools

The software generation tools for CONIC have been designed for a host/target development environment. The host computer system provides a module **compiler** and **static system builder**. Once built, systems can be loaded on to the target distributed system by down-line loading, floppy disk, ROM, etc. A **configuration manager**, which can run on the host or target system, is provided to allow systems to be dynamically configured. We require the static configuration capability for two reasons.

1) In CONIC, the operating system which supports dynamic configuration operations on the target system is itself written as a set of CONIC modules. Consequently, to support the dynamic configuration of application systems we must first be able to build the operating system from its configuration specification.

2) For systems (or parts of systems) which do not change it is uneconomic to provide the run-time support for dynamic configuration (e.g., ROM based stations). Since the operating system is itself configurable, it is simple to leave out the modules which perform configuration operations. The operating system and application system are built together from a configuration specification using the static builder.

As shown in Fig. 7 the CONIC compiler produces two output files for each module source file submitted for compilation. The module descriptor file contains a description of the

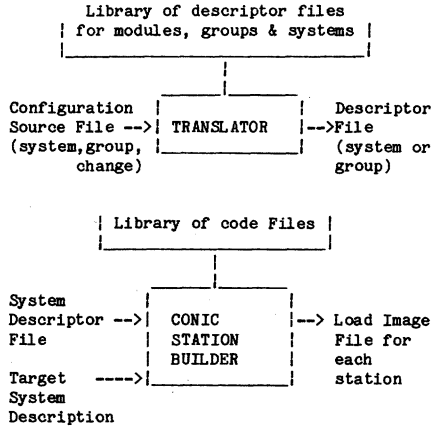


Fig. 8. The CONIC static system builder.

module interface. It is a symbol table which gives the names and types of the module's ports and the addresses assigned to them by the compiler. The descriptor file also includes information on formal parameters. The code file contains the object code in relocatabe form for the module. It also includes a header block which has information on the module's store requirements (i.e., for code, data, heap space). These files are used by both the builder and the configuration manager. Currently, code files are only generated for LSI11 computers. When CONIC supports more than one target processor type a module may have more than one code file, although it will still have only one descriptor file (i.e., a code file for 68000 and a code file for LSI11).

A. Static System Builder

The Builder processes system configuration specifications to produce a load image for each station in the distributed target system. It consists of two programs (Fig. 8). A configuration language **translator** validates **system** and **group** specifications and translates them into a symbol table form (descriptor file) which can be used by the **station builder**. Validation of a system specification includes checking module instance parameters are of the correct type and that exitports are only linked to entryports of the same type. The **station builder** uses the **system** descriptor file together with information on the physical target system to produce load images for each station.

Static building involves submitting a configuration specification to the Translator which produces a System Descriptor File. The Station Builder takes this file and fetches module code files to produce a Load Image for each station in the target system. The "target system description" used by the **station** builder specifies the type (currently only LSI11) and memory size of stations in the target system. The station builder does not need to know how stations are physically connected to each other since the CONIC run-time support provides a communication system which allows a station to transmit a message to any other station (i.e., logically, stations are fully interconnected). It should be noted that the Translator can be used to update a System Descriptor to produce an updated descriptor by processing Change Specifications. To implement this change, the updated System Descriptor

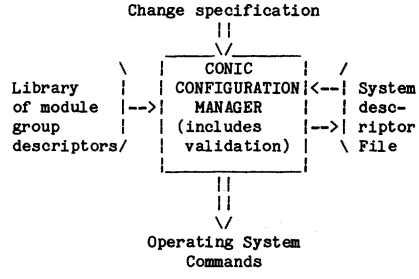


Fig. 9. The CONIC dynamic configuration manager.

file must then be submitted to the Station Builder to produce a new Load Image for each station. The target stations must then be reloaded with the new system. The CONIC Station Builder illustrates the static configuration process given in Section II-B.

B. Dynamic Configuration Manager

The **Configuration Manager** translates requests to change the system, expressed in the CONIC configuration language, into commands to the distributed operating system to execute reconfiguration operations. Although the Configuration Manager could run on the host system and communicate with the target operating system via a communications link, we have chosen to implement it in CONIC enabling it to execute on the target system if desired. Implementing the Configuration Manager as part of the target system allows the system itself to initiate configuration changes in response to failures. The environment in which the Configuration Manager exists is shown in Fig. 9.

The Configuration Manager has a similar function to the Translator in that it validates change specifications against the System Descriptor File and subsequently produces an updated descriptor. The System Descriptor File is a compiled form of the system configuration specification. The initial descriptor file for a system is produced by the Translator. In addition to updating the system descriptor, the Configuration Manager sends commands to the target operating system to invoke reconfiguration operations. The configuration operations provided by the CONIC Operating System satisfy the required properties given in Section III-C and are as follows.

load(stationid, codefile, moduletypeid)
unload(stationid, moduletypeid)

create(stationid, moduletypeid,
moduleinstanceid, parameterlist)
delete(stationid, moduleinstanceid)

link(exitportid, entryportid)
unlink(exitportid, entryportid)

start(stationid, moduleinstanceid)
stop(stationid, moduleinstanceid)

Each object in the target system (moduleinstance, moduletype, port) is allocated a system identifier by the Configura-

tion Manager (or Translator). The Configuration Manager communicates with the Operating system in terms of these identifiers. It should be noted that the Operating System does not know about **groups**. They are purely a specification structuring construct. To illustrate how the above operations are used by the Configuration Manager, the set of operations invoked to effect the change of adding a datalogging facility to the ward monitoring system is outlined below.

```
change ward;

    use datalogger;

    create log:datalogger(500) at diskstation;

    link family k:[1 · · nbed]
            log.getdata[k] to bed[k].status;

end.
```

Operations:-

load(5, datalogger, 27)
- loads codefile datalogger into station 5 with moduletypeid = 27.

create(5, 27, 31, 500)
- creates an instance of moduletype 27 (i.e., datalogger) in station 5 with moduleinstanceid of log = 31.

link((5, 31, 1), (1, 4, 2))
- links log[1].getdata to the monitor .status entryport of the bed[1] group on station 1.

link((5, 31, 2), (2, 4, 2))
- log[2].getdata to bed[2].status

link((5, 31, 3), (3, 4, 2))
- log[3].getdata to bed[3].status

link((5, 31, 4), (4, 4, 2))
- log[4].getdata to bed[4].status

start(5, 31)
- start module instance 31 (i.e., log) in station 5.

The operations **unload**, **delete**, **unlink**, and **stop** undo the effect of **load**, **create**, **link**, and **start**, respectively. Each operation returns a code indicating the success or failure of the operation. Operations are implemented by operating systems modules and invoked by sending messages to the entryports of these modules. The CONIC Manager is responsible for allocation and specification/system consistency (Section III-E) but does not provide any intelligent strategies for change or allocation.

The CONIC compiler has been implemented using the Amsterdam Compiler Kit [28] which includes a table driven code-generator generator. We have currently implemented run-time support for the LSI11 family of computers and work is in progress on VAX and 68000 backends. The operating system which supports dynamic configuration is, as mentioned previously, implemented by a set of CONIC modules and as such requires little work to port it to different target architectures. The communications component of the OS has modules which support Cambridge Ring, Omninet, and serial line hardware. An Ethernet driver module is planned. To simplify the testing of CONIC systems, we have provided the capability of running configurations on the Host operating system (UNIX). Each target station is simulated by a UNIX process.

VI. Conclusions

This paper has been concerned mainly with the problems of describing and modifying the structure of a system. We have presented a basic model of the configuration process which permits dynamic incremental modification and extension. Using this model we have determined the properties required by languages and their execution environments in order to support dynamic configuration. The existing and proposed systems which were surveyed provide some but not all of the properties required. Most do not exhibit configuration indepedence in their programming language, usually because the interconnections between modules are specified in the modules themselves. This appears to be a crucial property. Since most do not even distinguish between programming and configuration, few systems provide a configuration specification at all.

CONIC has been used as an example of a system environment which goes some way to providing the required facilities. We have now had about four years experience using earlier versions of the CONIC programming and configuration languages for implementing operating system utilities, communications systems, distributed simulations, and, of course, games. Perhaps the greatest success of CONIC has been the relative ease with which naive students, with no previous experience of distributed systems, have been able to implement distributed application programs. Also, the configuration independent nature of CONIC modules has permitted the reuse of many existing modules and thereby considerably reduced the effort required to program new applications.

The reconfiguration facilities have attracted much outside interest. A research project is investigating the use of reconfiguration for adaptive control. The Mining Research and Development Establishment in Britain are currently using CONIC to produce software for distributed underground monitoring and control systems. They hope to use the reconfiguration facilities to help introduce the system in a series of phases, and to cope with the changing underground environment as old faces (areas where the coal is cut) are shut down and new ones are opened. Another interesting use of the reconfiguration facilities is to provide fault tolerance. This takes the form of a secondary passive standby module which is configured into the system when the primary module fails. We have provided two forms of standby: "cold" standby does not retain the state information of the primary, and "hot"

standby uses a transparent checkpointing mechanism to ensure a consistent state [29].

In general, one of the main difficulties in dynamically changing a system configuration is to determine the effect of the change on the running system. This has two aspects of interest.

Firstly, how will the system behave while the modification is being performed? We have largely ignored the problem of maintaining a safe and consistent system state while the system is being modified. We believe this problem is separable from the issues in modifying structure. It is concerned with the way a change is performed rather than the change itself. As such, we believe state consistency is a problem of change strategy. The strategy used to perform a change is a function of the Configuration Manager. The more that is understood about how to perform changes while maintaining a safe and consistent system, the more automated will be the Configuration Manager. We are investigating whether the notion of module quiescence [30], a form of module stable state, can be used as an indication of the point at which reconfiguration can be more safely initiated. Mechanisms to save and retrieve module state information will also be required, and there is an obvious need to provide some form of change atomicity to ensure a consistent configuration in the face of possible failures.

The second aspect concerns the resultant system after modification. Determination of the logical behavior of the modified system depends on the existence of specifications of individual module behavior and on methods of composing these behaviors to describe configured system behavior. CCS [19] is one of the few approaches which can do this but the primitives used are restrictive and analysis is difficult. Another approach [31] is investigating the use of regular expressions to specify process behaviors as sequences of events. System behavior (parallel composition) is given by a shuffle operator which combines module behaviors by forcing simultaneous participation (intersection) in those events specified as synchronizing. Although a simple tool has been built to analyze small systems for properties such as deadlock, the expressive power of the approach is limited. Other aspects of a system behavior affected by modification are its timing and reliability. Much work remains to be done in this area.

Acknowledgment

We gratefully acknowledge many useful discussions with our colleagues M. Sloman, K. Twidle, and N. Dulay, and comments on an earlier draft by W. Turski and the referees.

References

[1] M. M. Lehman, "Program evolution," in *Proc. Symp. Empirical Foundations of Information and Software Science*, Atlanta, GA, Nov. 1982.

[2] F. DeRemer and H. Kron, "Programming-in-the-large versus programming-in-the-small," in *Proc. Conf. Reliable Software*, 1975, pp. 114–121.

[3] M. Wilkes and D. Wheeler, "The Cambridge digital communication ring," in *Proc. Local Area Communications Network Symp.*, Boston, MA, May 1979, pp. 47–61.

[4] "The ETHERNET: A local area network, data link and physical layer specifications," Xerox Corp., Version 1.0, Sept. 1980.

[5] J. Kramer, J. Magee, M. Sloman, and A. Lister, "CONIC: An integrated approach to distributed computer control systems," *Proc. Inst. Elec. Eng.*, vol. 130, no. 1, part E, Jan. 1983.

[6] *Reference Manual for the Ada Programming Language*, U.S.A. Dep. Defense, Proposed Standard Document, July 1980.

[7] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *Commun. ACM*, vol. 21, no. 11, pp. 934–941, Nov. 1978.

[8] G. R. Andrews, "The distributed programming language SR–Mechanisms, design and implementation," *Software Practice and Experience*, vol. 12, pp. 719–753, 1982.

[9] Robert P. Cook, "*MOD–A language for distributed programming," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 563–571, Nov. 1980.

[10] C.A.R. Hoare. "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[11] G. J. Mitchell, W. Maybury, and R. Sweet, "Mesa language manual version 5.0," Xerox Palo Alto Res. Center, Rep. CSL-79-3, Apr. 1979.

[12] H. R. Simpson and K. Jackson, "Process synchronization in MASCOT," *The Comput. J.*, vol. 22, no. 4, pp. 332–345, Nov. 1979.

[13] R. J. Leblanc and A. B. Maccabe, "The design of a programming language based on connectivity networks," in *Proc. 3rd Int. Conf. Distributed Computing Systems*, FL, 1982.

[14] B. Liskov and R. Sheifler, "Guardians and actions: Linguistic support for robust distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 381–404, July 1983.

[15] V. Lesser, D. Serrain, and J. Bonar. "PCL–A process oriented job control language," in *Proc. 1st Int. Conf. Distributed Computing Systems*, AL, pp. 315–329, Oct. 1979.

[16] W. R. Franta, E. D. Jensen, and R. Y. Kain, "Real-time distributed computer systems," *Advances in Comput.*, vol. 20, pp. 39–82, 1981.

[17] A. B. Maccabe, "Language features for fully distributed processing systems," Georgia Inst. Technol., Rep. GIT-ICS-82/12, Aug. 1982.

[18] L. E. Thorelli, "A Linker allowing hierarchic composition of programs," CSALAB Working Paper, 1982-09-1.

[19] R. Milner, *A Calculus of Communicating Systems* (Lecture Notes in Computer Science, vol. 92). New York: Springer-Verlag 1980.

[20] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An experiment in distributed operating structure," *Commun. ACM*, vol. 23, no. 2, pp. 92–104, Feb. 1980.

[21] A. S. Tanenbaum and S. J. Mullender, "An overview of the Amoeba distributed operating system, *ACM Operat. Syst. Rev.*, vol. 15, no. 3, pp. 51–64, 1981.

[22] R. Rashid and G. Robertson, "Accent: A communication oriented network operating system kernel," in *ACM Sigops Proc. 8th Symp. Operating Systems Principles*, CA, Dec. 1981, pp. 64–75.

[23] C.A.R. Hoare, "Proof of correctness of data representations," *Acta Inform.*, vol. 1, pp. 271–281, 1972.

[24] W. P. Myers, G. F. Myers, and L. C. Constantine. "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.

[25] N. Dulay, J. Kramer, J. Magee, M. Sloman, and K. Twidle. "The Conic configuration language version 1.3," Imperial College Res. Rep. DOC 84/20, Nov. 1984.

[26] J. Kramer, J. Magee, M. Sloman, K. Twidle, and N. Dulay, "The Conic programming language version 2.4," Imperial College Res. Rep. DOC 84/19, Oct. 1984.

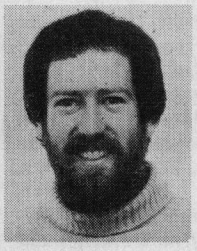
[27] J. Magee, "Provision of flexibility in distributed systems," Ph.D. dissertation, Dep. Computing, Univ. London, Imperial College, Apr. 1984.

[28] A. Tanenbaum, H. van Staveren, E. Keizer, and J. Stevenson, "A practical tool kit for making portable compilers," *Commun. ACM*, vol. 26, pp. 654–662, Sept. 1983.

[29] O. G. loques Filho and J. Kramer, "An approach to fault tolerant distributed process control software," presented at TELCON 1984, Greece, Aug. 1984.

[30] J. Kramer and R. J. Cunningham, "Towards a notation for the functional design of distributed processing systems," in *IEEE Proc. 1978 Int. Conf. Parallel Processing*, Aug. 1978, pp. 69–76.

[31] L. Holenderski, "An approach to mechanised verification of behaviours of concurrent systems," Imperial College Res. Rep. in preparation.



Jeff Kramer received the B.Sc. (Eng.) degree in electrical engineering from the University of Natal, South Africa, in 1970. In 1972 he received the M.Sc. degree in computing, and in 1979 he received the Ph.D. degree for an aproach to the design and verification of distributed systems, both from Imperial College of Science and Technology, London, England.

He is currently a Lecturer in the Department of Computing at Imperial College, teaching courses in program design, software engineering, and distributed systems. His current research interests include the specification and design of distributed systems, and tools for the production of verified software. A special interest of his is in the production of large systems which are expected to evolve as requirements change.

Dr. Kramer is a member of the Association for Computing Machinery, and he is a founder member of the EWICS (European Workshop on Industrial Computer Systems) TC11 on Application Oriented Specifications.

Jeff Magee received the B.Sc. degree in electrical engineering from Queens University, Belfast, Ireland in 1973. In 1978 he received the M.Sc. degree in computing, and in 1984 he received the Ph.D. degree for work aimed at providing flexibility in distributed systems, both from Imperial College of Science and Technology, London, England.

He is currently a Lecturer at Imperial College, teaching courses on operating systems and the programming and design of embedded systems. His current research interests include distributed operating systems and development of support environments for large distributed systems.

Dr. Magee is a member of the Institution of Electrical Engineers.