CONTENTS

SELECTED PAPERS FROM THE ELEVENTH AUSTRALIAN COMPUTER SCIENCE CONFERENCE

49 Guest Editor's Introduction

50-57 A Behavioural Specification of Cache Coherence
    P.J. ASHENDEN and C.D. MARLIN

58-64 A Distributed Operating System Kernel for a Closely-Coupled Multiprocessor
    F.A. VAUGHAN, C.D. MARLIN and C.J. BARTER

65-72 An Integrated Programming Environment Based on Multiple Concurrent Views
    R.A. ALTMANN, A.N. HAWKE and C.D. MARLIN

73-78 An Experiment in Modelling Communication Protocols Using NPNs
    C.C. ANG and E.J.P. COUNTY

79-84 Instrumenting Systems to Measure Components of Interactive Response Times
    J.P. PENNY, P.J. ASHTON and D.G. TRIPP

85-93 Robot Navigation with Learning
    A. ZELINSKY

SPECIAL FEATURES

57 Call for Papers — ISDN in Europe

64 Call for Papers — ACSC 12

72 Obituary — Ross Nealon

94-96 Book Reviews
ONE STEP AHEAD: CIM as a Strategy.

Systec 88

For further Information: German-Australian Chamber of Industry and Commerce, G.P.O. Box 4247, Sydney NSW 2001, phone 02/293996

SYSTEC 88
Following the successful premiere of Systec 86, where international CIM and CAD congresses sponsored by the VDI (Association of German Engineers) were booked solid with a total of 1,600 participants, a topical and technically complementary congress program is being prepared by VDI specialists in anticipation of Systec 88, the Second International Trade Fair for Computer-Integrated Manufacturing to be held in Munich from 25-28 October 1988.

The central theme of this year's International CIM Congress (25-26 October 1988) is "Integrated Information Processing in Manufacturing". Plans have been made to present comprehensive reports within the framework of the congress on the experiences of companies who have implemented integrated information processing to a large extent. Thus, practical recommendations for a sensible introduction to CIM can be expected. Focal points of the congress are CIM planning, impediments to implementation and personnel and profitability problems.

Detailed information and program schedules will be available as of June 1988 from the VDI Society for Production Engineering (VDI-Gesellschaft Produktionstechnik, Postfach 11 39, 4000 Dusseldorf 1, Federal Republic of Germany. Telephone (0211) 6214-518; Telex: 8586525 VDI).

The international CAD Congress on 27 and 28 October 1988 with the theme "Data processing in design 88", will present comprehensive, industry-oriented reports on the status of CAD technology in machine building, automotive engineering, construction engineering, electrical engineering and electronics and in conjunction with computer sciences in one plenary and five parallel sessions. Experiences and problems encountered within the individual engineering disciplines in the application of CAD technology are major program topics.

There will be a panel discussion to address questions from the audience. The congress is sponsored by the VDI Society for Development, Design and Marketing. Co-sponsors are the VDI Society for Construction Engineering, the Federation of German Machine Manufacturers (VDMA), the Central Association of the Electrical Engineering Industry (ZVEI), the VDI/VDE** Center for Information Technologies and the Association for Computer Sciences. The Institute of Mechanical Engineers (GB) and "Ingenieurs et Scientifiques de France" (F) have also agreed to co-sponsor the congress.

Requests for details about the CAD Congress and its final schedule can be addressed to the VDI Society for Development, Design and Marketing (VDI-Gesellschaft, (Continued on inside back cover)
Guest Editor’s Introduction:

Selected Papers from the Eleventh Australian Computer Science Conference

The Eleventh Australian Computer Science Conference was held at the University of Queensland from 3-5 February, 1988. 40 papers were presented at the conference, chosen from the 77 that were submitted. It is interesting to note the increasing activity in computer science in Australia, or possibly the attractions of a Brisbane venue. By previous arrangement between the authors, the conference organisers and the Australian Computer Society, seven of these papers were selected for inclusion in this issue of the Journal. Our intention was to obtain a representative selection of the best papers that would both interest the Journal’s readers, and illustrate the scope of current research in Australia.

One of the seven papers selected, by Abramson and Egan, has been held over until the August issue. It describes an RMIT/CSIRO project to develop a parallel computer architecture and will make interesting reading alongside a paper describing aspects of the AIM system, another Australian multiprocessor architecture project.

Three papers were presented at the conference describing various aspects of the Leopard project underway at the University of Adelaide. This is a major Australian project of great potential significance to our information technology industry and contributes to our knowledge and understanding of workstation-based computing. We have taken this opportunity to publish all three papers in the Journal to present the current state of this important project. The distinctive feature of the Leopard workstation is that its central processor consists of a number of closely-coupled microprocessors that share the same main memory. This architecture presents novel opportunities for the software designer, both in creating an operating environment and in designing application systems that take advantage of the hardware. It also generates interesting hardware design problems. The first paper, by Ashenden and Marlin, discusses solutions the problem of multiple memory caches (one per processor) that need to be kept coherent with each other and with the shared main memory. The second, by Vaughan, Marlin and Barter, examines the structure of an operating system kernel that controls the operation of the multiple processors, and the third, by Altmann, Hawke and Marlin discusses Multiview, an integrated programming environment, that can maintain concurrent but different views of the same object. Those readers who are interested in the Leopard project as a whole would benefit from reading some of the other papers referenced by these three.

Ang and County present a paper describing the modelling of part of the ISO OSI FTAM protocols using numerical Petri nets. Communication protocols are notoriously complex and in the past the difficulty of detecting errors in their definition has led to failures when the protocols were used. Australia has established an international reputation for using formal specification and other methods to define and validate communication protocols. This paper illustrates some of the techniques that are used and presents problems that were uncovered in the ISO protocols.

We are all familiar with the frustration experienced when waiting for a response from a (possibly) heavily loaded timeshared computer system. The paper by Penny, Ashton and Tripp proposes a model for measuring the source of delayed interactive response in an operating system and point out that such models are difficult to implement unless measurement is considered at the time when the operating system is designed and the necessary hooks are built into the design. This is true too of other features, such as management and security, often ignored until much later.

Finally the paper by Zelinsky demonstrates some of the work being done in robotics in this region. It is concerned with the problems of navigating a mobile robot to find a target, negotiating its way amongst a collection of initially unknown objects.

Copies of the complete conference proceedings are available from the Department of Computer Science, University of Queensland, phone: (07) 377-2097.

An important event that happened at ACSC-11 was the formation of the Computer Science Association as an organisation for computer professionals devoted to research both academically and in industry. Readers who are interested should contact the President, Professor C.J. Barter (Chris) at the following address:

Department of Computer Science
University of Adelaide,
P.O. Box 448,
Adelaide, SA 5001
Telephone: (08) 228-5586
ACSnets: chris@uacomsci.ua.oz

The Computer Science Association will be responsible for the organisation of future Australian Computer Science Conferences (next year ACSC-12 will be held in Wollongong in early February). The Australian Computer Journal hopes to continue to print these selections of the best conference papers in future years (subject to discussions with the CSA).

Kerry Raymond,
Rob Cook,
University of Queensland
A Behavioural Specification of Cache Coherence

P.J. Ashenden† and C.D. Marlin†

Multiprocessor systems with shared memory are of increasing interest, because of their flexibility, incremental expandability, and potentially low cost. To reduce bus contention and to improve memory access time, such multiprocessor systems commonly incorporate a memory cache per processor. The use of memory caches leads to the need to ensure that the contents of the caches are coherent with each other and with the shared memory; this is the so-called cache coherence problem. A number of strategies have been proposed to overcome this problem, but little is known about their advantages and disadvantages.

This paper contributes to the study of cache coherence strategies by proposing a formal model of cache coherence. This model, which is an information structure model, is described and its application illustrated by outlining how it can be used to describe the cache coherence strategy used in the Futurebus standard.

Keywords and Phrases: cache coherence strategies, information structure models, multiprocessor computer systems, Futurebus.


1. INTRODUCTION

A multiprocessor computer architecture which has received much attention recently is the bus-based symmetric multiprocessor. It consists of a pool of homogeneous processors connected via a system bus to a globally shared memory. This basic configuration may also be augmented with processors and controllers for particular functions such as I/O interfacing, graphics processing and graphics display. Several examples may be found, both as commercially manufactured systems [e.g. Sequent Balance (Fiel-land and Rodgers, 1984) and Symmetry (Manuel, 1987), Encore Multimax (Anzelmo et al., 1985) and the DEC VAX 8000 Series multiprocessors (Digital Equipment Corporation, 1985)], and as experimental systems for research, e.g. SPUR at U.C. Berkeley (Hill et al., 1986), and Leopard at University of Adelaide (Ashenden et al., 1987)]. The symmetric multiprocessor architecture has a number of important advantages for many applications. See Ashenden, Barter and Marlin (1987) for a discussion of these.

From consideration of processor-to-memory access bandwidths and bus data transfer bandwidths, it is clear that a data cache on each processor is crucial to the successful operation of a symmetric multiprocessor. Without a cache, the bus would be a source of congestion, with each processor having to wait for access to code and data stored in shared memory. A first order approximation would indicate that the use of caches increases the number of processors which can effectively use a bus by a factor related to the cache hit-rate.

Of interest is the work undertaken by the IEEE Future-

Copyright © 1988, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

† Department of Computer Science, University of Adelaide, GPO Box 498, Adelaide, SA, 5001. This paper was presented at the Eleventh Australian Computer Science Conference at the University of Queensland in Brisbane, Queensland in February 1988.


2. CACHE COHERENCE

The model of a symmetric multiprocessor used to consider the behaviour of caches is illustrated in Figure 1. It consists of a shared memory accessible via the system bus and a number of caches, each of which serves a client. (Typically the clients would be processors.) Data in shared memory is
The specification of cache coherence strategies has a number of parallels with the specification of programming language semantics. The same range of possible specifica-

mentioned in experimental systems over the past four years. These include Goodman's write-once strategy (Goodman, 1983), Papamarcos' Illinois strategy (Papamarcos and Patel, 1984), the Berkeley ownership strategy used in SPUR (Katz et al., 1985), and the XEROX Dragon and DEC Firefly strategies (Archibald and Baer, 1986).

These strategies are typically defined in an informal manner. Comparisons between them have been made in an informal descriptive manner, using analytic models (Vernon and Holliday, 1985) and using the results of simulation studies (Archibald and Baer, 1986). The analytic models used differ from the model to be presented in this paper in that they are aimed at estimating performance of systems incorporating the modelled strategies, whereas our model is aimed at precise specification of strategies in general.

All of the above strategies have some points in common. Firstly, they all augment the usual valid/invalid state bit of a cache entry with additional status bits to reflect further attributes of a line (e.g. degree of sharing, whether the line has been modified, etc.). Secondly, they all require caches to monitor bus transactions, and possibly to change line state if they have a copy of the data being accessed by the transaction. Thirdly, they all assume special bus support for maintaining coherence. This support takes the form of additional transaction types beyond the usual read and write, such as invalidation, intervention and broadcast transactions.

Prior to the development of Futurebus, standard buses provided no protocol mechanisms for such things as notification of invalidation or intervention. Those proprietary buses developed to support particular cache coherence strategies only included the necessary mechanisms for their particular strategies. The goal of the Futurebus Committee was to provide protocol mechanisms to support all coherence strategies. This required investigation of the published strategies to determine the set of transactions required. As a result of this investigation, a preliminary model of coherent cache behaviour was formulated (Sweazey and Smith, 1986), and the basic bus protocol of IEEE Standard 896.1 was designed to support implementation of this model.

3. SPECIFICATION TECHNIQUES

The current work of the Cache Coherence Task Group of the P896.2 Working Group is to draft a specification of the behaviour of caches in a Futurebus system. The specification must ensure that any conforming implementation maintains coherence, and must be flexible enough to include published coherence strategies as subset implementations. A problem to which the Task Group has given much consideration is the selection of a specification language. This is a significant problem, since the solution has bearing on the effectiveness of the specification as a standard. It must combine precision and completeness with intelligibility by its intended audience.

The specification of cache coherence strategies has a number of parallels with the specification of programming language semantics. The same range of possible specifica-
A Behavioural Specification of Cache Coherence

Figure 2. Helix system organisation.

A common criticism of specification languages in general is that they themselves need to be specified formally, and this is also true of specification systems based on information structures. However, the two components of an information structure model can themselves be specified in precise terms, and are not easily understood by the large number of system designers who must read the specification. In addition, it is not always clear how to test a physical system for conformance to a specification written in such a notation.

A compromise between the above extremes is represented by the notion of an information structure model (Wegner, 1971). This kind of model is especially useful in the description of programming language semantics, where the state of a program is described by an information structure (essentially a data structure) and the semantics of a particular language feature is described by giving its effect on the information structure. For example, the information structure used by Basili (1975), in his description of the semantics of some language features for graph manipulation, is a collection of sets modelled as graphs. Similarly, Marlin and Oudshoorn (1985) use an information structure consisting of a collection of tables in their description of the data control aspect of programming languages.

An information structure model can also be used to describe the behaviour of coherent caches in a multiprocessor system. In this case, the state of each cache is represented by an information structure, and the changes in cache state are represented as transformations on the information structure. The contents of the information structure include the representation of attributes of lines of data (e.g. validity, sharing, etc.). The transformations represent the behaviour exhibited by a cache in response to transactions on the system bus, and to requests made by the corresponding cache client.

Using an information structure model as a specification has several advantages, particularly in the context of a standards document. Firstly, the information structure and the transformation operations can be expressed in a familiar “programming language” form. This makes it easier to produce the standard, and leads to a standard which is more intelligible to its intended audience.

Secondly, because of the programming language form, the specification of a system can be simulated. This requires an interpreter for the language, provision of some concrete representation of the information structure, and an environment for executing the transformation operations in response to some externally provided stimulus events.

Thirdly, as a result of a simulation, test vectors can be created, and subsequently used to verify conformance of an implementation to the specification. This is analogous to the use of validation suites to test programming language implementations for conformance to a semantic specification. The simulation can also be used as a reference implementation, being the arbiter in the case of disagreements between implementations.

A common criticism of specification languages in general is that they themselves need to be specified formally, and this is also true of specification systems based on information structures. However, the two components of an information structure model can themselves be specified in precise terms:

— The information structure can be described precisely using algebraic techniques for specifying abstract data types (Goguen, 1975; Goguen et al., 1977; Guttag, 1980 and Guttag et al., 1978); for illustrations of how this can be done see Friedel et al. (in preparation), Marlin and Oudshoorn (1985) and Oudshoorn and Marlin (in preparation).

— The transformations can be written in the notation of a programming language (either pre-existing or designed for the purpose) whose semantics can be specified formally using techniques such as denotational semantics (Tennent, 1976).

In this way the information structure model will use only primitives which have precise descriptions, thus ensuring that the model has firm foundations.

4. A BEHAVIOURAL SPECIFICATION USING HELIX

As part of the work for the IEEE P896.2 Cache Coherence Task Group, we have developed a behavioural specification of the Futurebus Cache Coherence protocol using the Helix simulation system (Silvar-Lisco Corporation, 1986). This system is part of a Computer Aided Engineering suite, and is a discrete event simulator specialised for modelling electronic systems.

The behavioural specification could be written using some other programming language, such as Pascal, Lisp, or a concrete form of such mathematical notations as temporal logic (e.g. Tempura (Moszkowski)). The requirement is that there be a formal semantic basis underlying the language used. Helix was chosen because its modelling language is specially designed for expressing behavioural interactions in complex electronic systems, and the runtime system provides most of the infrastructure required for managing a simulation and collecting results.

4.1 Overview of the Helix System

Figure 2 illustrates how the Helix simulation system is used. Graphical symbols representing components, and a schematic representing a circuit of interconnected symbols, are created using a graphics editor. A netlist extractor is used to determine the electrical connectivity drawn in the schematic, and to perform some validity checks on the circuit. For each component type to be simulated, a behavioural model is written using the Helix Hardware Description Language (HHDL). These models are then checked and compiled into an intermediate code. Next, the simulation linker is invoked to check for consistency between component symbols and models, and to create a simulator for the schematic.

The HHDL language is based on Pascal, augmented by constructs for representing component pins and concurrency constructs for implementing component actions in response to pin stimuli. An HHDL program firstly defines nettypes, which are (almost) arbitrary Pascal data types used to represent values passed on signal nets between component pins. Then, for each component type in the circuit, a comptype is defined. This consists of the specification of the pins, naming the nettypes they may connect to, some local state expressed in the form of local variables, a collection of subprocesses, and a main body for initialisation. The subprocesses are bodies of code which are activated when specified conditions occur; typically, the conditions are changes of values on input pins.

```hhdl
module flipflops;
const Tpd = 10;
nettype lognet = (Unk, Z, Lo, Hi);
comptype Dff;
inward D, CLK : lognet;
outward Q : lognet;
subprocess sample :
upon (CLK=Hi) and (recall(CLK)=Lo)
check CLK do
begin
    case D of
        Lo, Hi : assign D to Q delay Tpd;
        Unk, Z : assign Unk to Q delay Tpd;
        end;
    end; (* sample *)
begin (* Dff *)
    Q := Lo;
end; (* Dff *)
```

Figure 3. HHDL model for a D-type flip-flop.

Figure 3 is an example of HHDL code for a D-type flip-flop (called Dff), contained in a separately compiled module called flipflops. The net-type lognet defines the four-state type commonly used for logic simulation, with values for unknown, high impedance, low and high logic levels, respectively. Comptype Dff has two input pins, for data and clock, and a data output pin. Its initialisation body resets the output to the low logic level. The subprocess sample is sensitive to changes in the value on the CLK input (indicated by the phrase “check CLK”), and is activated when the new value is Hi and the previous value was Lo. When activated, a new value is passed onto the output pin, based on the current value of the data input pin.

The simulator created by the simulation linker is a program which contains an instance of a component model for each use of the component in the schematic. A run-time environment is provided which represents the signal nets connecting component instances, schedules updates of nets on a simulation timeline, and activates comptype instances at the required timepoints. For example, the assign statements in Figure 3 cause the run-time system to schedule an update of the net connected to the Q output of the flip-flop at a time point Tpd units after the current activation. The run-time system also collects a history of net updates when the program is run, and this history is used by formatting tools to create tabular or logic-analyser type displays of the simulation. Additional information about the behaviour of individual components in the circuit may be obtained by embedding trace write statements in the HHDL models. The simulator is controlled by a debugger-style command interface, providing single-stepping, breakpoints, and other similar facilities.
4.2 Specification of Futurebus Cache

The Helix system can be readily used to create an information structure model of cache coherence. This is done by defining a symbol for a cache, with pins to connect to a client and a Futurebus (see Figure 4), and specifying an HHDL comptype for the cache behaviour. The comptype contains local variables to implement the information structure representing the cache, and the transformation operations are encoded as comptype subprocesses.

One of the problems in designing any model is to determine which aspects of the system under consideration are actually relevant, and which can be ignored. In the case of modelling caches for the specification of coherence, the actual lines of data are not relevant; it is the line addresses and attributes that are important. For this reason, the information structure representing a cache does not contain any reference to the data which may be stored in a cache.

Another difficulty in model design is the temptation to incorporate details of some particular implementation. For example, a model might represent the cache storage as a small array of entries for storage of line tags and status bits. However, this approach would then require attention to the various cache design parameters (as mentioned in Smith, 1982) in order to specify cache behaviour. A better solution is to represent the attributes of each shared memory line, including the validity attribute.

Given the above two considerations, the information structure for a Futurebus coherent cache is implemented as shown in Figure 5. The store is represented by an array of status attributes, one entry for each line of shared memory data. The attribute valid indicates whether the line is stored in the cache's memory, shared indicates whether some other cache in the multiprocessor may also have a copy of the line, and owner indicates whether this cache is the owner of the line. Ownership means that the line has been updated without being written to shared memory, and implies an obligation to supply the data in response to bus read transactions, and to perform a copy-back or transfer of ownership at some stage. Note that lack of validity implies lack of the other two attributes.

The variable config is used to represent the diversity of possible coherence strategies which must be encompassed by the Futurebus coherence protocol. It contains a set of switches which are used at decision points in the transformation operations to govern how the information structure is modified. Following it are state variables used to communicate between the different HHDL subprocesses which define the transformations. The remaining variables are temporary storage used to implement the transformations.

The HHDL subprocesses which implement the transformations on the information structure fall into two groups: the snoop group, in which subprocesses are activated in response to Futurebus transactions, and the server group, in which subprocesses are activated in response to client requests. To illustrate the way in which the transformations are implemented, the snoop group will be discussed in some detail. The interested reader is referred to the draft P896.2 document (IEEE Draft Standard 896.2) for the protocol relating to service of client requests.

The snoop group contains three subprocesses, invoked on the address beat, each data beat and the end beat, respectively. The address beat subprocess, outlined in Figure 6, implements the cache lookup function, checking whether the cache has a hit at the address of the bus transaction. It is sensitive to changes in the address strobe input (AS_rx), and is activated when this pin changes from Lo to Hi, and the cache is not acting as a bus master. The address and command information placed on the bus by the bus master is accepted for use in this beat and the subsequent beats comprising the transaction.

Next, the cache arbitrates for exclusive use of the line of data being accessed by the bus transaction. The arbitration is required, as the cache client may asynchronously request access to the same line, possibly modifying its attributes.
Once access is gained, the cache determines whether it has a hit at the address of the bus transaction, by checking the valid attribute of the corresponding line. If there is no hit, then no further action is required for the transaction, and so exclusive access to the line is released, no Futurebus status signals are asserted, and the cache remains disconnected.

If the cache does detect a hit, its actions depend on the master’s command, the remaining attributes of the line, and the cache’s configuration settings. For example, in the case of a broadcast update transaction (IM and BC command bits set), a properly configured cache can connect as a selected slave to receive the new data. The remainder of the subprocess body sets information structure flags for use by the data beat and end beat sub-processes, and specifies the Futurebus status and handshaking for the address beat.

The subprocess which handles data beats is shown in Figure 7. It is sensitive to changes in the data strobe input (DS.rx), but is only activated when the cache is connected, either to accept broadcast data, or to participate as a third party. This latter case is shown in detail. The data beat command, consisting of a write signal (WR) and four byte-lane disables (LW, . . . LZ, c.f. byte-enables on other buses), is accepted from the master and used to control the reading or writing of data. Next the Futurebus status is determined, based on whether or not a line wrap has occurred, and then the data beat handshaking is performed.

The end beat subprocess, shown in Figure 8, specifies how a cache updates its line attributes at the end of a transaction. Again, only the detail for third party participation is shown. Where the cache is allowed to keep a copy of the line and intervened on a transaction initiated by a caching master (indicated by CC being true), the cache must include the shared attribute in the line’s status set. If the cache reflected, and shared memory was completely updated with the modified (dirty) data from the line, then, depending on the cache’s configuration, it modifies the line attributes accordingly. If the cache had detected a cache hit, exclusive access to the line is released. The last action of the cache is to complete the transaction handshaking.
A Behavioural Specification of Cache Coherence

subprocess do_end_beat :
  upon not master and (AS.rx = Hi) and (recall(AS.rx) = Lo)
  check AS.rx do
  begin (* do_end_beat *)
  if bus_hit then
  begin
  bus.disconnect_command := CH;
  with bus_addr_command do
  if connection_status = selected then
  begin (* end of broadcast update *)
  ...
  end
  else if connection_status in [reflecting, intervening] then
  (* end of third party participation *)
  if bus.disconnect_command.DER then
  begin (* do transaction back-out *)
  ...
  end
  else if bus.disconnect_command.DER then
  begin (* do transaction error recovery *)
  ...
  end
  else (* successful completion of transaction *)
  with store[bus.line.index] do
  begin (* do transaction error recovery *)
  ...
  end
  else (not master and (AS.rx = Hi) and (recall(AS.rx) = Lo))
  begin
  if keep.copy and not (CC and IH and not BC and read_occurred) then
  (* allowed to keep the line *)
  if connection_status = intervening then
  if CC then
  status := status + [shared]
  else (* connection_status = reflecting *)
  if all_dirty_reflected then
  if config.keep_after_reflect then
  if CC then
  status := status + [shared];
  if not config.keep_after_reflect then
  status := status + [owner]
  else (* not config.keep_after_reflect *)
  status := status + [owner]
  else (* not all_dirty_reflected *)
  if CC then
  status := status + [shared]
  else (* must invalidate *)
  status := status + [shared]
  else (* connection_status = unselected *)
  begin (* end of non-participation *)
  ...
  end
  bus.release_mutex;
end; (* if bus_hit *)
(* do handshake to complete disconnection beat *)
assign Lo to di.tx;
assign Z to di.tx;
assign Z to ah.tx;
end; (* do_end_beat *)

Figure 8. The end beat subprocess.

for the Futurebus interface. A circuit is created with nets linking the interface pins of a cache symbol instance to an instance of each of the driver symbols. The circuit is then linked to create a simulator. The purpose of the driver models is to activate the input pins of the cache model according to some predetermined command file or algorithm. They can also monitor the output pins of the cache model and report through the trace write mechanism. Using this approach, the behaviour of the cache under various driving conditions can be observed. For example, the client and Futurebus models could be programmed to stimulate the cache with transactions using the same cache line at the same simulation time point, in order to investigate the effects of collision of mutual exclusion requests.

5. CONCLUSIONS

In this paper, we have shown that an information structure model for describing the behaviour of coherent caches has several advantages over other modelling techniques, particularly where the model is to be used as a reference document. It combines an appropriate degree of precision and completeness, while remaining intelligible to an audience not expert in reading highly formal notations. In addition, a specification based on an information structure model expressed in programming language form can be executed to simulate the system being modelled. This simulation can be used as a reference implementation, and can provide test vectors for conformance validation. Where more formal specification is required, well-known semantic specification techniques for programming languages and data types can be applied.

The model we have described can be used for a number of purposes, all being investigated by the authors. Firstly, the IEEE P896.2 Cache Coherence Task Group is drafting a specification of the Futurebus cache coherence protocol based on the ideas presented here. Secondly, the previously published cache coherence strategies can be described in terms of this model, by defining a set of configuration constants for each strategy. Thirdly, a formal proof can be constructed to show that any cache system conforming to the specification does, in fact, maintain coherence. Fourthly, experiments can be performed to determine how various aspects of conforming coherent caches affect overall system performance. The outcomes of these investigations will add significantly to the understanding of cache behaviour in multiprocessor systems.

6. ACKNOWLEDGEMENTS

The authors are grateful for the helpful comments of one of the referees.

7. REFERENCES


A Behavioural Specification of Cache Coherence


IEEE Draft Standard 896.2, Firmware Protocols for Futurebus, IEEE.


BIOGRAPHICAL NOTES
Peter Ashenden is a Senior Research Officer with the Department of Computer Science at the University of Adelaide. He completed his Honours Degree in Computer Science in 1982 at the University of Adelaide and continued there as a Research Assistant when the Leopard Project was founded. His main areas of research interest are computer architecture and computer engineering, particularly related to multiprocessor architectures. He is also interested in computer-aided engineering for electronics design.

Chris Marlin has been with the Department of Computer Science at the University of Adelaide, where he is now a Senior Lecturer and Deputy Chairman, since 1984. He completed his Honours Degree in Computing Science in 1973 and his Ph.D. in Computing Science in 1979, both at the University of Adelaide. From January 1980 to December 1983, he was an Assistant Professor of Computer Science at the University of Iowa, Iowa City, Iowa (USA). His research has primarily been concerned with programming language design, specification and implementation, especially in relation to coroutines and parallel processes, and various aspects of integrated incremental programming environments.

First Announcement and Call for Papers

ISDN IN EUROPE

THEME: This International Conference will be oriented towards an examination of the impact of Integrated Services Digital Networks (ISDNs) on all aspects of computer communications in Europe. The 3-day conference is sponsored jointly by the International Council of Computer Communication (ICCC) and the International Federation for Information Processing (IFIP), and will be hosted by the Netherlands PTT.

ISDN IN EUROPE will be held at a time when both national transitional plans and European policies for the introduction of ISDN will have matured further. Thus, the Conference provides a strategic venue for users of international data communications and provides of value-added services and computer network facilities, whether public or private.

The Conference focus will be on developments which appear to be of special interest to professional users of the future integrated networks in Europe. The Conference language is English.

ORGANISATION: ISDN IN EUROPE is being planned by an Organising Committee, advised by an International Advisory Committee mainly composed of European members of IFIP Technical Committee on Data Communications (TC6) and European ICCC governors, and assisted by a Programme Committee. More information about the conference can be obtained from the Secretariat.

Papers must be submitted by 15 September 1988.

ISDN IN EUROPE Ms. Marijke Newman-van Alderen IBM Nederland NV, Johan Huizingalaan 265, 1006 AP Amsterdam, The Netherlands Teleph.: +31 20 513 35 61

A Distributed Operating System Kernel for a Closely-Coupled Multiprocessor

F.A. Vaughan†, C.D. Marlin† and C.J. Barter†

This paper describes the experience of the authors in developing a distributed operating system kernel for a closely-coupled multiprocessor workstation prototype. The strategy adopted was to obtain the distributed V-System from Stanford University and then to develop a kernel to permit this system to run on our architecture. Since the V-System was developed on a loosely-coupled architecture, namely a network of Sun workstations connected via Ethernet, the experience described here highlights the differences between distributed kernels required for the two kinds of multiprocessor architecture.

Keywords and Phrases: distributed operating system, closely-coupled multiprocessor, operating system kernel, V-System.

1. INTRODUCTION

The Leopard project at the University of Adelaide is concerned with the investigation of various hardware and software issues in the development of multiprocessor workstations. The project began in 1983 and its early stages focussed on the development of a monoprocessor prototype, the QDS-1000 (Knight, Ashenden, Marlin and Barter, 1985). Since the completion of the QDS-1000 prototype in 1985, efforts have been directed at the development of a multiprocessor workstation and associated software.

The software under development includes the distributed operating system discussed in this paper, and an integrated incremental programming environment called MultiView (Altmann, Hawke and Marlin, 1988). The eventual aim of the operating system component of the Leopard project is the investigation of the combination of concurrency in a multiprocessor system with the notion of persistence, as embodied in the Persistent Information Space Architecture (PISA) described by Atkinson, Morrison and Pratten (1986). At present, operating system development for the Leopard workstation is focussed on obtaining a distributed operating system kernel; this kernel can then be used as a foundation for PISA, as a suitable abstract machine on which to implement the programming environment, or for building a standard operating system interface (say, one with a UNIX*-like appearance).

Apart from the support for persistence in a concurrent environment, other goals of the work on the distributed operating system include support for fine-grained parallelism, self-configuration and distributed scheduling.

In order to facilitate the development of the operating system kernel, it was decided to develop the multiprocessor hardware in two stages:

— Leopard-1 is a three-processor prototype which has already been designed and fabricated; it uses NS32032 processors and a locally-designed interconnection bus called L-bus. This prototype was built specifically to allow earlier development of the operating system kernel.

— Leopard-2 is currently being designed; it will be a four-processor system and will employ NS32532 processors, a memory cache per processor and the emerging IEEE standard bus known as Futurebus. The operating system kernel will be moved to this prototype when it is available (sometime in early 1989).

The general architecture of all Leopard workstations is as shown in Figure 1, which is adapted from (Ashenden, Barter and Marlin, 1986). It is an example of a class of architectures known as symmetric multiprocessors.

The architecture is highly modular, and the flexibility thus afforded allows particular systems to be configured for a wide range of computing tasks.

![Figure 1. The Leopard workstation architecture.](image-url)

The main processing resource in a Leopard System is a pool of General Data Processors, each with its own local memory (consisting of 1 Mbyte in the case of the Leopard-1 prototype); this local memory is used to store the code for...
the operating system kernel and to hold some local data, thus reducing bus traffic. Other processing resources are provided by Special Data Processors and Device Processors; Special Data Processors are optional components optimised to perform particular processing services, such as array manipulation, signal processing and display list interpretation, whereas Device Processors are processors with special hardware for interfacing to outside world devices, such as disks, networks and displays. The Leopard architecture also allows for the inclusion of devices without an attached processor; communication with such Device Controllers is achieved through device registers in the conventional manner. The remaining module in the architecture is the Memory System, which provides a primary memory resource shared between processors and controllers. The means of interconnecting Processors, Device Controllers and the Memory System is a high speed parallel backplane bus (either L-Bus or Futurebus). For more information on the Leopard architecture, the interested reader is referred to (Ashenden, Barter and Marlin, 1986) and (Ashenden, Barter and Marlin, 1987).

Within a closely-coupled system such as the Leopard, a number of different kernel configurations are possible. Current commercial realisations of closely-coupled multiprocessors, such as the Sequent Balance (Freiland and Rodgers, 1984) and Symmetry (Manuel, 1987) systems, and the Encore Multimax (Anzelmo, Moore and Bell, 1985), are designed as cost-effective multi-user systems. They run a variant of UNIX and, to the user, are no different to a monoprocessor; the form of parallelism that they employ can be termed coarse-grained; since the tasks executed on processors are relatively large (such as the execution of the code for a user program). The performance of such systems is measured in terms of the number of users that can be supported. Typically, however, when a computationally intensive task is run, it will give a maximum performance equivalent to that of one of its processors. This is mainly because neither the application nor the kernel were designed to take advantage of the parallelism inherent in the architecture of the system. Parallelizing compilers and associated tools have begun to appear for commercial closely-coupled multiprocessors such as those above, but these aids are in their infancy as yet.

The Leopard is specifically intended to be a single-user workstation. We expect that demands on the processing capability of the machine will come from three different areas. The first area is related to interactive use; this comprises the display manager, display drivers, and interactive development software, where the latter may include CAD systems or integrated programming environments. The second area is that of occasional computationally intensive tasks; these would be run in the background and their presence should not impinge on the performance of the interactive tasks. Thirdly, there will be a number of low priority background tasks that spend most of their time inactive, such as mail daemons, performance monitors, and so on.

A naive approach would allocate different activities to different processors, with perhaps some degree of process migration to help even out the load. This suffers from the same problems as the coarse-grained approach discussed above. It is again unlikely that an activity could ever get more than a single processor's worth of speed or that an activity could naturally migrate out to efficiently use other processors if they became free. We wish to provide an environment where activities may be naturally decomposed into small cooperating tasks and be scheduled to take maximum advantage of the processing power available. Whereas the design of software systems to take advantage of the parallelism in such a system is difficult in its own right, we are primarily interested in providing a good support base. In view of our desire to support fine-grained parallelism to take advantage of the closely-coupled parallelism of the Leopard, we have been mindful of the cost of context switches (including switching virtual address spaces) and have attempted to reduce this cost as much as possible.

Initial work on the Leopard's operating system kernel focussed on the requirements placed on it by other software intended for the system (such as PISA and Multi-View). It rapidly became apparent that, whilst the more fundamental and machine-related aspects were within the scope of our limited resources, the task of writing all of the ancillary support routines from scratch would be sufficiently large to delay the project to an unacceptable degree. The V-System (Cheriton, 1984) provides a complete and well-tried operating system which is well suited to our needs. It uses a UNIX host as a server, thus eliminating the need (initially, at least) for disk device drivers; as a complete operating system, it also supplies all of the high-level services that we require for our future development work.

The Mach operating system (Accetta, Baron, Bolosky, Golub, Rashid, Tevanian and Young, 1986) also suits our needs well. The emphasis in the design of Mach has been to provide a comprehensive set of facilities, as opposed to the minimalist high efficiency approach taken in the design of the V kernel; this difference is particularly evident in the design of the message passing facilities in the two systems. We expect that the closely-coupled multiprocessor support required for the V kernel, which comprises the major task in the present work, should transfer in large part to a later Mach implementation.

Other distributed operating systems are described in (Tanenbaum and van Renesse, 1985). Apart from the V-System, the other major operating systems described are the Cambridge Distributed Computing System (Needham and Herbert, 1982), Amoeba (Mullender and Tanenbaum, 1986) and Eden (Almes, Black, Lazowska and Noe, 1985); these have structures that are, in our estimation, unsuited to (or opposed to) a closely-coupled hardware architecture.

2. THE V KERNEL

The V-System is intended for a network of workstations in a configuration of the form shown in Figure 2, which is adapted from (Cheriton, 1984). It is a distributed operating system which runs on diskless nodes, relying on UNIX-
based file servers for mass storage; communication between the various components of the configuration is via Ethernet (perhaps with Gateway Server Machines connecting more than one Ethernet network). Instead of being totally separate nodes, each running their own instance of the operating system, each node is intimately related to all the others. In many respects, they can be seen as sharing parts of the same instance of the operating system. The V-System supports communication between tasks in such a way that the node on which a particular process resides is transparent. It also supports process migration from one node to another, with the restriction that an entire virtual address space and all tasks within it are migrated at once; furthermore, they must migrate to a similar machine (that is, one with the same processor type and which has at least as much physical memory).

Central to the design of the V kernel is its message passing. The nature of this message passing has been developed by its designers from previous designs, namely Thoth (Cheriton, Malcolm, Melen and Sager, 1979) and Verex. Rather than providing a completely general message passing protocol, they have attempted to cover those needs actually found in practice and optimise the implementation accordingly. One of the more important aspects of the V kernel is the use of the same message passing protocol throughout the system. This is analogous to a common calling standard for a given architecture and reaps similar rewards. All system requests are handled by sending a message to the kernel, which is represented as a known server task. Message passing is also used for all synchronisation within the system.

The message passing protocol, which is depicted in Figure 3, is a mixture of synchronous and asynchronous elements. A message is a small, fixed length block (of 32 bytes). By itself, a message can be used for synchronisation, signalling or the transfer of very small amounts of information. A sender, such as the one in Figure 3, sends to a known receiver and blocks on the send. The receiver will receive the message when it issues a receive call; should it request a receive without a message being available, it will block until the message is ready, as shown in Figure 3. When the receiver has received the message, it must reply to the send. Until this is done the sender will remain blocked. During the time between the send being received and the reply being sent, the receiver can make use of the fact that the sender is blocked. In particular, a message may contain a segment descriptor that describes an arbitrary piece of contiguous memory. The receiver may access this memory with the V kernel routines CopyTo and CopyFrom. A requirement of using these requests is that they can only be used between tasks in the situation depicted in Figure 3—that is, when the sender is blocked awaiting a reply from the receiver. In this way, arbitrary amounts of information may be passed between processes. It is transparent as to whether the communicating tasks are on the same node or on different nodes. Specialisation of the message passing protocol described above are also provided, such as ReceiveReply which saves the expense of two calls when simple message transfers are being done. Message passing communication with servers is often embedded in routine stubs, so that the caller sees an ordinary procedure call.

Two important concepts within the V kernel are tasks and teams. A task may be thought of as a particular thread of control in an activity. Tasks are grouped together into teams, where the members of a team all share the same virtual address space; this shared virtual address space is known as a team space. A task is a very lightweight object, in the sense that context switches are very fast between members of the same team (since there is no need to change virtual address translation tables). Tasks are also used as general purpose building blocks. For instance, the V kernel does not provide for any queued communication, but it is trivial to make the receiver of a message a task that manages a queue and then passes the messages onto another task upon request. A general tenet of the V kernel's design is that whenever concurrency is needed, this task structure is used, and another is that message passing is used for all communication.

The V-System distribution also includes VGTS, the Virtual Graphics Terminal System (Nowicki, 1985), a device-independent graphics system with some special properties. It has been designed to act as a graphics server, serving client processes anywhere on a network running the V-System. To make this possible, great care has been used in the design of a suitable communications protocol; in fact, VGTS uses graphical objects that have many of the characteristics of structured display files, but respond as objects to messages. Each object is instantiated on the server and graphical editing tasks run on the display device (using algorithms suited to the device) and the object communicates the result of the transaction to the client. In this way, the communications cost and latency are greatly reduced and it becomes very easy to provide a single

![Figure 2. A typical V-System configuration.](image)

![Figure 3. The message passing protocol for the V kernel.](image)
package that takes advantage of different methods and degrees of sophistication in hardware.

3. KERNEL SUPPORT FOR THE V-SYSTEM ON THE LEOPARD WORKSTATION
In replacing the V kernel by a suitable kernel to support the V-System on the Leopard workstation, some aspects of the kernel become simpler and others become more complex. The differences revolve around the introduction of shared memory. Maintaining distributed state information becomes trivial, but problems inherent in concurrent access to the same structure arise. Leopard workstations will themselves be members of loosely-coupled systems (that is, workstations in configurations like that in Figure 2) and hence maintaining the advantages already inherent in using the V-System in a distributed environment is essential.

As mentioned earlier, one of our goals is to have a self-configuring system. We wanted code that would run on any configuration of Leopard hardware, from a single processor with only its local memory (which is essentially the same as a node in a V-System configuration) to a multiprocessor Leopard with an arbitrary number of processors and an arbitrary amount of global memory. This goal was motivated partly by the elegance of such a system, and partly by a pragmatic need: in the continuing development of hardware and software for the Leopard, we cannot afford the time, space or effort required for multiple versions.

A simple approach to implementing a kernel for the V-System on a closely-coupled multiprocessor like the Leopard might well be to configure it to look like a set of loosely-coupled processors, each running in local memory and only using the global memory as a buffer for messages between processors. Whilst fulfilling the goal of having the V-System running on the Leopard, this approach loses many of the advantages one would hope to gain from a closely-coupled multiprocessor, such as true concurrent execution within a team and the sharing of data structures. What is required is to expand the functionality of a single node in the loosely-coupled system, using many of the existing concepts of the V kernel. A Leopard multiprocessor is to be regarded as a node in a loosely-coupled distributed system; precisely the same restrictions on internodal communications apply as for a monoprocessor node. Within a multiprocessor node, the restrictions in the V kernel arising from disjoint memory systems can be avoided, as can the overhead of communication between processors.

As a result, three levels of kernel can be distinguished:

- a kernel instance residing in a single processor of a multiprocessor node,
- a multi-threaded kernel which is distributed over a multiprocessor node, and
- the distributed kernel executing over a loosely-coupled collection of nodes.

Since the local memory on each Leopard processor is used as a local store for each kernel instance and all user processes reside in the global memory, the state of a multi-threaded kernel is represented by the contents of the global shared data structures, and the state of a kernel instance is represented by data structures resident in the local memory of the corresponding processor. This organisation allows a team to be spread across processors, with each processor potentially running different tasks within the team simultaneously, giving the fine-grained parallelism desired.

The V-System has previously been implemented on VAX and Motorola MC68000-based machines. As mentioned in Section 1, the Leopard workstation is based on the National Semiconductor NS32000 processor family. The NS32000 is very similar to the VAX for general coding purposes (say, in terms of instruction set), but the architecture for handling the hardware interface (such as interrupts and memory management) and the provision of some additional features (such as support for modules) makes the NS32000 significantly different overall.

Implementations of the V kernel on the VAX and MC68000 architectures have made use of a very fast technique for passing messages. A message, which is (as mentioned in Section 2) only 32 bytes long, will fit entirely within the register set of these machines, and is left there when a message is sent. The task most likely to be chosen to run next is the task currently blocked awaiting that message. When activated, it will find the message already in the registers. If the next task to run is not the recipient, the scheduler can place the message in the appropriate memory-based buffer at a cost which is no different to that of message passed via memory. In a closely-coupled multiprocessor, particularly one employing memory caches (as will be the case with the Leopard-2), this technique is not useful, since there is no guarantee that the receiver task will be run on the same processor as the sender. In such a situation, the processor registers cannot be used as a common message buffer and some additional mechanism must be provided to extract the message from the first processor's registers for the recipient. Also, the NS32000 register set cannot hold both enough context and the message, preventing the use of the method employed in these implementations of the V kernel. However, the cache coherency system provides the mechanism for message transfer: even when the receiver is running on the same processor, a copy of the message will reside in the cache. Whilst not as fast as a register copy, it is still considerably faster than a physical memory copy. Cache coherency logic on the Leopard-2 processors will ensure that the correct copy of the message buffer is accessed by the receiver independent of the processor on which it is executing. The cache line size of the Leopard-2 is 64 bytes; this means that a V kernel message will fit entirely within one cache line, along with some housekeeping context. Transfers of cache lines take place as single bus transactions, leading to significant gains in speed; thus, all messages are aligned on 64 byte boundaries. For these reasons, we do not use the register-based technique for fast message passing and rely instead on message buffers in cache memory.

The V kernel has a special protocol, called the Inter-
Kernel Communications (IKC) protocol, for communication between kernels running on separate nodes. The extra protocol is used to cope with the unreliable nature of the Ethernet connection between nodes and to hide the need for multiple packets for long transfers. At the coding level, this protocol is invisible and all message transfers look identical. However, in a closely-coupled multiprocessor system, the inter-kernel communication becomes more complex. Inter-kernel communications for the Leopard kernel is split into two parts: one is the current LAN-based part for communication with other nodes, and the second part is for communication between processors within the same node. Because tasks are allowed to migrate between processors on the system, it is not possible to regard a task as being under the control of a given instance of the kernel that can take care of its communication needs. Therefore, we regard the message passing system on a multiprocessor as a single multi-threaded manager, not as a set of cooperating tasks, one on each processor. The IKC protocol is still necessary for communication between nodes; however, it is no longer regarded as inter-kernel, but rather inter-nodal. For communication where the kernel instances need to notify one another of specific events, in a manner analogous to hardware interrupts, the Leopard's local attention register can be used to asynchronously notify the kernel instance of a waiting message; it is not expected that this facility will be used extensively.

Probably the most important function of an operating system kernel is that of resource management. In fact, the kernel may be regarded as little more than a resource manager. In a closely-coupled multiprocessor system, resource management may be divided into the management of memory and processors. The scheduler is the processor resource manager and is discussed later in this section.

Within the local memory of each processor, each kernel instance maintains a memory manager for its own purposes. A global memory manager must also exist for global physical memory; this manager is accessed by each kernel instance as needed. To increase parallelism, each kernel instance maintains a small pool of free pages (say, 1 or 2% of the global free list), from which it is able to allocate physical pages to requesting tasks without the cost of locking the global memory allocation data structures. Global locking is only needed when a local pool is depleted.

For a particular team, the team-wide free list must be locked during allocation of virtual memory for tasks. Due to the small size of tasks, and the potentially large number of them, it is not feasible to maintain a pool for each task. Should serialization become a problem in the team memory allocator, it is possible to create a small number of pools that can be individually locked by individual processors for each task. When a demand-paged virtual memory system is used, pool managers become more important; if the manager can keep memory in large contiguous chunks, memory allocated close together in time will occupy close virtual addresses, yielding important gains in locality during page faults. When the kernel updates the virtual memory configuration of a team, care must be taken in updating Page Table Entries (PTEs). Potentially, other processors could be executing tasks in that team and may have cached the relevant PTEs in their memory management chip. The hardware does not provide any mechanism for maintaining coherence in this case; it is left to the software to maintain. Each Leopard processor can be interrupted by writing to its local attention register. Should other processors be executing in the same team space whilst updating the PTEs, they can be interrupted and the interrupt routines can invalidate the cached PTE.

In fact, the current distribution of the V kernel does not support demand-paged virtual memory. In an experimental system such as this, with sufficient physical memory, this does not represent a great problem. The V kernel does provide a simple mechanism to allow tasks in different teams to access the same physical memory. This allows for simple sharing of code images and for very fast CopyTo and CopyFrom calls to be implemented with page switching. In the longer term, we would wish for more sophisticated virtual memory management, providing facilities such as direct-mapped files, as is provided for example in VAX/VMS (Digital Equipment Corporation, 1985) and Mach.

The performance and correctness of the system lock manager is critical to the success of a shared memory multiprocessor implementation of a distributed operating system kernel such as this. The most basic kind of locking mechanism is known as a spinlock and amounts to a busy wait on a flag. Normally frowned upon in implementations of operating systems, the spinlock is the only way to ensure correct access protocols are maintained for multiprocessor scheduler structures. The Sequent Balance 8000 uses special hardware to prevent a spinlock placing an undue burden on the system bus. The cache design of the Leopard-2 allows the use of the “test-and-test-and-set” method described in (Rudolf and Segall, 1984). In their method, a loop tests for a lock to be cleared, but rather than using an interlocked operation which always accesses the bus, it waits using a normal test that only accesses the cached version of the lock. The cache coherency logic detects the lock being cleared by the current owner processor and updates the local copy. Then, an interlocked test-and-set operation is used to gain the lock. Thus, the potential bus loading is eliminated. The support of multiple concurrent access to data requires the provision of counting semaphores as a high-level construct.

Ultimately, the effectiveness of a multiprocessor operating system such as ours will depend on the quality of the scheduler. It must make the best use of the processors, minimise overhead in context switches, and reduce unnecessary duplication or movement of data. The closely-coupled shared memory architecture of the Leopard opens up some new opportunities and some potential pitfalls with regard to scheduling. When a kernel instance is rescheduling, any task in the same team must be considered to be a preferred candidate for execution, since it will not be necessary to invalidate the virtual memory translation table caches. It is also probable that they may share some
global data structures that may be partially cached on that processor. It is also reasonable to run a task on a processor that has executed it a short time before; even though another task has intervened, there may be substantial data still resident in the cache (the Leopard caches physical rather than virtual addresses). We use either a simple form of cache ageing (employing a fixed time measure) or the heuristic of preferring a task if it was the task executed before the one being removed. Alternatively, on a lightly loaded machine, better performance could be obtained by leaving two cooperating tasks on separate processors, in an attempt to get maximum execution parallelism. This results in processors being left idle for short periods of time while message transfers occur. It is then necessary to make the kernel's idle process a pseudo-process (essentially a part of the scheduler) which does not affect any of the idle processor's context. In particular, it must not cause the page translation caches or system cache entries to be invalidated. As system load increases the scheduling heuristics will change.

As mentioned in Section 1, one of the goals of this project is to have a completely distributed scheduler, rather than having a single processor responsible for maintaining control. This places some severe constraints on the design of the scheduler. A great deal of care must be taken to prevent the scheduler itself from deadlocking. Like all system resources, the scheduler's data structures must be accessed via locks. In its current form, the scheduler is treated as a single resource; access is serialised, with only one kernel instance having access to any of the scheduler data structures until it has completed its scheduling task. Since this represents a serialisation of the entire system, care is taken to make the actual time spent within the locks as short as possible. The scheduler is intimately dependent upon the message manager and much of the work required to determine rescheduling priorities can be done within the message manager at non-critical times. As with other parts of the kernel, the scheduler is designed to auto-configure. At boot time, each processor in turn builds the required structures to allow scheduling on itself. In this way, the same kernel code can be used for different configurations of machines.

As illustrated in Figure 1, the Leopard architecture uses separate device processors for device control, freeing the general data processors entirely from this task. This design is also intended to allow the system to run almost without hardware interrupts. On the Leopard-1, however, we have a mixture of device configurations. This allows us to test different software configurations preparatory to the implementation of the V-System on the Leopard-2. The Leopard-1 processor is more akin to the device processor on the Leopard-2, since it must manage its own communications. The Leopard-1 will also be equipped with a graphics subsystem (Fang, 1987; this graphics subsystem has no processing capability of its own (except for bitblt and similar graphics functions) and so a processor/graphics pair can be regarded as a device processor for software design purposes. Currently, work is under way to write a VGTS driver for the graphics subsystem. Since the frame buffer for the graphics subsystem is mapped into global memory, it is accessible by any processor. It is therefore possible to have concurrent graphics drivers running. The global lock manager described above can be used to limit access to the frame buffer, but in the longer term much more sophisticated management is possible and desirable, since operations on non-intersecting areas of the buffer may then be carried out in parallel. The design of VGTS allows us to easily achieve concurrent access and so use the Leopard-1 to produce a prototype of a multiprocessor dedicated graphics subsystem.

4. CONCLUSIONS
The experience of the authors in developing a distributed operating system kernel for the Leopard workstation, a closely-coupled symmetric multiprocessor, has been described. This kernel is initially intended to support the V-System, but we also plan to support the Mach operating system with the same kernel, or a closely related one. Instead of being able to simply port the existing V kernel, which was intended for a loosely-coupled network of workstations, it was necessary to write a largely new kernel based on the V kernel. The Leopard kernel extends the V-System to allow it to execute on a network which includes closely-coupled multiprocessor workstations, in addition to the existing ability to run on single processor workstations.

Of particular interest is the way in which the memory cache in the Leopard architecture can be exploited to good effect, as demonstrated by the scheme to pass messages without using registers and the method for lock management. The notion of multi-threaded managers also appears to be one which is likely to be generally useful in operating systems kernels for closely-coupled multiprocessors with shared memory.

Once the V-System is functioning properly on the Leopard, it will be used as a foundation for the construction of higher level software systems. On the one hand, it will be used as the basis for the implementation of PISA on the Leopard, a first step in the construction of a concurrent persistent operating system. On the other hand, the V-System also provides a suitable base (especially via VGTS) for the implementation of the MultiView integrated programming environment on the Leopard workstation.

5. ACKNOWLEDGEMENTS
Since the development of the distributed kernel described in this paper is part of a larger project, the authors wish to acknowledge the contributions of many others involved in this project. Of particular note are the contributions of Peter Ashenden, Andrew Cagney, Peter Cassidy and Mike Petty.

This work was supported in part by grants from the Australian Telecommunications and Electronics Research Board, the CSIRO Collaborative Programme in Information Technology, and the University of Adelaide.

6. REFERENCES
ACSC-12

CALL FOR PAPERS

Wollongong, 8-10 February 1989

The Twelfth Australian Computer Science Conference will be held at the University of Wollongong, Wollongong NSW during the period 8-10 February 1989.

The conference will follow the usual format. The purpose is to promote interaction between computer scientists working in academic and other institutions around Australia. New members of the profession and graduate students are welcome to submit papers and attend the conference. Student accommodation is available and the registration cost will be kept to a minimum.

The conference will be preceded by a Computers in Education Conference sponsored by Apple in the various Pacific rim countries in conjunction with the various Apple consortia. There will be opportunities for interaction between delegates to the two conferences, and some people may wish to attend both.

Papers are sought for ACSC-12 which present the results of new research or survey current research in any area of Computing Science. Authors should follow the guidelines for papers in the Australian Computer Journal and papers should not exceed 5000 words in length.

Initial versions of papers are needed for refereeing by 19 September 1988. Authors of accepted papers will be notified by 7 November 1988, and final versions of accepted papers will be needed by 5 December 1988.

Three (3) copies of the paper are required for refereeing.

Papers should be sent to:

ACSC-12 Papers Chairman, Department of Computing Science
University of Wollongong, PO Box 1144, Wollongong NSW 2500

Further information can be obtained from the above address, or from acsc@uowcsa.oz (ACSnet) or telephone (042) 27 0859.

BIOGRAPHICAL NOTES

Francis Vaughan received a BSc in Computer Science at the University of Adelaide in 1982 and an Honours degree in Computer Science in 1983. He is presently a Senior Programmer in the Department of Computer Science at the University of Adelaide, where he has been involved in the Leopard multiprocessor workstation project for about five years. His research interests include distributed operating systems, and inheritance mechanisms in programming languages.

Chris Barter is Professor and Chairman in the Department of Computer Science at the University of Adelaide, a position he has held since 1984. He received his PhD in Computer Science from the University of New South Wales in 1971.
An Integrated Programming Environment Based on Multiple Concurrent Views

R.A. Altmann†, A.N. Hawke† and C.D. Marlin†

Traditional programming environments typically provide a collection of loosely-connected, separate programme development tools such as text editors, compilers, linkers and debugging aids. Integrated programming environments combine the functions of these separate tools into a single program development system, presenting the user with a single interface to the system. This paper describes progress on an integrated programming environment, called MultiView, which presents the user with a variety of views of the program under development, each of these views being automatically updated when changes are made to the program using any other view. In order to improve performance and to ensure that all displayed views are up to date, MultiView has been implemented as a collection of parallel processes communicating via message passing; MultiView is expected to be one of the first applications for the Leopard multiprocessor workstation, also under development at the University of Adelaide.

Keywords and Phrases: integrated programming environments, multiple concurrent views, graphical editing, distributed programming, message-passing systems, language-specific editors, syntax-directed editing, language-sensitive editing, language-oriented editing, structure editing, Modula-2.


1. INTRODUCTION

The kind of programming environment used by most programmers consists of a collection of separate tools which have to be repeatedly invoked during program development. Typical of these tools are text editors, compilers, linkers and symbolic debuggers. Developing a program under these circumstances then becomes a process of invoking these tools a number of times and making sure that the results of one step correctly flow into the next step (for example, that the object code file produced by the compiler is used as input for the linker, and that the result of this step is fed to an execution supervisor).

Recent research has led to the concept of an integrated programming environment as a way to better support the programming activity. These environments are integrated in the sense that the distinctions between the separate tools discussed above disappears, although the various translations and other tasks must still be performed. The interested reader is referred to (SIGPLAN and SIGSOFT, 1984), (SIGPLAN, 1985) and (SIGPLAN and SIGSOFT, 1987), for various examples of integrated programming environments.

The integration provided by such programming environments greatly facilitates the programmer’s task, as does the fact that the translation tasks mentioned above are typically performed incrementally (that is, as the program is entered or modified). However, most integrated programming environments provide access to the program via a single kind of representation. This representation is usually textual, as in the case of the Cornell Program Synthesiser (Teitelbaum and Reps, 1981) and Gandalf (Notkin, 1985). With the advent of cheaper workstations with high-resolution displays (such as the Sun and Apollo workstations), some experimental systems have been developed which make use of graphical depictions of programs; for examples of such systems, see (Albizuri-Romero, 1984) and (Raeder, 1985). However, once again, the programmer is usually provided with only one kind of program representation.

The principal motivation for the development of the MultiView programming environment is the observation that programmers tend to make use of various program representations during program development. These representations cover both the program specification, known as the static program, and the program in execution, which is referred to as the dynamic program. Typical program representations include:

— textual representations of the static program, such as “source listings” and text editor displays,
— tree representations of the static program, such as parse trees and trees depicting the nesting of subprogram definitions,
— other diagrammatic representations of the static program, such as flow-charts and Nassi-Schneiderman diagrams, and
— various representations of the dynamic program, such as stacks of subprogram instances and graph structures representing data structures composed of dynamic variables linked by pointers.

Copyright © 1988, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ’s copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

† Department of Computer Science, University of Adelaide, GPO Box 498, Adelaide, SA, 5001. This paper was presented at the Eleventh Australian Computer Science Conference at the University of Queensland in Brisbane, Queensland in February 1988.

It is also likely that programmers would find other program representations useful if they were generated automatically.

In view of this tendency to use multiple program representations, the MultiView programming environment aims to facilitate programming by presenting the user with a variety of views of the program under development. These views can be used to edit (i.e., modify) the program, rather than merely being visual representations to assist in the understanding of the program and its behaviour. Consistent with the above observation about programmers' use of multiple representations, some views are planned to present information textually, but others contain graphical representations of the program. For a given object, say the static program, the user may choose to display several representations of the object. These multiple views of the object are all updated simultaneously whenever one of the views is used to make changes to the object. Among existing integrated programming environments, only PECAN (Reiss, 1984) appears to include a similar notion of multiple concurrent views; however, in the case of PECAN, only a limited range of views permit editing and there appears to be no true concurrency of the updating of the various views.

As shown in Figure 1, the MultiView implementation is a collection of parallel processes communicating via message-passing. At the heart of the implementation is the program database, containing a complete description of the program under development; this database is controlled by the Database Process shown near the centre of the figure. Each view instance is managed by a View Process. Parallelism occurs within such an implementation in two major ways: between the handling of the various views of the program and between the incremental translation tasks. The shared program database and the associated Database Process are clearly crucial to the operating of the MultiView implementation and so a considerable part of the effort so far has been devoted to the development of this aspect of the MultiView implementation. This development has included the design of a uniform protocol which is used between all View Processes and the Database Process.

This paper describes the progress which has so far been made towards the goal described above. The present MultiView prototype is a partial programming environment for Modula-2, is written in Modula-2 and C, and runs on a network of Sun workstations. Section 2 of this paper describes the nature of the representation used for the program database and Section 3 goes on to discuss the interface between the Database Process and View Processes. Section 4 briefly describes one view which has been developed, a graphical view called Kookaburra; in Section 5, we describe another view, this time a view called Eucalypt which enables the user to interact with the program database. Finally, Section 6 presents some conclusions and discusses our future plans.

2. THE PROGRAM DATABASE

As described above, the program database plays a central role in the MultiView programming environment and interacts with the other components via message passing. The program representation used in the database is an abstract syntax tree. This is the form of program representation most commonly used by integrated programming environments and language-specific editors; other forms of program representation are possible and the interested reader is referred to (Marlin, 1986b) for a brief discussion of them. An abstract syntax tree describes the syntactic structure of a program in a manner which is convenient for the editing of programs. The particular form of abstract syntax used in MultiView was inspired by the approach used in Mentor (Donzeau-Gouge, Kahn, Lang, Mélèse and Morcos, 1983). In this form of abstract syntax, each node has a kind (or phylum), which defines the set of possible language constructs (or operators) that the node may take on as its value. At a given stage in the development of a program, it is to be expected that some nodes in the tree (which will, in fact, be leaf nodes) will be unexpanded, whereas others will have been expanded into a particular operator.

Since Modula-2 is a language supporting multiple compilation units, the program database must accommodate several abstract syntax trees. The internal structure of the database consists of a binary tree of directory nodes that are pointers to records. Each directory node contains information concerning the abstract syntax tree which it manages; this information includes the name of the file corresponding to that compilation unit, the largest node number in the tree, a pointer to the root of the tree, and a hash table keyed on node numbers. The binary tree is sorted alphabetically on the filenames corresponding to the abstract syntax trees.

The Database Process has a handle for each View Process in contact with it. This handle consists of a pointer to a directory entry managing a tree, a pointer to a private clipboard, and a pointer to the current node in this tree.

Each node in an abstract syntax tree has a value, links to other nodes, a status, a count field (recording the number of views with handles pointing to that node or its children), and the node number. The node's value consists of a phylum, an operator and a string identifying the node (the phylum name if unexpanded, the operator name if expanded, or the value if this is a literal). Nodes are connected right, left, up to the parent and down to the first

![Figure 1. The structure of the MultiView implementation.](image-url)
child. The status of a node is one of “first”, “last”, “middle”, “unique” or “deleted”. The first two of these are used for nodes which are the first or last children of some node, respectively; the status “middle” is used for those nodes which are neither the first nor the last child, and “unique” means that the node is an only child.

When the current node part of a handle moves to a node, the count field of the node is incremented. If the handle moves left, right or up from a node, the node’s count field is decremented. Hence, the criteria for disposing of a node is that the status must be “deleted” and the count field must contain zero.

The Database Process is written in a combination of C and Modula-2. The parts written in C make up the interface between Modula-2 and the SunView* window management primitives (Sun Microsystems, 1986), and the UNIX** socket primitives which enable message passing between processes. The executing program is under the control of the SunView Notifier. Whenever a new view requests access, it is assigned a unique socket number (that is, a data channel) which the Notifier is told about. The Notifier waits for messages coming in on the various sockets and calls a procedure, with the appropriate arguments, to read the message and perform the necessary actions. The Notifier also manages the buttons, menus, panels, and redrawing and painting of the display.

3. THE INTERFACE BETWEEN THE DATABASE PROCESS AND VIEW PROCESSES

When the Database Process receives a request from a View Process, say as a result of a user employing that view instance to make some change to a compilation unit, it will first check if the view’s handle is pointing to a deleted node. This is a node that has been deleted as a result of another view requesting an action which led to the deletion of the node; this is possible because there may be more than one view in a given tree at one time. If this is the case, then the handle is moved up the tree to the first node that is not deleted and an error message and the new node number are sent in reply to the view’s request. At this point, a routine is also called to see if it is now possible to dispose of the deleted nodes.

If the Database Process accepts the request as legal, it makes the changes and broadcasts notification to all View Processes and to the processes (not yet implemented) which perform incremental semantic analysis and code generation. These processes will then make the consequent changes to their internal data structures, the displayed program representations, the symbol table and the object code for the program, respectively.

Communication between View Processes and the Database Process is via message passing using UNIX sockets. The details of how this occurs is hidden from someone playing program representations, the symbol table and the database. If the latter option is requested, the supplied procedure “dump” is called to cause the Database Process to send the View Process a description of the abstract syntax tree for that compilation unit.

It is worth noting at this point that View Processes will all construct their own cache copy of the abstract syntax tree. This is done both to reduce message traffic between View Process and the Database Process, and because a View Process will require some local information to be associated with the abstract syntax tree. Typical of what needs to be recorded locally is information about the screen coordinates of the depictions of the nodes in the tree.

As mentioned previously, when a View Process requests a change in the program under development, a message is sent by the Database Process to all the other View Processes, informing them of the modification. However, these View Processes are not necessarily “busy waiting” for communication from the Database Process: two procedures are provided to enable a View Process to switch communication interrupts on and off. The procedure “async” turns on interrupts and specifies a procedure that will read the incoming message from the Database Process and take appropriate action. When the View Process initiates a conversation, it turns off interrupts by calling the “sync” procedure and, after completing the predefined sequence of sends and reads, it calls “async” to turn interrupts back on again.

Each node in an abstract syntax tree for a compilation unit has a unique node number. These numbers are used by the View Processes to determine what parts of their local database has been affected by a change notified by the Database Process. Thus, there must be a one-to-one correspondence between the node numbers used in the central database and the node numbers used in other views. The commands provided for communication between a View Process and the Database Process can be divided into five main categories:

1. commands that move round the tree (such as “GotoNode”),
2. commands that change the tree (such as “InsertDown” and “Delete”),
3. clipboard-related commands (such as “PasteTree” and “ClipTree”),
4. commands which are concerned with files (such as “SaveModFile”, which saves a tree as Modula-2 source text), and
5. inquiry commands (such as “ReadNode”).

Two commands that merit special mention are “Start-Batch” and “End-Batch”. When a view makes an extensive set of changes to an abstract syntax tree (such as inserting a large structure), it begins by sending a “StartBatch” command to the Database Process which is relayed to all the other views. The other views may then choose not to update their own databases and/or window displays until
the corresponding “EndBatch” command is received, thus being able to make all the changes at once.

A variety of error types may be returned by the Database Process on receiving a request and attempting to obey it, including “Success”, “UnrecogCommand” and “ASTDNE” (where the last of these stands for “abstract syntax tree does not exist”).

4. KOOKABURRA: A GRAPHICAL TREE-ORIENTED VIEW

Kookaburra is an example of a view for the MultiView programming environment. It is a graphical view which can be used to edit the abstract syntax tree directly; that is, a Kookaburra view displays the abstract syntax tree for a compilation unit. It is intended that Kookaburra be used in conjunction with other editors, such as a textually-oriented editor, rather than forcing a user to employ it to perform all editing. It is likely that Kookaburra will be most useful in clearly displaying the structure of the program, which may be difficult to discern in another view. For example, the structure of an expression may be better illustrated in the form of a tree than as a line of text, since the tree can show the order of operators more clearly. Because of space limitations, only a brief introduction to Kookaburra can be given here; a more detailed account can be found in (Altmann, 1986).

A typical Kookaburra view is shown in Figure 2. The large panel occupying most of the view displays the abstract syntax tree in graphical form. The tree may, of course, be too large to fit in the panel (as it is in Figure 3, for example) and so a “panning control” is provided in the top left hand corner of the view; this control permits the panel to be panned over the tree structure. The remainder of the top of the view is occupied by a row of buttons, the first four of which control which of the nodes in the tree is regarded as the current node in obvious ways; the “Next” button finds the next unexpanded phylum. The button called “Center” can be used to reposition the lower panel so that the current node is in the centre of the panel. The “Clipbrd” button displays the current contents of the clipboard, “New” can be used to change the compilation unit being edited, and “Quit” and “Help” are self-explanatory.

Kookaburra is a template-based editor, meaning that the user must create a program by gradually expanding the program in a top-down fashion. As explained earlier, each node has a phylum representing the set of legal operators which may be given as the value of that node; there will be a template for each of these possibilities, giving its structure according to the abstract syntax of the language. In Figure 2, for example, the fact that the node “Statement” is displayed in reverse video indicates that it is the current node; pointing at this node and pressing the right hand mouse button displays a menu of the operators which are valid for this phylum. Of these possibilities, the operator “ProcedureCall” is currently being selected. The template for another of the operators in this list, that for “Assign” (an assignment statement), is shown as the first child of the parent of the “Statement” node. The operators discussed so far all have a fixed number of children; there is another type of menu for those nodes with a variable number of children, but this will not be discussed here.

If a node is unexpanded, selecting a particular operator expands the node by defining some children for it; if the node was already expanded, the selection of an operator replaces the present children. Thus, through the use of the menus provided by Kookaburra, the programmer is restricted to only selecting operators which will lead to a syntactically correct program. The only components of the program entered via the keyboard are the identifiers, numbers and string literals used in the program (and which are underlined when displayed as part of the abstract syntax tree).

In addition to the menus of operators discussed above, there is also a command menu, which is obtained when the

Figure 2. A typical Kookaburra view.
An Integrated Programming Environment Based on Multiple Concurrent Views

Figure 3. Another Kookaburra view.

Figure 4. The Eucalypt view.


cursor is over white space. This menu is shown in Figure 3 and is used to select commands which apply to the current node or to the entire program. An important group of commands in this menu are "Ellipsis", "Expand" and "Showall"; these are used to hide or redisplay the details of a subtree in the abstract syntax tree. For example, the details of "importlist", the third child of "Implementation Mod" in Figure 3, have been elided and the node now has a double line drawn around it; this may well have been achieved by making "importlist" the current node and then selecting the Ellipsis command. If more detail is required of an elided node, the Expand or Showall commands can be used. Expand displays only the next level of the tree, whereas Showall displays the entire subtree.

The remaining commands in the command menu provide various miscellaneous facilities. These include a command to delete the current node and its subtree, some facilities related to the provision of a clipboard with each view, and commands to save the abstract syntax tree (either in text form or in the form which permits fast reloading later). The clipboard can be used to move subtrees around the program, including between compilation units; however, a subtree stored in the clipboard can only be copied to syntactically legal places, as determined by the root of the subtree and the phylum of the node that it is to replace.

5. EUCALYPT: A VIEW ON THE PROGRAM DATABASE

The Eucalypt view is depicted in Figure 4 and enables the user to interact with the Eucalypt database, loading compilation units, saving them onto files, removing them from the database, and so on. At the top of the view is the current directory pathname; the upper of the large panels contains command buttons and permits the specification of a new current directory, while the lower panel lists the files associated with all the compilation units currently in the database.

Compilation units can be loaded into the database from files which have one of the following two formats: either from a file which has been written out by Eucalypt previously and which permits the fast reloading of a tree (files with an extension of ".db"), or from a textual description of a Modula-2 compilation unit which will then be parsed by the Database Process into a suitable abstract syntax tree (these files have an extension of ".mod" or ".def". Similarly, compilation units can be saved in the form which permits fast reloading or as a text file (with suitable prettyprinting).

6. CONCLUSIONS AND FUTURE PLANS

An integrated incremental programming environment, called MultiView and based on the notion of multiple concurrent views, has been outlined. This programming environment is designed to support the program development process by giving the programmer access to a variety of different views of the program under development.

A prototype implementation of MultiView has also been described; this implementation consists of a collection of parallel processes communicating via message
An Integrated Programming Environment Based on Multiple Concurrent Views

Figure 5. A session with the present MultiView prototype.

passing and runs on a network of Sun workstations. This prototype is certainly minimal in terms of the number of different views (with only one kind fully implemented at present); however, it has enabled us to verify that the system architecture depicted in Figure 1 is both feasible and useful. A uniform protocol is used for communication between the View Processes and the Database Process and this certainly contributes to the ease with which new kinds of views can be added. Also, the interface between a View Process and the Database Process has been encapsulated in such a manner that the details of socket communication are hidden from the person writing a new View Process.

One of the advantages of the system architecture and the use of a uniform protocol is the ease with which new kinds of views can be added. This means, for example, that a student undertaking a project to implement a new kind of view can quickly begin the project work; the student’s work is separate from the rest of the system, preventing interference, but they have to write very little of a View Process before they can begin to see something happening on the screen. The Kookaburra view described above was the result of such a project, as is a textual view called Koala, currently nearing completion (Lee, 1987). In fact, we have recently further reduced the time it takes to implement a new view by writing a “skeleton” View Process which already contains a basic local database and the code for interaction with the Database Process.

Figure 5 shows a MultiView session using the present prototype. Two Kookaburra views have been activated and are, in fact, connected to the same compilation unit in the Eucalypt database (one called “prog.db”); changes made to the compilation unit with one of the two views are propagated automatically and immediately to the other view. In addition to the Kookaburra view instances, there is also a Eucalypt view, showing the compilation units currently in the program database. The icons at the top of the screen represent views which have been closed (but not “quit”) to save space on the screen; they represent a Kookaburra view and a Koala view, respectively.

The present prototype, as mentioned earlier, is an environment for Modula-2 and is written in Modula-2. The first prototype, described in (McCarthy, 1985), was an environment for a subset of Ada* and was written in Lisp. It is a goal of the MultiView project to eventually be able to

* Ada is a registered trademark of the US Department of Defense — Ada Joint Program Office.
generate MultiView programming environments from language descriptions, as is presently possible with tools such as the Synthesiser Generator (Reps and Teitelbaum, 1984) for less sophisticated environments. However, it is also our policy that the design of the MultiView facilities should not be restricted by what can presently be generated from a language description. Thus, at present, only certain Modula-2 modules embodying the abstract syntax of a language (including the operators valid for a particular phylum, the nature of the operator templates, and so on) are generated from a description of the abstract syntax of the language, using a program called Gumnut. Other aspects of the implementation, such as the unparsing schemes used in the Koala view, have been carefully designed to permit automatic generation at some later point.

Obvious future enhancements to the MultiView prototype described in this paper include increasing the variety of views, implementing the incremental semantic analysis and code generation processes mentioned earlier, and introducing program execution (along with some suitable views of the dynamic program). For a discussion of some aspects of these, see (Marlin, 1986a).

The MultiView programming environment is intended to be one of the first applications for the Leopard multiprocessor workstation (Ashenden, Barter and Marlin, 1987), also being developed at the University of Adelaide. In fact, the multiprocessor nature of the MultiView implementation has been designed with the Leopard in mind. Before MultiView can be moved to the Leopard, the latter will have to reach the stage of development of having several processors, a shared memory subsystem, a high-resolution graphics subsystem, a distributed operating system and window management software. In the meantime, development will continue on the Sun network and this development work has already included, and will continue to include periodically, experiments where the MultiView implementation is run across the Sun network as a distributed application (rather than being run on a single node).

7. ACKNOWLEDGEMENTS

The authors would like to acknowledge the contribution of Michael McCarthy, who developed the particular form of abstract syntax used in the MultiView environment and who has since been involved in generating some of the components of MultiView concerned with the abstract syntax of Modula-2.

This work was supported in part by grants from the Australian Computer Research Board, the Australian Telecommunications and Electronics Research Board, the Australian Research Grants Scheme and the University of Adelaide.

8. REFERENCES


BIOPRAGICAL NOTES

Richard Altmann received his BSc(Ma) degree in Mathematics and Computer Science from the University of Adelaide in 1985, and his Honours degree in Computer Science in 1986. Currently he is employed as a Tutor in the Department of Computer Science at the University of Adelaide. He is working on the design and implementation of a structured programming environment that will be used to teach and support the department's courses. He is also involved in the development of a distributed programming environment that will be used to support the department's research in parallel and distributed computing.

Andrew Hawke was born in Adelaide, South Australia on 30 December 1963. He received his BSc degree in Mathematics and Computer Science from the University of Adelaide in 1984, and his Honours degree in Computer Science in 1985. He is currently working on the development of a parallel programming environment that will be used to support the department's research in parallel and distributed computing.
An Integrated Programming Environment Based on Multiple Concurrent Views

Chris Marlin has been with the Department of Computer Science at the University of Adelaide, where he is now a Senior Lecturer and Deputy Chairman, since 1984. He completed his Honours degree in Computing Science in 1973 and his PhD in Computing Science in 1979, both at the University of Adelaide. From January 1980 to December 1983, he was an Assistant Professor of Computer Science at the University of Iowa, Iowa City, Iowa (USA). His research has primarily been concerned with programming language design, specification and implementation, especially in relation to coroutines and parallel processes, and various aspects of integrated incremental programming environments.

OBITUARY

ROSS NEALON

Readers of the Australian Computer Journal will be sad to learn of the death of Ross Nealon on Sunday, 27 March 1988, after his fifteen-month struggle with Hodgkinson's disease. Ross was Wollongong University's first Honours Graduate in Computing Science and was one of Australia's pioneers and leading experts on the UNIX system. He accomplished much during his thirty years but there was much that he was still anxious to achieve.

Ross joined the University of Wollongong as a first year student in 1976 in the University's first group of Computing Science majors, and graduated with First Class Honours in 1979. Ross was a naturally gifted programmer. For his third year project he designed and implemented a universal cross-assembler for all then available microcomputers. The concept and code were very good, so that the cross-assembler was contributed to the Usenix distribution tape. It was so good that some years later, GenRad Inc., a well-known maker of circuit board testing devices in the USA, volunteered a payment for the use of Ross's software in their products, sufficient to buy his first microcomputer.

In December 1976, Richard Miller and Juris Reinfelds decided to port UNIX to the Interdata 7/32 computer. Ross, not having then learnt what is impossible, offered to help and was made responsible for the UNIX editor, ed, which he ported successfully in only a few weeks. This was a more impressive feat for a first year student!

Ross's last major project, which he finished in a remarkably short time, was software for the Cambridge Ring which now supports connections at Wollongong University between Computing Science terminals and computers without extra wiring or expensive crossbar switches.

Ross put a lot of work and energy into developing and adapting the nroff and troff suite of formatting tools for UNIX. In 1985, the Australian Computer Journal was experiencing difficulty in obtaining typesetting services of an adequate standard, especially for its more mathematical types of article. Since many of these were already available in machine readable form, the editor, John Lions, converted these to troff format and arranged with Ross to get them typeset on the Wollongong University Computer Centre's phototypesetter. A phototypesetter is no ordinary computer peripheral, and there were more than a few software hurdles for Ross to cross before all worked successfully: the first articles phototypeset by him appeared in November 1985; the final ones appeared in February 1988. The high spot was the August 1986 issue of the ACJ that was set entirely at Wollongong with Ross's assistance. His work on this task was extremely valuable. Much of it was done in his own time but with typical modesty, he often failed to claim full reimbursement for his services.

Ross helped to shape Computing Science in Wollongong from its beginnings. He will be sorely missed by his young family and many colleagues.
An Experiment in Modelling Communication Protocols using NPNs

C.C. Ang† and E.J.P. County†

This paper describes the results of some work done recently in the area of communication protocol modelling using a formal description technique. The communication protocol chosen to be modelled is a section of the yet unrealised OSI FTAM communication protocol (ISO DIS 8571) (ISO, 1986). The FTAM communication protocol is specified in three sections — the basic file protocol, the basic bulk data transfer protocol and the error recovery protocol. Due to time limitations, the basic bulk data transfer protocol and the error recovery protocols were not modelled. The part of the FTAM protocol that was modelled is the basic file protocol. The formal description technique used is an extension of Petri Nets (Reisig, 1985) known as Numerical Petri Nets (NPN) (Symons, 1978). The NPN model constructed was later verified using the PROTEAN validation tool.

Keywords and Phrases: Numerical Petri nets, formal specification and verification, computer network protocols, file transfer protocol, OSI FTAM, reachability analysis, PROTEAN.


1. INTRODUCTION

The International Organisation for Standardisation (ISO) has introduced a seven layer reference model (Day and Zimmerman, 1983) to try to achieve compatibility among heterogeneous computers. This model is known as the Basic Reference Model for Open Systems Interconnection (OSI). It provides a framework from which a set of ISO standardised communication protocols may be developed. By mutual use of these rules defined in the OSI model, independent systems can communicate with one another. To reduce complexity of the modelled system, the reference model is decomposed into seven smaller, hierarchical layers. This makes the model more manageable and allows parallel development of the layer protocols. These seven layers are known as the physical, the data link, the network, the transport, the session, the presentation and the application layer (Day and Zimmerman, 1983). Figure 1 shows the structure of the ISO Reference Model.

Each part or layer (layer N) in the model performs a function and together with the services provided by the layers (layer N-1) below, if any, offer a service to the layer (layer N + 1) above it.

The co-operation between these layers and their peer entities is handled by a communication protocol. The communication protocol of interest in this paper is the file transfer protocol in the OSI model which is known as File Transfer Access and Management or FTAM. This protocol is found in the application layer of the OSI reference model. A section of this communication protocol is specified using Numerical Petri Nets (NPNs). This paper presents part of the NPN model constructed together with results of the verification of this model. The purpose of this exercise is to determine the suitability of NPNs as a modelling tool for complex communication protocols such as the ones in the Application layer of the ISO Reference Model (ISORM).

2. THE APPLICATION LAYER

The application layer (Barton, 1983) is the highest layer in the OSI Reference Model. This layer is responsible for providing a framework that can be used to design communication protocols which can be implemented in the network for information processing and data transfer. Services provided by the OSI environment can only be accessed through this layer.

The services provided in this layer can be categorised into two classes, Specific Application Service Elements (SASE) and Common Application Service Elements (CASE).

The SASE group includes services like electronic mail, file transfer and remote job transfer. CASE consists of services which can be used by all applications.

2.1 FTAM

FTAM is the communication protocol which is worked with in this paper. FTAM is a file transfer protocol (FTP)
which defines the set of rules that determines the procedure of transferring a file or a set of files from the file system of one computer to the file system of another. Its primary objective is to enable the sharing of data and programs regardless of the variations in file storage between computer systems. This service is of fundamental importance in computer communications. FTAM is a SASE and the ISO defined communication protocol for file transfer in their OSI model. The ISO is currently working on making this Working International Standard (DIS) into an ISO International Standard (IS).

FTAM provides a reliable transfer of bulk data. It also has provisions for file access and file management. Two levels of error management are provided, a User Correctable File Service (UCFS) and a Reliable File Service (RFS).

UCFS is the file service which consults the user when an error arises. Thus, with this level of service, the user can select the style of error management. The basic protocol, which consists of the basic file protocol and the basic bulk data transfer protocol, supports this service level using the CASE and the Presentation layer services.

RFS provides automatic error recovery. The user has no awareness of error recovery which is handled by the FTAM service provider. The quality of the service which is required has to be specified by the user. The entire file transfer will be seen by the user as a series of error free operations. The error recovery protocol supports this file service level using the UCFS service level.

2.2 The Virtual Filestore

The model which FTAM works on involves three entities — the controlling entity, the source and the destination entity. This reduces to a two-party model when the controlling entity is also the source or destination entity.

FTAM is based on the concept of virtualisation. It uses a definition of a virtual file structure to move files around. This virtual file structure provides a common model for file service and attributes and a common set of file operations. This common model is known as the virtual filestore. This filestore describes how files are stored and structured together with other bits of information about a file. This description includes creation and access dates, permitted actions on the file, access rights and privileges, owners plus other information. The above information together with information about the virtual file specification will be part of the information transferred by the protocol.

A host wanting to transfer a file to another host would have the specified file locally mapped from the real file system onto the virtual filestore. This process is reversed at the destination and is translated from the virtual file structure onto the real file structure on the remote system.

The more common functions performed by a file transfer protocol include data transformation, information transfer and file manipulation.

Figure 2 shows the FTAM file service regimes. It can be seen from this diagram the phases that have to be established in order to achieve a file transfer.

2.3 Concatenation of Service Primitives

FTAM provides a facility where the user is allowed to concatenate requests which can be processed as a group. This facility is indicated by a “Begin Grouping” service primitive and terminated by an “End Grouping” service primitive.

This facility allows a number of regimes to be established or released in one interaction. The responder will not respond to these requests until the group is completed and will return these responses in a group.

3. SPECIFICATION OF THE MODEL

There are a number of different techniques which can be used to specify a model. The more popular ones include temporal logic (Hailpern, 1982), a technique where the desired behaviour of the protocol is formulated and then proven or disproven; and finite state machines, a simple type of a state transition model. Sunshine (1979) has reviewed some of these methods and an evaluation of them can be found in Sajkowski (1984).

3.1 Numerical Petri Nets

Numerical Petri Nets (NPNs) is a specification technique that belongs to the family of state transition models. It can be specified either mathematically or graphically (Best and Fernandez, 1986). The graphical NPN consists of a bipartite directed graph, fixed firing rules and an initial marking.

A NPN $G$ can be defined by a quintuple $(P, T, F_t, F_o, M_0)$ where

- $P$ set of all places in the net $G$, where a place is represented by a circle in the NPN graph and can be used to represent a machine state;
- $T$ set of all transitions in the net $G$, where a transition is represented by a bar in the NPN graph and can be used to represent an event;
the input firing rules, usually written on the arcs between the input places and their respective transitions; this specifies what tokens are required to be in the input places to enable the transition and what tokens are destroyed/removed when the transition is fired;

$F_0$ the output firing rules, usually written on the arcs between the transitions and their respective output places; this specifies what tokens are produced/created in the output places;

$M_0$ the initial marking of the net, this specifies the initial marking of each place, $p_i \in P$.

A new marking of the net $G$ may be obtained by firing a transition, $t_i \in T$, according to the rules specified in $F_i$ and $F_0$.

Numerical Petri Nets is actually a generalisation of Petri nets (Reisig, 1985) that was developed by F. Symons (1978). These extensions include a global memory on which transitions can perform read and write operations. This memory area can be referred to in order to enable a transition. In the graphical NPN, this read operation is written next to the transition inside square brackets. A definition of NPNs can be found in Wheeler (1985).

A specification of the logical behaviour of the initiating and responding entity actions of the FTAM communication protocol was constructed using NPNs following the specifications and definitions set out in the ISO FTAM Working International Standard (ISO, 1986). The model that was constructed was set up in a similar fashion to the way in which Billington (1982) specified the logical behaviour of the Transport layer protocol.

3.2 The NPN Model

The FTAM file protocol specification is given in three parts. There is the basic file protocol, the basic bulk data transfer protocol and the error recovery protocol. The first two parts support the UCFS service level and the third supports the RFS file service. The basic bulk data transfer and the error recovery protocols are not modelled. Only the basic file protocol is modelled. The model of this was set up using Numerical Petri Nets based on the file protocol machine shown in Figure 3 and the specifications for the basic file protocol provided in the ISO documents.

Figure 4 shows the NPN model constructed for the FTAM file delete service. This model contains eight places and twelve transitions. Figure 5 shows the NPN model for the Beginning of Grouping service and the End of Grouping service, which makes up the FTAM File Grouping control.

The places $ucfsA$ and $ucfsB$ in the figures represent the initiating and responding entities respectively. The places $inA$, $inB$, $outA$, and $outB$ represent the input and output ports for the initiating and responding entities. The two places in the dashed box serves to model the queue for signals through to the OSI environment (OSIE).

The protocol states corresponding to the values that the variables $A$ and $B$ have in the NPN diagrams are as follows:

**State**

0 state "Idle".

6 state "Delete Pending".

11 state "Grouping Requests".

12 state "Grouping Pending".

21 state "Initialised".

![Figure 3. Model of the File Protocol Machine.](image-url)

![Figure 4. FTAM File Delete Service.](image-url)

![Figure 5. FTAM Begin and End Grouping.](image-url)
22 state “Selected”.
23 state “Data Transfer Idle”.
24 state “Grouping Responses”.

The variables delin, ratin, catin and cloin are state indicators which help model the expected response list. The variable isi (initial state indicator) is used to record the state of the entity at the time when invoking the grouping function. The variable nsi (next state indicator) records the state which will be entered when the grouped responses are received. These variables, delin, ratin, catin, cloin, isi and nsi are unset initially and are unset again in the NPN model when a transition with a clear ind (clear all indicators) operation is encountered. The operators AND and OR are denoted by the symbols & and | respectively in the NPN model.

The input and output firing rules for the transitions in the NPN model for the FTAM file delete service in Figure 4 are as follows:

Transition
t1, t2 consumes 1 token of any type, produces a delete-request service token in inA.
t3, t9 consumes and produces the same token into OSIE.
t4 consumes a delete-request service token; produces a delete-request protocol data unit (PDU) token.
t5 token consumed must be a delete-request PDU token; produces a delete-request PDU.
t6 token consumed must be a delete request PDU token which must be the oldest token in the place (modelling a queue).
t7, t8 token consumed must be a delete request PDU which must be the oldest token in the place; produces a delete-response service token.
t10 consumes a delete-response service token; produces a delete-response PDU token.
t11 token consumed must be a delete-response PDU token; produces a simple token.
t12 token consumed must be a delete-response PDU token which must be the oldest token in the place; produces no tokens.

The firing rules associated with the transitions for the FTAM Begin and End Grouping NPN model in Figure 5 are as follows:

Transition
t1 consumes 1 token of any type, produces two tokens, a simple token into ucfsA and a begin-grouping request service token into inA.
t2 consumes 1 token of any type and produces an end-grouping service token in inA.
t3 consumes all and produces the same tokens into OSIE.
t4 consumes all the request service tokens and produces the same corresponding request PDU tokens into OSIE.
t5 token consumed must be a end-grouping request PDU token which must be the oldest token in the input place; produces a end-grouping request PDU.
t6 consumes and produces a begin-grouping request PDU token.
— A net is **safe** iff $\forall p_i \in P$ with initial marking $M_0$ and $\forall$ markings in set $M$, the marking of $p_i \leq 1$.
— A net is **proper** iff $M_0 \in M$ i.e. the initial marking of the net can be reached again.
— A net is **live** iff $\forall$ markings in $M$, and $\forall r \in T$, $\exists$ a firing sequence from $M_0$ that fires $t$ at least once.

The NPN model is analysed using *Reachability analysis*, a major analysis technique used with Petri nets. This method is very suitable for use with specification techniques that belong to the family of state transition models of which NPNs is one.

This method involves the exhaustive investigation of every marking that the net can produce from a given initial marking.

Consider the Petri net in Figure 6.

![Figure 6. Petri Net example.](image)

The initial marking of this net, $M_0$ is $(1,1,0)$. At this point, the transition $t_1$ is enabled since all its input places ($p_1$ and $p_2$) have the required number of tokens (in this case 1). When $t_1$ is fired, the tokens in the input places $p_1$ and $p_2$ are removed and a token is inserted into $p_3$, which is the output place for the fired transition. The marking of the net is now $(0,0,1)$ and transition $t_2$ is now enabled and can fire.

By repeatedly firing the enabled transitions, new markings are produced and a reachability tree can be constructed. The nodes on a reachability tree of a net $G$ are all the reachable markings of the net.

Properties of the net such as the ones outlined previously can be verified by examining the reachability tree. A reachability tree can get very big which makes analysis difficult but techniques exist which can be used to reduce its size. Tools exist which have automated some of the techniques which can be used to verify some net properties.

### 4.1 Validation Tool

Reachability trees that are generated by more complicated nets tend to be quite big which makes their analysis long and difficult. There are many software packages around which have automated techniques for analysing the reachability tree (Feldbrugge and Jensen, 1986).

The tool that was used here to verify the model is one that was developed at the Australian Telecom Research Laboratories and is known as *PROTEAN* (Billington, Wilbur-Ham and Bearman, 1985). It produces a reachability graph using NPN specifications as input. Functions which can be used to analyse and manipulate the reachability graph are also provided. These analysis functions include the programs LIVENESS, which determines all the strongly connected components of the graph i.e. all sets of nodes where each node in the set can reach every other node in the set, SCENARIO which can find the path that matches a specified transition or marking sequence in the graph, CYCLE which can find cycles in the graph and PATH which can find all the nodes that cannot lead to a specified node.

### 4.2 Validation Results

From the NPN model, the protocol is tested to see if it meets its service specifications. The procedure used is reachability analysis, where the markings in the reachability tree associated with the model is investigated.

The validation is broken down into two parts; the model for the concatenation of service primitives is validated separately. Using PROTEAN, the nets were checked for deadlock freeness, safeness, boundedness and liveness.

Initial processing of the net revealed a typographical mistake in the specification for *FTAM Regime Termination*. The document defines the state "context relinquishing pending" to be the initiator’s present state in order to receive a `terminate response PDU`. However, this state does not seem to be referenced anywhere else in the document and is identified to a state name left over from an earlier version of this ISO document. This state was then changed to become state "terminate pending", which is the state the initiator enters after issuing a `regime termination request primitive`. This modification removed the deadlock caused by the previous state.

Further processing of the net produced 90 markings and 3 deadlocks. These deadlocks were found to be the NPN models for *FTAM Regime Termination, File Deselect* and *File Open*. The cause of these deadlocks appear to be that the machine states that are specified for communication between the initiator and responder do not correspond. For example, the protocol specification states that the responding machine has to be in state "selected" to receive a `deselect request service primitive`. Checking the reachability tree produced, the responder seems to be in state "data transfer idle" instead. These deadlocks prevented the further validation of other parts of the net. Steps were then taken to correct these deadlocks so as to further the validation. A possible solution to the above deadlock was provided by the specifications for the initiator. The document states that the initiator should be in state "data transfer idle" when allowed to accept a `deselect request service primitive`. When the responders’ state was altered from state "selected" to state "data transfer idle", the deadlock was removed. This allowed the validation to proceed to parts of the net which were previously unexplored. This solution is unique since when other possible states were substituted with the above, the same deadlock was produced. The deadlocks for the other two NPN models were removed in similar fashion.

The validation process was then repeated with the modified NPN input. PROTEAN produced 105 markings with no deadlocks. This time, the net revealed that it was safe, proper and live. The sequence of the regimes established in the model corresponded to that given in the specifications (refer Figure 2).

Using the state tables provided in the ISO document,
which does not consider the state indicators, to construct
the NPN model for the concatenation of service primitives.
the analysis showed that the model was not safe but is
deadlock free, bounded and proper. It is not safe because of
the service primitives that will have to be grouped
together. However, this net is deadlocked if the state indi-
cators are taken into consideration. The deadlock occurs
after an OPEN file is requested. The values of the state indicators
do not correspond after the OPEN service primitive is
received.
A more thorough description of these results can be
found in Ang (1986). At the time of writing, there is no
version of the ISO FTAM specifications which supercedes
the one used.
5. CONCLUSION
An attempt has been made to try to model a section of the
ISO FTAM communication protocol using Numerical
Petri Nets. Only the basic file protocol (the initiating and
responding entities' action) has been modelled. Work on
modelling the entire OSI FTAM communication protocol
especially the error recovery protocol together with other
application layer protocols using NPNs are being done in
other projects. These results will be reported at a later date.
A state exploration analysis technique has been applied
to the set of the FTAM models' states and several undesir-
able properties have been detected in the model. NPNs
seems to be an adequate tool which can be used to specify
and validate communication protocols. It appears to be
"powerful" enough to model more complex systems like
the communication protocols in the OSI suite. Like all net
based techniques, it does suffer from the state explosion
problem but it is not as severe as its less developed coun-
terparts. Analysing large numbers of states have been
made easier with the use of computers. There exists a
variety of software (Feldbrugge and Jensen, 1986) around
which have automated analysis techniques. The availability
of these software aids make the validating task much
quicker and less tedious. There has been a lot of work done
in making these tools functionally more powerful but more
effort still needs to be put into making it possible to de-
scribe systems using nets in a more modular and hierarchi-
cal manner.
6. ACKNOWLEDGEMENTS
The authors would like to acknowledge the assistance of
Telecom Australia for providing access to the PROTEAN
modelling package, ATERB and CSIRO Division of
Information Technology (DIT) for their financial support.
The authors are deeply grateful to Professor T.S. Dillon,
Mr R. Lai, Dr K. Parker of CSIRO DIT, and the anony-
mous referees for their helpful and constructive comments.
Mr C.C. Ang is presently under a postgraduate grant under
the CSIRO DIT collaborative scheme.
7. REFERENCES
and Simulation, pp. 141-145, Melbourne, Australia, 14-16 October
1987.
ANG, C.C. (1986): Formal Modelling of Some ISO FTAM Commu-
nication Protocol Sections. October 1986. BSc(Hons) Project Report, La
Trobe University.
of Open Systems Interconnection. Proceedings of the IEEE,
Datenverarbeitung, January 1986. (Arbeitspapiere no. 195.)
Workshop on Protocol Specification, Testing and Verification, pp. 77-
100, IFIP, North-Holland, Ildillyd, California, May 1982.
BILLINGTON, J., WILBUR—HAM, M.C. and BEARMAN, M.V.
International Workshop on Protocol Specification, Testing and Verifi-
cation, pp. 59-70, IFIP, North-Holland, Toulouse-Mossac, France,
June 1985.
W. Reisig, W. Brauer and G. Rozenberg, editors, Petri Nets: Applica-
tions and Relationships to Other Models of Concurrency, LNCS 255,
Logic. Volume 129 of Lecture Notes in Computer Science, Springer-
ISO/TC 97/SC 21/WG 5 (1986): Revised Text of ISO/DP 8571, Informa-
tion Processing Systems — Open Systems Interconnection — File
Transfer Access and Management — Parts 1, 2, 3 and 4, Working
International Standard DIS 8571, International Organisation for
Standardisation, New York, USA, July 1986.
Prentice-Hall, Englewood Cliffs, NJ.
Monographs on Theoretical Computer Science, Springer-Verlag.
J.M. Bennett, editor, Proceedings of the Seventh International Confer-
ence on Computer Communication, pp. 579-584, The New World of
the Information Society, Sydney, Australia, 30 October-2 November
1984.
Protocols using Numerical Petri Nets. PhD thesis, Department of Elec-
trical Engineering Science Telecommunications, University of Essex,
May 1978.
Report 7780, Telecom Australia Research Laboratories, Clayton,
Instrumenting Systems to Measure Components of Interactive Response Times

J.P. Penny†, P.J. Ashton† and D.G. Tripp†

A formal model is presented of a procedure for measuring system responsiveness, and breaking interactive response times into components to show where delays occur. The model allows for a most flexible decomposition of response times, one that would measure the contributions to response times from using, or being queued for, any designated resources, singly or in any combinations. It is suggested that an implementation of the model would be very simple, if considered at the time of system design. The model has also provided insights found useful in adding experimental performance monitors to existing systems, and these monitors are described.

Keywords and Phrases: Interactive response times, performance measurement, time-sharing systems, distributed systems.


1. INTRODUCTION

Ferrari (1986) has pointed out that, as a discipline, computer systems performance evaluation has developed in relative isolation from the kindred areas of computer design, installation and management. As a result, measurement has been an “afterthought” in system design, done most frequently with instruments added from the outside to running systems. In this paper, we consider the instrumentation needed to measure what the on-line user of an interactive system is likely to see as the most important class of quantitative performance indices (Barker, 1983), those which describe the responsiveness of the system.

Measurements of system responsiveness are required to show how well (or badly) the system is performing, and also to determine the causes of unsatisfactory performance. An important type of measurement tool is one that can break response times into components in order to show where response time delays occur. In this paper a formal model is presented of a procedure for recording the state of a virtual process that is assumed to support each interactive user, and for measuring the time spent in each state. The model allows for a most flexible decomposition of response times, showing the contributions made to these times by using, or being queued for, any resources, singly or in any combinations. For the model, the term “resources” is not limited to hardware units, but can be taken as generally as may be desired.

Facilities to implement the model could very easily be “built-in” to operating systems if considered at the time of design. While adding measurement facilities to working systems is difficult, experimental performance monitors are described that have been designed for three different systems, with the design being based on insights derived from the formal model.

The present is a time of evolution from traditional, single-processor time-sharing systems to systems that are more elaborate in the number of processors and variety of their configurations. As the number of hardware and software units on the other side of the terminal proliferates, the number of causes of possible delays increases, as does the difficulty of finding out when contributions are made to the total response time. It is shown that the model is also applicable to a range of more distributed processing systems.

2. CONCEPT

The formal model of the recording and measurement procedure is based on the view that on-line users are the key components of an interactive system and that, associated with each user, there is a virtual interactive process that carries out the work needed to service requests from that user. The fact that, in most systems, the work of each virtual process may be done by many internal processes causes no difficulty in practice.

Components of Response Times

The model has been developed from a standard technique for evaluation of program performance described by Ferrari (1978), where a program is assumed at any time to be in one of a set of disjoint states. The behaviour of a program is then described by “the sequence of states visited during execution and the time spent in each state”.

The model is developed in detail in Penny, Ashton and Wilkinson (1986). An arbitrary set I = {I₁, . . . , Iₚ} of interactions I, between user and machine is considered, for example the interactions performed in some measurement interval by those users for whom interactive performance is of interest.
The interactive process supporting any interaction is assumed to spend its time in one or other of a set \( S = \{ S_1, \ldots, S_q \} \) of disjoint states \( S_j \). If \( T_{S_j}(i) \) is the total time spent in state \( S_j \) during interaction \( I_i \), then response time \( RT(i) \) for that interaction is

\[
RT(i) = \sum_j T_{S_j}(i)
\]

and

\[
\sum_i RT(i) = \sum_j \sum_i T_{S_j}(i) = \sum_i \sum_j T_{S_j}(i).
\]

If \( C_{S_j} = \sum_i T_{S_j}(i) \) is the total time spent in state \( S_j \) by the interactive processes servicing all interactions in \( I \), then values of \( C_{S_j} \) give components of the sum of response times for the set of interactions considered.

**State Recording and Measurement**

Servicing an interaction requires resources. The state of an interactive process at any instant is determined by the resource(s) being used or being queued for, by any task doing work for that interactive process. The set \( R = \{ R_1, \ldots, R_r \} \) of resources \( R_m \) employed for state definition can include any resource whose use may contribute to the time for servicing any interaction. These need not be only hardware units; software modules, or parts of a database, can be included.

Assume that, for each user, the state of the interactive process will be held in a *state word* of \( 2^r \) bits, one pair of bits for each resource in \( R \). The first bit of the pair is to be \( 1 \) iff work for that interactive process is being run on that resource; the second bit is to be \( 1 \) iff work for that process is queued for that resource. When the integer equivalent of the state word is \( j \) the interactive process is said to be in state \( S_j \).

Measurement of times spent in each state requires a sampling monitor to scan the set of state words at fixed intervals of \( \Delta T \) seconds. If there are \( K \) samples, and \( f_{kj} \) interactive processes are found in state \( j \) on sample \( k \), then \( C_{S_j} \) is estimated by:

\[
C_{S_j} = \sum_{k=1}^{K} f_{kj} \cdot \Delta T.
\]

**Implementation**

Algorithms for an implementation of the above model are given below in Pascal; elsewhere in the text Pascal identifiers are given in italics. Where a Pascal identifier differs from the notation of the model above, any equivalence is shown in a comment to the program.

The stateword for user *user* is `stateword [user]` with a pair of bits for each resource. The first bit, `rbit`, shows whether the interactive process for *user* is actually using resource `resource` (for example, running on that processor, executing that piece of code, referencing that part of the database . . .). The second bit, `qbit`, shows whether work for *user* is queued to use `resource`.

An implementation requires two parts: a method of recording the state of each interactive process, and measurement by a state sampling monitor of time spent in each state. The *measurement* phase is:

```pascal
program monitor_MeasurementPhase;
const
    no_of_states = .......(2^r; see selective measurement in Section 3)

type users = (user1, user2, ..., userN); //list's of users to be monitored!
resources = (R1, R2, ..., Rr); //set R of the model
resourceBits = record
    rbit, qbit: boolean;
end;
stateword = array[resources] of resourceBits;
var statewords: array[users] of stateword;
statecount: array[1..no_of_states] of integer;
//to accumulate \( \Sigma f_{kj} \)
user: users;
resource: resources;

begin (current sample)
    for user := user1 to userN do
        for resource := R1 to Rr do
            statewords[user][resource].rbit := false;
            statewords[user][resource].qbit := false;

    for user := user1 to userN do
        statecount \[j\] := 0; \( j = 1, 2, ..., 2^r \)
    for user := user1 to userN do
        for resource := R1 to Rr do
            for j := 1 to no_of_states do
                intval := intval(statewords[user]);
                if intval = j then
                    statecount \[j\] := statecount \[j\] + 1;
    end;
end; //response time components found by statecount[]
```

In theory, the *recording* phase is very simple. Event recording is needed in procedures managing each resource whose effect on responsiveness is of importance:

```pascal
// when a process for user is added to a queue for resource resource
statewords[user, resource].qbit := true;

// when a queued process for user starts to use resource
statewords[user, resource].qbit := false;

// when use of resource is completed
statewords[user, resource].qbit := true;
```

In practice, the installation on a working system of these fragments of event recording code is likely to be difficult. In the working monitors to be described that are based on the model, the states of the interactive processes are determined by the monitor itself, by working out what conditions the (real) process(es) effecting each interactive process are waiting on. Determination of an interactive process state was found to range from very easy to impossible (at least to the degree of detail wanted), the difficulty varying according to the condition causing a process to be blocked.

**3. GENERALITY**

If set \( R \) has \( r \) resources, then an interactive process may take any of \( 2^r \) states. The particular case where all bits in the stateword are *false* can be taken as "think state"; the interactive process is not using, or queued for, any resources.

If \( R \) contains many resources, the number of possible states is large. However, the maximum amount of information should be available for measurement if required. The recording phase is, at least in theory, very simple, and recording the large number of possible states presents no difficulty because the statewords are so concise.

For the measurement phase, one is unlikely to want, as the monitor outline suggests, a `statecount` for each of \( 2^r \) possible states. Interest will probably be to measure peri-
ods in which the interactive processes, individually or collectively, satisfy some condition that might be specified as a simple function of states of individual interactive processes.

Examples are given by Penny, Ashton and Wilkinson (1986) of using selective measurement:

1. To measure resource utilizations — by counting the proportion of samples for which \( rbit \) of any stateword = true for that resource.
2. To measure the extent of overlapping operation of any subset of resources — by counting the proportion of samples for which the \( rbits \) of any stateword = true for each resource of that subset.
3. To find whether any resource is a bottleneck — by counting the proportion of all readings (that is, for all samples and all statewords) for which the \( rbit \) or \( qbit \) for that resource = true.

The model procedure is entirely general:

1. It can deal with any group of users, from a single user to all users on a system.
2. The resources whose influence is to be measured can include anything of interest: hardware units, physical or logical; software modules; major parts of a database.
3. Contributions to response times can be measured due to use of, or queuing for, any resources, singly or in any combinations of concurrent operation.

4. A MONITOR FOR PRIMOS

The model procedure for state recording and measurement was developed in parallel with design and construction of a software monitor for Prime machines running under PRIMOS. The monitor is described briefly in Penny, Ashton and Wilkinson (1986) and documented in Ashton (1984). Examples of results are given in Penny and Ashton (1984), and in Penny, Ashton and Wilkinson (1986); Penny and Ashton (1984) includes a description of how monitor results are used to plot response ratios against workload. Plots of this kind have been found particularly useful.

For a reasonably substantial operating system, PRIMOS may be unusual in having a one-one correspondence between each interactive user and an internal process control block (PCB). Concurrency of process execution for one user is achieved either by system processes which control, for example, disc transfers, or (less often) by the user establishing “phantom” processes to run in the background.

Each PCB spends its time attached by a “semaphore pointer” to any of a number of semaphores, locks, lists, or queues. An early Prime architecture manual includes the curious remark that “every PCB in the system must always be somewhere”! Measuring the times spent by a PCB in each position gives an initial decomposition of response times for the user associated with that PCB. For example, the PCB at the head of the ready list is recorded as being in CPU-run state on the assumption that that was its state at the time of the clock interrupt to run the monitor.

The sampling monitor itself was run initially as a user process at highest priority, but later installed as a system process at a priority immediately above that of the disc interrupt handlers, making it more accurate. Run typically at one-second intervals, the monitor cycles through the PCB's for a designated set of interactive users. A skeletal outline of the monitor is:

```plaintext
program Primos_monitor
    type ......(declarations for users, resources, stateword, resourceBits as in model_measurementPhase)
        pcbtype = record
           ...........
           sema4ptr:integer;
           ...........
        end;
    var ......(declarations for statewords, user, resource, as in model_measurementPhase)
        pcb: array[user] of pcbtype;
        run, queue: boolean;
    begin (current sample)
        ..........(set all bits of statewords:= false)
        for user := user1 to userN do begin
            analyse(pcb[user].sema4ptr, resource, run, queue);
            (from the sema4ptr value, determine
            (a)the resource on which the process is waiting, and
            (b)whether it is running (run:=true) or queuing
            (queue:=true))
            statewords[user, resource].rbit := run;
            statewords[user, resource].qbit := queue;
        ..........(increment relevant statecount, as in model_measurementPhase)
        end;
    end.

For each PCB, the monitor first examines the semaphore pointer. To determine most states, a more detailed analysis is needed, and the step summarised above as analyse is substantial. For example, a process waiting for a disc transfer has its PCB attached by the semaphore pointer to a disc queue control block (QCB). From the position and contents of the QCB, the monitor can determine: whether the transfer has been initiated, which physical disc is used, and what logical partition is involved. A very detailed breakdown is possible of the contribution by disc transfer to response times.

The Primos monitor is a substantial piece of software; production of a useful measurement tool was as important an objective as demonstrating the utility of the formal model. The monitor uses 10K words of memory and, running at a rate of one sample/second, was measured as using 0.7% of CPU time. If states of interactive processes were explicitly recorded, as suggested for the model, sampling would be much simpler. It should then be economic to run monitors at very high sampling rates.

5. A UNIX MONITOR

The one-one correspondence between interactive user and PCB under Primos simplified construction of a monitor that, in its measurement phase, corresponds to the formal model. At the time of writing, a monitor is being constructed for a Vax11/750 under UNIX, whose interactive users tend to create a large number of concurrent processes.

The Unix monitor has been designed to show that a one-many correspondence between (virtual) interactive processes and (real) internal processes presents no difficulty for an implementation corresponding to the model. The design also conforms to the model in that interactive process states are explicitly recorded.

The monitor is divided into two parts: the first determines the state of each interactive process and records it in

the appropriate state-word, the second part records the time in each state as specified in the measurement phase of the model. A skeletal outline is:

```
program Unix_monitor;
const nproc = ... (maximum number of processes in the process table)
type ... (declarations of users, resources, stateword, resourceBits as in monitor_measurementPhase)
var ... (declarations of statewords, statecount, j, user, resource as for monitor_measurementPhase)
proc: integer;
r, q: boolean;
begin ................{ set all bits in stateword :* false }
begin ...................(set all bits in stateword := false )
{analysis of interactive process states - recording phase}
for proc := 1 to nproc do
  begin
    user := finduser(proc); (find which user initiated proc )
    analyze(proc, resource, r, q); (find the resource that proc is using or queued for)
    statewords[user,resource].rbit := r or statewords[user,resource].rbit;
    statewords[user,resource].qbit := q or statewords[user,resource].qbit;
  end;
  begin [state sampling - measurement phase]
    ......................{as for monitor_measurementPhase}
  end;
end.

As in the Primos monitor, interactive process states are determined by the monitor on each sample. From work done so far, a number of comments can be made. First, it was found easy to decide from the process table which user initiated each process (giving the value user).

Second, a Unix user can have both foreground and background processes associated with a terminal. For most users, the primary interest lies in performance of the foreground job; the user may be editing one file while waiting for compilation of another in the background. However, a user will sometimes run a foreground job simply to check on progress of the background job, interest being in performance of the latter.

Though not shown in the Pascal code above, it was decided to keep separate statewords for foreground and background processes, so that analyses can be performed for response of foreground processes, background processes, and (by combining the two statewords) all processes.

Third, the determination of states in several cases presented difficulties, as it did under Primos, again showing that requirements for performance evaluation are not considered during system design. Some information can be gleaned from the process table, but further analysis is required.

6. MORE COMPLEX SYSTEMS

Systems are becoming more complex. A request from a user may be serviced by a local processor, a remote processor, or a combination of several processors both local and remote. Increasing complexity makes it more important, and more difficult, to locate and measure the components of response time delays.

The monitors so far described were designed for single-processor time-sharing systems. Systems in which a computation may run on more than one processor are now examined to show that the model procedure can be used for response time decomposition on more complex systems. The system classification is adapted from definitions given by Peterson and Silberschatz (1985) or Kleinrock (1985), with systems considered in an ordering of being progressively more distributed.

A tightly-coupled system is one in which processors share memory and clock, with communication through the common memory. For an implementation of the model, statewords for the interactive processes can be stored in the common memory, accessible for both the bit setting of the recording phase and for the monitor itself.  

The model procedure should be particularly valuable for evaluations of parallel processing, the form of multiprocessing that takes place when multiple processors cooperate closely to process tasks from the same job (Kleinrock, 1985). For any user or set of users, the model provides for measurement of the extent of concurrent use of any designated subset of resources. As an example, suppose that a program comprised of modules $M_1 \ldots M_n$ is being run on a system with processors $P_1 \ldots P_b \ldots P_N$. If each of $M_n$, $P_b$ is a resource with corresponding bits in the state-word, one can measure the extent of concurrent operation of any combination of processors (or any other resources) in any parts of the computation.

In a more loosely-coupled system, processors do not share memory or clock. To implement the model we need:

1. For each user’s stateword to be itself distributed across memories accessible to at least one of the processors executing tasks for that user,

2. The passing of the user id (in terms of the model, the value user) to any routine that will allocate or deallocate resources,

3. Access by a sampling monitor to each memory.

Of these requirements, the third seems the most difficult, but an example of what is needed has been described by Balkovitch and Soceanu (1982), in their case for a parallel processing system. Their experimental system had separate serial interfaces through which performance data collected locally could be reported back to a host processor. Any system equipped with this type of performance monitoring hardware would satisfy the remaining requirements for implementing the model.

The stateword for each user’s interactive process would be distributed across several memories, as suggested in Figure 1.

Any loosely coupled system with hardware to read...
performance data from each memory should allow an implementation of the model with little more difficulty than for a single-processor system. The passing of the value user requires an addition to the protocols for remote system calls. If the called processor did not already know of that user, it would have to establish a (partial) stateword for the user, with bits corresponding to the resources accessible to jobs running on that processor.

7. A DISTRIBUTED SYSTEM
A simple distributed system may be based on a local area network, with personal workstations sharing a file server and other resources. With powerful workstations available at low costs, these configurations are becoming very common.

Here, one wants to know first what contribution is made to response times in either the workstation or the resources accessed through the network and, possibly, the extent of concurrent operation. A division is needed of response times into local and remote components, and a component of overlap. The effects of upgrading either the local or remote resources can then be estimated.

Local, Remote and Overlap Components
A Novell network with IBM PCs connected through a Netware LAN (Novell, 1986) to a file server was studied. In this case, if a workstation requires service from the network, local execution is blocked until completion of the remote service. Therefore, local and remote states are disjoint, and response time RT(i) for some interaction can be divided:

\[ RT(i) = RT_L(i) + RT_R(i) \]

where \( RT_L(i) \) is the local component, time spent using local resources, and \( RT_R(i) \) is the remote component, time spent communicating through, running on, or being queued for, remote resources.

From this point, decomposition of local and remote components can be done independently, as for the model in Section 2. It is unimportant where the workstation lies in the spectrum between the simple, single-user, single-process system and the powerful, multi-user, multi-process system. The model was developed to cater for the latter and includes the former as a special case.

In some networks, such as the SUN NFS (SUN, 1986), each workstation will run many processes. While one process requests and waits for remote services, other processes will continue to run locally. The initial decomposition must include a further component:

\[ RT(i) = RT_L(i) + RT_R(i) + RT_D(i) \]

where \( RT_D(i) \) is the overlap component, the time spent using local and remote resources concurrently.

The solution in all cases is to divide the stateword for each interactive process into a local and a remote part, that is:

\[ RT(i) = RTL(i) + RTR(i) \]

where \( RTL(i) \) is the local component, \( RTR(i) \) is the remote component, and \( RT(i) \) is the response time for some interaction.

Local state, remote state, and overlap state are disjoint subsets of the set of interactive process states which, with think state, constitute a partitioning of the full set of states. Assuming that ifany is a function that returns true if any bit of the word given as its argument is non-zero, then the interactive process for user is:

\[ \begin{align*}
(1) & \text{ in local state ifany(statewords(user).local_part) and not ifany(statewords(user).remote_part)} \\
(2) & \text{ in remote state ifany(statewords(user).remote_part) and not ifany(statewords(user).local_part)} \\
(3) & \text{ in overlap state ifany(statewords(user).local_part) and ifany(statewords(user).remote_part)} \\
(4) & \text{ in think state not ifany(statewords(user).local_part) and not ifany(statewords(user).remote_part)}
\end{align*} \]

Figure 2 illustrates a distribution of statewords for workstations in a simple LAN. If remote resources are spread across several processors, the remote_part should be subdivided. Apart from the division of each stateword across different memories, the model should apply exactly as described earlier.

A Monitor
Running on an individual workstation, the Novell monitor was designed first to measure the local and remote components of response times, and the number of users logged in to the file server. The monitor was constructed using system calls, and a large-scale monitor was not possible because operating system source code could not be obtained.

The monitor is a very small implementation of the

```plaintext
type
local_resources = (LR1, LR2, ......... LRx); 
remote_resources = (RR1, RR2, ......... RRy); 
resourceBits = record
  bit, qbit: boolean;
end;
stateword = record
  local_part: array[local_resources] of resourceBits;
  remote_part: array(remote_resources) of resourceBits;
end;
var statewords: array[users] of stateword;
```

Figure 2. Distribution of a stateword across a simple LAN.
model with an interactive process state word of three bits, one showing use of local resources, another the use of remote resources, and the third showing think state. Despite the monitor's limitations, it has been useful for evaluations of two systems similar in all but their workloads. In each case the aim is to determine the balance between local and remote components, to estimate the effect on performance of possible server changes.

One case is a medical information system with a shared database on the file server, with interest being to record performance changes over a long period of time as the database grows. The remote component is at present changing from being smaller than the local component to being the dominant component. The second case is a local network used for large, first-year computing classes. Here, the remote component of response times was found to be very small, so that in this case the effect on individual performance of increasing the number of workstations is negligible.

8. CONCLUSION

The model procedure for data recording and measurement shows how systems should be instrumented to give an exhaustive decomposition of interactive response times. The model has also given insights found useful in designing sampling monitors. However, addition of instrumentation to running systems, though successful up to a point, has not been particularly easy.

Of the monitors described, the one for PRIMOS is the most substantial. It was written under conditions that were as near ideal as one could expect. System source code was available and, with a Prime 750 supporting users, a spare Prime 2250 was for a time available to install the sampling monitor as a high priority system process. Unix source code is available for the work on the Vax monitor, but the small monitor for the Novell system was written under the major limitation of having no system source code available.

For all monitors, it was sometimes found difficult to determine (in the terminology of the model) the state of the interactive process. One of many possible examples is that process control blocks on the Prime are often on the clock ring semaphores, but the event waited for often cannot be determined. It seems a shortcoming of systems design if the cause of a process being blocked — and therefore of a user's response time being extended — is under any conditions difficult, perhaps impossible, to determine by external sampling.

Our most important conclusion is that, as shown by the simplicity of the model, instrumentation of systems to measure components of response time should be very straightforward — if done at the time of design.

ACKNOWLEDGEMENTS

The authors are grateful to Tim Bell for his valuable comments. The Unix monitor, designed by Ashton and Penny, is being constructed by Richard Maurice. The cartographic software is being designed and constructed by Greg Ewing.

REFERENCES


BIOGRAPHICAL NOTES

John Penny obtained his PhD in Mathematics from the University of Adelaide, where he designed and constructed (what would today be called) the operating system for the CIRRUS computer, one of the world's first time-sharing systems. Since then, he has been inter alia a Senior Research Scientist for the CSIRO Division of Computing Research, an Associate Professor of Computing Science at the University of Alberta and since 1972, Professor of Computer Science at the University of Canterbury, where he was Head of Department from 1972-1982. His research interests are in the areas of computer systems, especially evaluation of interactive system performance, and computer cartography.

Paul Ashton graduated from the University of Canterbury with First Class Honours in Computer Science in 1984, and was awarded a UGC post-graduate scholarship, and the IBM postgraduate scholarship, of which only one is awarded each year in New Zealand. He worked with Adata Software in New York before returning to Canterbury as an Assistant Lecturer and PhD student. His PhD project is in the area of interactive system performance.

David Tripp graduated from the University of Canterbury with First Class Honours in Computer Science in 1986. He is at present completing a BA degree, majoring in Political Science.
Robot Navigation with Learning

A. Zelinsky†

Navigation is the most important function of an autonomous mobile robot. Research into mobile robot navigation has concentrated on the problem of finding optimal paths through known environments. In such cases the robot will need to learn and understand the environment. This paper will present an algorithm for learning in an unknown environment with a mobile robot. The robot will learn the environment by sensing while it is executing paths, which have been generated by a path planner. As obstacles are encountered en route to a goal, the model of the environment is updated and a new path to the goal is planned to avoid the obstacles. The paths generated by this system will be initially locally optimal, but as knowledge of the environment increases the optimality of the paths generated will improve until eventually global optimality is attained. The learning algorithm presented in this paper makes use of the quadtree representation to model the environment and uses the distance transform methodology to generate paths of navigation for the robot.

Keywords and Phrases: Mobile robots, path planning, environment learning.

CR Categories: E.1, I.2.6, I.2.9, I.2.10.

1. INTRODUCTION

The problem of finding optimal paths for robot manipulators and autonomous mobile robots through environments cluttered with obstacles has attracted much interest and research. A great deal of the research has concentrated on situations where the environment in which the robot operates is completely known and supplied to the path planner (Lozano-Perez and Wesley, 1979; Gouzenes, 1984; Brooks, 1983; Kuan, Zamiska and Brooks, 1985; Miller, 1985; Lozano-Perez, 1983; Brooks and Lozano-Perez, 1985; Schwartz and Sharir, 1983 and 1984; Udupa, 1977, Singh and Waugh, 1987; Kambhampati and Davis, 1986 and Kant and Zucker, 1986). The path planner finds a collision free path for the robot between the start and goal locations. Many path planners do not address the problem that path planning may require rotation as well as translation to negotiate a path to the goal. This problem has been referred to as the “Piano Movers” or “Couch Around the Corner” problem. Solutions put forward to this problem are computationally expensive (Brooks and Lozano-Perez, 1985; Schwartz and Sharir, 1983 and 1984); a popular solution is to approximate the robot as a cylinder. Path planners optimise a particular property whilst finding a path from the start location to the goal. Criteria for optimisation which have been used include the minimisation of path length, time to complete the path and energy required to negotiate the path or angle of deviation from the straight line path to the goal. For a robot navigating in a partially known or completely unknown environment these path planning techniques are not directly applicable. Path planners of this type have been modified to work in unknown or partially known environments. Difficulties arise in deciding how to treat the unexplored regions of the environment. A typical approach is to treat unexplored regions as obstacles, and only proceed into the unexplored regions if the goal lies there (Thompson, 1977; Chatila, 1982 and Moravec, 1980).

One of the problems with mobile robots operating in partially known or completely unknown environments is the updating of the environment map with sensor data. Sensor data is prone to noise and inaccuracy. This introduces uncertainty into the environment map and consequently makes the task of navigation more difficult. Similar difficulties are encountered matching sensor data with a robot produced map of the area, when a location has been revisited (Chatila and Laumond, 1985; Moravec and Elfes, 1985 and Crowley, 1985). A way forward suggested by Brooks (1984) is either to engineer away sensor inaccuracies or develop robust navigation algorithms which handle uncertainty. At present to develop and implement a navigation system for a mobile robot with current technology sensors clearly requires sophisticated algorithms.

Research efforts in environment learning by mobile robots have followed a variety of approaches (Thompson, 1977; Chatila, 1982; Moravec, 1980; Chatila and Laumond, 1985; Moravec and Elfes, 1985; Crowley, 1985; Brooks, 1984; Rao et al., 1986; Iyengar et al., 1986 and Thorpe, 1984). These can be broadly classified into two groupings; adaptive models and rigid models. Adaptive models use the nature and clutter of the environment to drive the representation of the environment. Adaptive models typically represent the environment as a network of free space regions (Chatila, 1982; Chatila and Laumond, 1985; Moravec and Elfes, 1985; Crowley, 1985; Brooks, 1984; Rao et al., 1986 and Iyengar et al., 1985) or as a graph of obstacle vertices (Thompson, 1977). Adaptive model environment learning methods require accurate sensor information. Rigid models impose a structure, such as a grid onto the environment without any regard to the nature and clutter of the environment (Moravec, 1980; Moravec and Elfes, 1985 and Thorpe, 1984). Adaptive model environment learning methods offer elegant solu-
tions, but in practice are difficult to implement. The converse can be said of rigid model environment learning methods.

A recent approach has considered path planning as finding paths from the goal location back to the start location ( Jarvis and Byrne, 1986). This method covers the environment with a grid, and generates all the paths from the goal location back to the start location. The paths are generated using distance transforms. To find the path of minimum length from the start to the goal, the grid neighbours of the start location are examined to find the grid with the lowest distance transform. This grid cell represents the start of a chain of cells of least distance to the goal. However this method does have drawbacks, namely the inefficiencies of a grid representation, the costly computation of the distance transform and the ambiguity of optimal minimum length paths.

Environment learning by a mobile robot can be accomplished in one of two ways. Learn the environment by putting the robot into a 'mapping' mode (Moravec and Elfes, 1985 and Crowley, 1985). In this mode the robot traverses the entire environment scanning it with its sensors and updates a map. The map is then used for all subsequent path planning exercises. Difficulties arise with this method if the environment is allowed to alter after the mapping has been completed. The other mode of learning is to sense the environment, while executing paths which have been generated by a path planner. As obstacles are encountered en route to a goal, the model of the environment is updated and a new path to the goal is planned to avoid the obstacles ( Rao et al., 1986 and Iyengar et al., 1986). The paths generated by this system will initially be locally optimal, but as knowledge of environment increases the optimality of the paths generated improves until eventually global optimality is attained. Environment learning methods of this type have difficulty in detecting that a goal is not reachable since they use heuristics when planning locally optimal paths. The restriction that the robot can recognise line of sight distances to obstacles and detect their edges without imprecision makes it hard to implement this learning algorithm with current sensor technology. Since sensing line of sight to obstacles can not be guaranteed an arrangement of obstacles in an environment can always be found where a heuristic path planner will fail (Cahn and Phillips, 1975; Chattergy, 1985).

A good compromise between adaptive free space models and rigid grid models are hierarchical planning methods. One such approach has been to use quadtrees ( Kambhampati and Davis, 1986). This approach assumed a map of the environment was supplied to the path planner. Quadtree's have the advantage of having a rigid structure, but are adaptive to the clutter of the environment. This paper will present an environment learning algorithm using quadtrees to model the environment and a modification of the distance transform methodology to generate paths of navigation for the robot which can be used with current sensing technology. Attention in this paper will be restricted to two dimensional path planning without rotation. Most path planners approximate the robot to be a cylinder, then shrink the robot to a point and expand all the objects in the environment by the robot's radius. This strategy is useful in known environments, but in unexplored environments this represents unnecessary processing, and important information is discarded. Namely the volume in which the robot is located and the volumes of the paths swept by the robot through the environment are definitely free space in a static environment and are probably free space in a dynamic environment. In learning situations the information a mobile robot requires is distances to obstacles, and then it is able to decide whether or not it can pass through a gap between obstacles.

2. PHILOSOPHY OF NAVIGATION BY LEARNING
To develop a navigation algorithm for learning an unexplored environment, it is often useful to study how humans navigate in unexplored environments. Consider a person in an unfamiliar city, who asks for directions to a particular destination. The reply the person may receive is that the destination is a certain distance in a certain direction. The person will then proceed to the goal by the most direct route. If the person is fortunate the goal is achieved without deviation from the planned path. However this is rarely the case, obstacles both stationary and moving will be encountered. Should an obstacle be met, during the course of the journey to the goal, the person classifies the object as either stationary or moving. If the object is stationary the person notes the location and features of the obstacle, whilst for moving obstacles the person notes only the features of the obstacle. At this point the person revises the original plan in light of the new information. If the object which is blocking the person's passage to the goal is stationary, the revised plan will generate, based on the person's knowledge of the environment the best shortest path around the object towards the goal. For the case of the moving obstacle, assuming the person is patient and polite, he or she will wait until the moving obstacle is out of the way. Such a strategy is a safe one, particularly in potentially dangerous situations, such as crossing busy streets. If a moving object comes to rest for a significant period of time, then it too can be treated as a stationary object. This process of planning and executing the plans continues until the person either reaches the goal or deduces that the goal is not reachable. For example realising that all the routes to the goal have been cut off due to flooding.

Should the person need to travel to the same goal again, then one of two approaches can be used. The person can proceed to the goal using the knowledge of the environment learnt on previous journeys. This approach will optimise the paths which were previously executed, and would be favoured if the person is in a hurry or is not in an adventurous mood. If a significant period of time has transpired since the previous journey then the person may have forgotten much of the information about the structure of the environment. In this case the person may wish to relearn the environment en route to the goal will be much quicker than the original learning of the environ-
Robot Navigation with Learning

The person will begin to remember the details. The other approach the person may take is not to hurry and to explore alternative paths to the goal, and in doing so to learn more about the environment.

3. NAVIGATION LEARNING DATA STRUCTURES

The data structure which is central to this navigation algorithm is the quadtree. The definition of quadtree used in this paper follows the definitions given by Kambhampati and Davis (1986). A quadtree is a recursive decomposition of a two-dimensional region into $2^i \times 2^i$ square blocks. A node of the quadtree represents a $2^i \times 2^i$ square portion of the region. The nodes of the quadtree can be classified as free, obstacle or grey nodes. Free nodes represent portions of the region which are free space. Obstacle nodes represent portions of the region which are occupied by obstacles. Grey nodes represent areas which are a mixture of free space and obstacles. A leaf node of the quadtree is a node which has no descendents. Refer to Figure 1 for an example of a two-dimensional environment and its quadtree representation. Each leaf of the quadtree stores a measure of the confidence that the navigation system has in this node belonging to a particular class e.g. 90% confidence that this node is a free node. The node confidence is updated by the navigation system every time the robot visits or observes the node. Storage within the leaf node is also provided for the distance transform value which is used in path planning.

The leaf nodes of the quadtree can be connected horizontally and vertically (four connected) or can be considered to be also diagonally connected (eight connected). Considering paths to be four connected avoids paths which clip obstacles (Kambhampati and Davis, 1986), but the penalty for this is that the paths are not optimal, and possible solution paths are excluded, refer to Figure 2. This paper will consider the leaves of the quadtree as four connected and a method will be presented which outlines a solution to the problem illustrated in Figure 2.

4. NAVIGATION LEARNING ALGORITHM

The navigation learning algorithm requires two pieces of information upon initialisation; the notional size $m \times n$ of the environment to be learnt, and a position reference $(x, y, \theta)$ of the cylindrical robot of diameter $d$. A quadtree $Q$ of sufficient size is generated to cover the area in which the robot will operate. The smallest quadtree leaf resolution size is of size $d$, a size which allows the robot to pass through. The size of the quadtree $Q = (2d)^i$, where $i$ is an integer such that $(2d)^i > s$ and $s = \max(m, n)$.

Given the start and goal locations, the best navigation strategy in an unknown environment is to assume that the unknown regions of the environment are free space; the confidence in this assumption is low. The navigation system invokes a path planning process, which plans a straightline path to the goal. The path generated by the planning process is passed onto the path execution process,
which ensures the planned path is executed by the robot. The robot proceeds cautiously toward the goal. One of two conditions will occur; either the robot reaches the goal or it encounters an obstacle. If the goal is attained then the path execution process reports the success to the navigation procedure and the free space confidence that the navigation system has in this path is updated. In the case of an obstacle blocking the robot's path, the path execution process returns the position and orientation of the robot and the robot's sensor readings at this location, to the navigation system. The navigation system invokes the navigation update process, which, given the robot's location and sensor readings, updates the environment model Q i.e. the quadtree structure, and the confidence model of the environment. Upon completion of the update of the environment and confidence models, the path planning process is invoked again. Essentially the path planning process locates the leaves of the quadtree where the current location and goal are found and then applies the distance transform methodology to generate a solution path, or deduces that no path exists. The revised plan is then attempted by the robot. This cycle of plan — execute — update continues until the robot reaches the goal, or deduces that the goal is unreachable. The general algorithm is given in the procedure NAVIGATION.

```
procedure NAVIGATION (Q, start, goal)
    repeat
        cost = 0
        perform PATH_PLANNING (Q, goal, cost, nil)
        if (goal reachable) then
            perform PATH_EXECUTE (Q, start, goal, location, sensors)
            if (location = goal) then
                perform MODEL_UPDATE (Q, start, location, sensors)
                start = location
            end if
        end if
    until (location = goal or goal not reachable)
end procedure
```

5. PATH PLANNING ALGORITHM USING DISTANCE TRANSFORM
The use of distance transforms to generate path planning solutions for mobile robots was recently presented in a paper by Jarvis and Byrne (1986). The path planning procedure used in this method covers the environment with a uniform grid, and then propagates distances through free space from the goal cell (cells are assumed to be 8 connected). The distance wave front flows around obstacles and eventually through all free space. For any starting point within the environment representing the initial position of the mobile robot, the shortest path to the goal is traced by walking down hill via the steepest descent path. If there is no downhill path, in other words if the start cell is on a plateau then it can be concluded that there is no path from the start cell to the goal cell. (Note: all the cells are initialised to high values.) Refer to Figure 3 for an example of the distance transform. The disadvantages with this approach are the inefficiencies of a grid approach, which leads to costly computations of the distance transform through free space, and the ambiguity of optimal paths caused by considering diagonal neighbours to have the same cost as vertical and horizontal neighbours.

The algorithm presented in this paper applies the distance transform to the quadtree structure. In this case distances are propagated from the goal quadrant leaf to the neighbouring quadrant leaves. The distance used is the straight line distance between the centres of the quadrant leaves, except in the case of the goal quadrant leaf where distance transforms are calculated from the goal to the centres of the neighbouring leaves. This approach improves the storage efficiency of the distance transform.

Figure 2. On a Four Connected grid, it is possible to miss diagonal paths. Here there is enough room for the robot to pass, but there is no clear path on the grid, since only vertical and horizontal paths are considered. If the grid is Eight Connected, a solution path can be found to the goal by considering diagonal paths in addition to vertical and horizontal paths.

Figure 3. The distance transform applied on a regular sized grid. To find a path to the goal, locate the start location in a grid cell and then descend down hill starting with the neighbour with the least distance transform. Ambiguity of optimal paths exists where there exist two or more cells to choose with same least distance transform.
and does not generate any ambiguous optimal paths. The general algorithm for distance transformation is given in the procedure PATH PLANNING. Assume that the function EXTRACT gets the distance transform cost currently stored in this leaf. A new cost is only stored if a freshly generated distance transform cost is less than the cost currently stored in the leaf. The procedure STORE records the cost of the distance transform. The procedure GET_NEIGHBOUR finds the next neighbour leaf to the current leaf in a particular direction. If more than one neighbour exists then they are accessed in EAST-WEST or NORTH-SOUTH order. The function DISTANCE calculates the distance between the centre of the leaf and the centre of the neighbouring leaf. The constant FACTOR is used to give a weight to the distance transform depending on the leaf type; free, obstacle or unknown. Jarvis and Byrne (1986) used the largest possible integer for obstacles as the FACTOR value, to ensure that no obstacle cells were included in the solution. A FACTOR of 1 was used for known and unknown cells. However the FACTOR could be varied to induce different behaviour. If an inquisitive behaviour was required the FACTOR in known cells was doubled, if on the other hand a behaviour was sought where the robot avoided unknown cells, the FACTOR in unknown cells was doubled.

procedure PATH PLANNING (Q, leaf, cost, predecessor)
minimum = EXTRACT(leaf)
if (cost < minimum) then
perform STORE(leaf, cost)
for direction = EAST, WEST, NORTH and SOUTH do
perform GET_NEIGHBOUR(Q, leaf, direction)
do while (neighbours exist)
if (neighbour != predecessor) then
newcost = FACTOR * DISTANCE(leaf, neighbour) + cost
perform PATH PLANNING(Q, neighbour, newcost, leaf)
end if
end do
end if
end procedure

This path planner when it is navigating in an environment uses the FACTORS set out in the following rules. In all modes of navigation the obstacle leaf FACTOR is considered to be FACTOR = VERY_HIGH_NUMBER*OCCUPANCY, where OCCUPANCY is a measure of how much of the leaf is occupied by an obstacle. If the robot is in a learning mode, free spaces and unknown spaces are considered to be of the same leaf class, since the best assumption that can be made of unknown space is that it is free space. In this case FACTOR = 1. Once the robot has learnt a portion of the environment, there could be a need to find the optimal path to a goal using the free spaces in which the navigation system possesses the greatest knowledge. In this case FACTOR = 1 + [1 - CONFIDENCE], where 1 - CONFIDENCE is a measure of the confidence the system has in a leaf NOT being free space (unknown spaces are free spaces with zero confidence in this assumption). Alternatively the robot may operate in an explorative frame of mind and favour unknown spaces en route to a goal. In this case FACTOR = 1 + CONFIDENCE, where CONFIDENCE is a measure of the confidence the system has in a leaf being free space. The robot may navigate in a mapping mode which attempts to learn all the unknown areas, this can be accomplished by making the free space quadrants with the lowest confidences the goals.

The distance transform algorithm can be implemented recursively or it can be implemented concurrently. In the concurrent implementation a separate process is spawned for the eastern, western, northern and southern neighbours of a given leaf; conceptually this is equivalent to dropping a stone into a pond and watching the ripples radiate out. Refer to Figure 4 for an example of the algorithm.

6. PATH EXECUTION ALGORITHM
Upon the completion of path planning, the navigation system is ready to execute the planned path. The path execution algorithm must first isolate the location of the robot into a leaf of the quadtree Q. The distance transform of this leaf must be examined. If the distance transform > 0 then this means that start and goal points are located in the same leaf, therefore the robot can proceed directly to the goal. If the distance transform > 0 then the vertical and horizontal neighbours of the leaf with the minimum distance transform must be found. The leaf with the minimum distance transform becomes a subgoal, the robot proceeds to the subgoal. The robot will either reach the subgoal or it will encounter an obstacle. If the robot reaches the subgoal, the free space confidence between the start and subgoal is updated. Upon updating of free space confidence the next subgoal is found and the path to this subgoal is attempted. This process is applied repetitively until the goal is attained or an obstacle blocks the robot's path. If an

![Diagram](image-url)
Figure 5. Four examples of path execution using different strategies. In R1 the strategy is to always steer through the middle of the quadrants and pass over the centroid of the quadrant. In R2 the strategy is to always steer through the middle of the intersection of edges which link the quadrants via entry and exit points. In R3 the strategy is to look ahead and line up on the entry and exit edges of quadrants further ahead in the planned path. In R4 the strategy is to smooth zig-zag paths created in four connected grids by steering down diagonal paths.

Figure 6. To generate optimal paths the strategy is to look ahead to the middle of the exit edge of next quadrant leaf and then find the shortest-path between this point and the current position of the mobile robot, such that the path passes through the exit edge of the current quadrant and the entry edge of the next quadrant. This optimisation procedure is repeated every time the robot enters a new quadrant leaf. In the above example, * marks the place where the optimisation calculation occurs. The broken lines mark the boundaries where the centre point of the robot can not cross. This ensures that the robot stays entirely within the boundaries of the quadrant. Combining the optimisation procedure with the zig-zag smoothing rule, can produce superior optimal paths. The heavy broken line shows the path generated by the zig zag rule.

7. MODEL UPDATE ALGORITHM

Once the robot has completed path execution, the navigation system is ready to update the environment and confidence models of the environment. Path execution terminates on one of two conditions; either the robot reached the goal or an obstacle was encountered. If the robot reached obstacle is sensed, path execution is terminated and the robot location and the sensor values are returned to the navigation system.

The process of updating the free space confidence of a quadrant leaf is done by calculating the area swept by the robot in the leaf and dividing it by the area of the leaf. This free space confidence value is added to the confidence value previously stored at this leaf, with weight given to the most recent navigated path. The free space confidence value could also include the free space detected by sensors on board the robot using the approach followed by Moravec and Elfes (1985). However due to the uncertainty of sensor readings, sensor free space confidence should be given less weight than confidences generated from areas visited by the robot.

During path execution a number of strategies can be used to generate a path through the free space quadrants, these are summarised in Figure 5. Clearly the strategy of executing paths by ensuring that the robot always passes through the centre of quadrants, generates inefficient paths. A more reasonable strategy would be to steer through the middle of the intersection of entry and exit edges of quadrants. Such a strategy generates relatively “safe” paths, and would be useful during the exploration or learning of an environment by a mobile robot.

Once an environment is well known and the confidence in the free space quadrants becomes high, the robot should find optimal paths rather than safe paths. One way of generating optimal paths is to look ahead one quadrant and find the optimal path between the current location and the mid point of the exit edge of the next quadrant. The robot proceeds toward the aim point, until the robot enters the next quadrant where it stops and repeats the procedure. Refer to Figure 6 for an example of this path smoothing. The optimality of the paths can be improved by looking ahead more than one quadrant, but this can become computationally expensive.

This strategy produces efficient paths, however in the cases where the robot must steer through quadrants of the smallest size resolution, the nature of four connected quadrants gives a zig-zag path. One way to counter this is by using the following rule; If the quadrant leaf is of the smallest resolution, the entry and exit edges are normal to one another and the neighbouring quadrant is free space, then the stop point required to navigate through this quadrant can be omitted. This causes the robot to steer diagonally between the free space quadrants. By combining the optimising procedure and the zig-zag smoothing rule reasonably optimal paths can be produced.
the goal, then the environment model does not change, and the confidence that the environment is of the current model structure is increased. If an obstacle is encountered the environment model must be updated to reflect the presence of the freshly sensed obstacle and the confidence in the environment is also updated. To perform the update of the models, the algorithm must know the current location of the robot, the sensor readings at this location, and the location of the robot when this procedure was last invoked.

Once the algorithm updating the environment models is invoked, the current leaf location of the robot in the quadtree is found. This leaf is checked to make sure that none of the obstacle sensor readings occur in the leaf quadrant. If the leaf is not isolated from the sensor readings then the leaf quadrant must be divided into four quadrants. Leaf division is continued until the leaf is sensor isolated from sensor data or the leaf reaches the smallest size resolution allowable. Every time a leaf is divided, the free space confidences of the new leaves must be calculated. This is done by fitting a line between the current and previous robot locations, determining what portion of the line cuts the newly generated leaves and then dividing this portion by the size of the new leaf. Upon the completion of sensor isolation of the current robot location, and the updating of the free space confidences, the sensor readings detected by the robot are each in turn isolated to a leaf of the smallest size resolution, this may require the subdivision of leaf quadrants and the updating of free space confidences. Once a sensor reading has been isolated to a leaf, an estimate of what proportion of the leaf is occupied by the obstacles is made and recorded. By isolating the sensor reading to the smallest resolution leaf, this may be seen as fragmenting the quadtree unnecessarily. A consolidation procedure is needed which can detect that neighbouring occupied leaves of a quadtree are part of the same obstacle and thus the quadtree can be pruned back to the parent of the leaves. The consolidation procedure can work in conjunction with the navigation system when the robot is operating in a mapping mode. When the robot is navigating in a mapping mode, it places goals in the centre of the leaf quadrants which have the lowest free space confidences. If the path planning procedure deduces that a goal is not reachable, this implies that the goal is surrounded by obstacles. The consolidation procedure can then prune the quadtree, to reflect the knowledge that the region surrounding the goal is one obstacle. Refer to Figure 7 for an example of the consolidation algorithm.

![Diagram A](image1)

![Diagram B](image2)

**Diagram A**

**Diagram B**

Figure 7. The Consolidation Algorithm consolidates blocked quadrants and unreachable quadrants of free space. Diagram A shows an environment prior to consolidation and Diagram B shows the same environment after consolidation.

8. PARTIAL DISTANCE TRANSFORM UPDATE ALGORITHM

Computing the full distance transform every time the robot encounters an obstacle can be regarded as an unacceptable computational burden. Considerable savings can be made if the distance transform is only partially updated when the environment model changes. The quadtree modelling of the environment can be elegantly used to limit the recomputation of the distance transform.

If the robot start and goal locations can be found in the same minimal subtree of the quadtree, then the distance transform need only be calculated for the subtree. Considering a subset of the possible solution paths will yield either locally optimal paths or no solution paths. Finding locally optimal paths is acceptable when the robot is learning an environment. To find globally optimal paths or if no solution paths can be found locally, simply move up one level in the quadtree structure and compute the distance transform for the fresh subtree.
Such a strategy is useful when the robot gets closer to the goal, however it does not limit the recomputation of the distance transform when the start and goal are separated by large distances. It is still possible to limit the recomputation to a partial update in such instances. When a change to the quadtree model of the environment occurs, isolate the change to a minimal subtree. The distance transform of this subtree either affects or is affected by its neighbours. Find the minimal subtree which isolates the subtree of change and its neighbours. If the goal is located in the minimal subtree then calculate the distance transform for this subtree only. Otherwise recompute the distance transform from all the neighbours of the subtree, but only in the direction of the minimal subtree. The general algorithm for partial distance transform update is described in the procedure PARTIAL. Assume the procedure ISOLATE finds the common subtree of two leaves. Assume the procedure PARENT finds the parent of the subtree SQ within the quadtree Q. The partial distance transform update mechanism can be incorporated into the navigation system. The modified navigation algorithm is given in the procedure NAVIGATION_2. For an example of the navigation algorithm refer to Figure 8.

```
procedure NAVIGATION_2 (Q, start, goal)
    SQ = ISOLATE(Q, LOCATE(Q, start), LOCATE(Q, goal))
    repeat
        perform PARTIAL(SQ, goal)
        if (goal reachable) then
            perform PATH_EXECUTE(SQ, start, goal, location, sensors)
            if (location = goal) then
                perform MODEL_UPDATE(SQ, start, location, sensors)
                start = location
            endif
            SQ = ISOLATE(SQ, LOCATE(SQ, start), LOCATE(SQ, goal))
        else
            SQ = PARENT(Q, SQ)
        endif
        if (SQ = nil) then goal is reachable
    until (location = goal or goal not reachable)
end procedure
```

```
procedure PARTIAL (Q, goal)
    SQ = nil
    for i = 1 to all changes in Q do
        perform ISOLATE(Q, SQ, change(i))
    endfor
    for i = 1 to all neighbours of SQ do
        perform ISOLATE(Q, SQ, neighbour(i))
    endfor
    perform LOCATE(SQ, goal)
    if (successful) then
        perform PATH_PLANNING(SQ, goal, 0, nil)
        for direction = EAST, WEST, NORTH and SOUTH do
            for i = 1 to number of SQ direction leaves do
                perform GET_NEIGHBOUR(SQ, leaf(i), direction)
            endfor
            cost = EXTRACT(leaf(i))
            newcost = FACTOR * DISTANCE(leaf(i), neighbour) + cost
            perform PATH_PLANNING(SQ, leaf(i), newcost, neighbour)
        end for
    end if
end procedure
```

9. FURTHER WORK
The environment learning algorithm presented in this paper is a learning mechanism for a static environment. An extension of the algorithm to handle dynamic environments is currently the subject of further investigation. One way the learning of dynamic environments can be accomplished, is to add a forget function to the navigation system. The forget function periodically reduces the robot's knowledge of free space by a percentage factor. This percentage factor could be varied through the quadtree depending on where the robot is currently located. Leaf quadrants which are in close proximity to the robot should be less easily forgotten than those further away. Such a function would have the effect of forcing the robot to relearn portions of the environment it has forgotten, and
hence the robot would be able to operate in a dynamic environment.

By considering the lowest size resolution of the quadtree to be the same size as the robot, this could exclude possible paths if a leaf is only partially occupied. Further investigation is warranted and possible approaches would be to further apply the quadtree division of space, but have exceptions to the rule that a robot can only travel within a quadtree leaf. Another possibility is to "relax" the quadtree and move the quadrant leaf a sufficient amount until it is in free space, and then attempt a path through the overlapping quadrants. Further work needs to be done on smoothing the paths the robot executes. It is inefficient to stop the robot every time it needs to change direction. Techniques are being developed to generate smooth continuous paths for execution by the robot.

10. CONCLUSIONS

Quadtrees and distance transforms provide an effective mechanism for learning a static environment with a mobile robot. The environment can be efficiently modelled with quadtrees, and distance transforms can be applied to explore paths in known and unknown portions of the environment. The problem of expensive computation of the distance transform is addressed by concurrent processing, and limiting the recomputation of the distance transform. Mechanisms have been provided for the efficient execution of planned paths. It can be seen that by varying the manner in which the distance transform is generated, different behaviours can be induced from the navigation system, such as explorative, learning and finding optimal path behaviours.

11. REFERENCES


BIOGRAPHICAL NOTE

Alex Zelinsky has been with the Department of Computing Science at the University of Wollongong, as a Lecturer, since 1988. He completed his Honours degree in Computing Science in 1983 at the University of Wollongong. Currently he is completing his PhD in Computing Science at the University of Wollongong. From January 1977 until February 1984 he was a Computer Systems Engineer with BHP Steel International. He was a Lecturer in Information Systems, in the Faculty of Commerce at the University of Wollongong from February 1984 until January 1988. His research interests are in the areas of: planning and learning applied to mobile robots, knowledge representation, and geometric and spatial reasoning.
**Book Reviews**


Two headlines on adjoining pages of a recent issue of the *Australian Financial Review* neatly state the main claims of Gene Gregory's book: "Research points to Japan as world's best at innovation"; and "Newly industrialising Asian countries challenging Japan". Although the author overstates the first point and underplays the second, his book is nonetheless a readable, informative and comprehensive overview of the Japanese electronics industry.

*Japanese Electronics Technology* consists of 30 articles written during the first half of the 1980s. There is surprisingly little overlap between the essays. The most interesting deal with the evolution of corporate strategies and government policies for promoting the development of Japan's now mighty electronics industry. Gregory reminds us just how much ground the Japanese have gained during the past two decades, particularly in the areas of consumer electronics, semiconductor devices and factory/office automation equipment. While paying due attention to the government's role in structuring and promoting local electronics firms, the author quite rightly attributes most of these organisations' success to their own commercial, manufacturing and technological expertise. In particular, he pays homage to the human resources policies of such corporate giants as Hitachi, Sony and Toshiba. Their commitment to hire armies of computing and engineering graduates, train them over many years in all facets of business and select as leaders only the most technically and organisationally competent managers has been, in Gregory's view, the key to their success. These are the men (for there are no women to be found in the boardroom) who have given the Japanese an unrelieved and still often unacknowledged capacity to 'innovate', i.e. to match successfully a technical possibility to a market opportunity.

Thebulk of Gregory's book is devoted to case studies illustrating his general thesis about the dimensions of and reasons for Japan's achievements in electronics. There are chapters on the fortunes of the local computer manufacturers, the growth of the computer services industry and the government's 'fifth-generation' project, as well as the more familiar stories of exponential sales increases in microwave ovens, VCR's and VLSI devices. Each article is buttressed by a wealth of statistical material, much of it produced by the Japanese Electronics Industry Association. Where Japan has not already shown itself to be omniscient in electronics, the author is ready to argue the toss that it soon will be.

Like many proponents of a good argument, Gregory ends up overstating his case. Indeed since his book has been published, the Japanese electronics industry has gone through a difficult period of reduced growth and sharp declines in profitability. The strong yen, fears of a trade war and increased competition from low-cost producers in Southeast European firms, have put Japan on notice that further gains in its global share of the world electronics market will be much harder to come by. However, as Gregory would be the first to point out, Japanese corporations are now prepared to grow through overseas investment and joint ventures, rather than relying exclusively on their own local resources. Whether or not these adjustments are successful, *Japanese Electronics Technology* can be highly recommended, especially to anyone interested in industry policy and corporate strategy, for its documentation of a great industrial success story.

---


About eight years go there was a man with a problem. He didn't want to program in 8080 assembler any more (and who can blame him?). So . . . he wrote himself a C compiler. Thus was Small C born. The source for the first (8080 only) version was published in *Dr Dobb's Journal* in 1980; here an updated version (2.1) is made available. An 8086 version is available in machine-readable form for the IBM-PC and compatibles. In the book also are various 'software tools' like the ones in Kernighan and Plauger (1976) (except, of course, for being written in C rather than RATFOR or PASCAL), a macro assembler for the 8080/280, several C-related articles taken from *Dr Dobb's* between September 1983 and February 1984, and a reprint of Ritchie, Johnson, Lesk and Kernighan's article *The C Programming Language* (1978).

The level of discussion is suitable for people who find the basic beginner's books for C too basic, and who want to move on. Programs are not described in full; one has to read the source to find out how a program works. Reading moderately sized well written programs is often a good way to learn the language they are written in, especially if one of those programs is a compiler for the language. The programs in the book are in the main well written, and are more useful than most example programs given in programming texts. What is more they are available in machine readable form for MS-DOS and CP/M computers — there is an order form in the back of the book.

The one program (a cross reference generator) I tried typing in and compiling had a couple of errors. Both were fairly obvious: a missing header file (style.h), and two transposed strings in a data array. The contents of the header file could be constructed fairly easily. The other error was not fatal, and merely caused the keyword void to be included in the cross reference listing produced by the program.

Apart from the software tools and the 8080 assembler, all the chapters have been published in *Dr Dobb's Journal*. There are some advantages in having the chapters together — there is no need to hunt through back issues looking for errata and corrigenda, and no 'continued on page 75' to contend with. Moreover, the original articles contained listings of programs in variant dialects of C, and so used non-standard library calls — these have been altered in the book to conform to Kernighan and Ritchie (1978).

Overall, the book will appeal to people who want to read C programs, and people who want to use the software in the book.

**References**


Peter Chubb
University of New South Wales


A great deal has been published about C and Unix and one is tempted to say 'yet another book on C and Unix'. It is quite difficult to choose among the wide range of books offered by various publishers on the subject. The title may appear attractive sometimes, and this is the case for this book, as it suggests a system programming view of the presentation of the C language. Unfortunately, the title is slightly misleading. Also, the author did not design the material as an introductory text or a manual for the practitioner but rather as a reference text.

Despite an attempt to dedicate the second chapter to the beginner 'who has never written a C program' the book is obviously intended for experienced programmers and computer professionals. A good working knowledge of C and Unix is desirable before reading the text. The third chapter is dedicated to the syntax using the metalanguage. This could simply join the appendices, as the notation is not heavily used throughout the text.

Chapters Four to Nine cover the language quite extensively, though the presentation lacks originality and does not escape the traditional sequence: Variables and Storage Classes, Operators and Expressions, Statements, Functions, Pointers and Structures. Nevertheless this constitutes a good 'walk through' the C language and the experienced pro-
grammer without knowledge of C should grasp its spirit very quickly. There are few examples scattered through the text, often too short or too simple. The beginner who has not lost his way yet, could probably still survive by struggling with the pieces of code.

In Chapter Ten, the author justifies the title by tackling Input/Output and file access in C language under Unix. Not being familiar, even at a user level, with the Unix operating system definitely becomes a handicap and the chapter merely describes the standard Input/Output Library. The examples have become scarce or have totally disappeared when they are necessary for the understanding of the non-conventional manner C and Unix perform I/O. Chapter Eleven attempts to summarise all the other functions 'to make programming in C easier and much more comfortable' as the author himself states. What one would expect to be the core of the book, namely the interfaces with the operating system, is the six-page-long Chapter Twelve. Programming style which is the backbone of a structured approach to C is reduced to a one and a half page list of rules with little explanation and no examples at all.

The content improves a lot in Chapter Fourteen, coined 'operating the programming environment' and Chapter Fifteen about testing C programs where methodology and Unix tools are entwined and reasonably described. Nevertheless, the text is not very comprehensive and important sections on security and protection, efficiency and performance seem to have been omitted. The appendices are valuable as good reference tables for error messages and functions. I would not recommend the book to the specialist who has access to a collection of well written manuals (some manufacturers do write good manuals!).

C. Cheneau
University of Technology, Sydney


This volume contains some 40 of the papers and working group reports from the conference of the same name held under the auspices of IFIP TC9/WG9.1 in Berlin in May 1986. These include twelve papers that formed plenary addresses. An earlier meeting, held in Italy in 1982, concluded that there was a need to support the professional and scientific communities working with the general issue of 'Computers and Society' and the current conference was intended to provide such support.

The papers grapple with the notion of 'participation'. Several definitions of the term are provided and a number of alternative structures for involvement of the users in the specification, design and development of information systems are given. It is worth noting that while there is consensus on the thrust of these definitions, there are significant differences between them. In some cases this can be ascribed to differences in the environment. There is, however, more to it than this. The conference shows that all is not yet agreed about how to handle these matters.

The book focuses on the social aspects of participation, and the specifics of computing science procedures for data analysis and similar topics rate scarcely a mention. Any reader who is interested in identifying options for user involvement or who wants to know what has been done and how successful it was, will find this book valuable. A substantial number of the papers sketch case studies or provide the result of field work. These are typically expressed in terms of a formalised theoretical framework. The investigations were in all cases limited and thus the results tend to provide corroboration of expected behaviour rather than statistical measures of confidence. A limitation on the number of cases is a typical feature of such work and thus this should not be viewed as a criticism.

All authors came from Europe. Three quarters were associated with academic institutions, and most of the remainder were employed by government or statutory authorities. There appears to be no paper from the private sector.

The typical paper length is 10-15 pages. Such a space is not sufficient to allow extensive detail on case studies. The references can thus be important, but here there is a problem unless one is fluent in several languages besides English. This difficulty is compounded as many of the cited items are departmental reports that are not widely available, so that follow up will not be practical for many of the papers.

The book pages appear to have been photo-reduced from the conference papers. This technique can work well when there is strict control on the original typescript. Here it does not work. Some papers were originally in a small point size and further reduction presents unanticipated challenges. In other cases the paragraph layout causes the problem. The worst example of poor presentation occurs in the opening paper where there are five pages of filled-in 'a', 'e', 'm' and 'w's. As English was clearly not the first language of several authors, there are many cases of unusual sentence construction and in correct punctation. This compounds the problems of poor presentation. Some papers are very hard work!

In summary, the book treats an important topic. Readers concerned with system design and the relationship between computing and social issues will find much to reflect on. The answers to many of the questions have not yet been determined but trends are starting to emerge. This collection of papers shows the current state of grappling with the issues.

John Hiller
University of New South Wales


The operative words in the title of this book are 'Software Tools' as the book is effectively an extension of the work of Kernighan and Plauger (1976) to the field of numerical software. It assumes very little knowledge of programming in C and almost the entire first chapter is devoted in an introduction to the language (47 pages). The tools developed in this book illustrate a philosophy as well as forming a tutorial on the C language. The use of C for developing the software tools is justified, in part, by the statements 'FORTRAN still lacks some features which are essential in the development of numerical tools' and 'FORTRAN ... has only two types of data structures, scalars and arrays, neither of which is sufficient for organising access to numerical data'. I think there are a lot of FORTRAN users who would argue strongly, and be able to demonstrate quite powerful numerical programs, against these statements. In the introduction the author states 'Access to a machine running the UNIX system or a UNIX-like operating system, while not essential, will simplify some of the programming'.

There are chapters on the topics of vector and matrix tools, optimisation and non-linear functions and solving differential equations. These chapters generally contain elementary methods and leave the more complex methods as exercises through references. For example the chapter on matrix tools has no tool for obtaining the inverse of a matrix. The inclusion of a chapter on graphics (48 pages) seems quite out of context in the development of numerical tools but the graphical techniques are used in later chapters to illustrate the output from some of the solution techniques. The level of complexity of the numerical techniques is illustrated by the level required for the understanding of the material - 'exposure to calculus and a first course in differential equations' and 'simple matrix algebra'. The chapter on non-linear functions extends the software tools philosophy to the socket wrench analogy where the correct piece (subroutine) must be attached (linked in) before the tool will work.

This book is clearly not a survey of numerical software. However, it is an exposition of a philosophial approach and a further contribution to computing usage with the aid of software tools.

References


Glynn W. Peady
Australian Nuclear Science and Technology Organisation


This book is the proceedings of the First Australian Online Information Conference held in Sydney, 20-22 January 1986. As the title indicates the conference dealt with issues in Online Information Storage and Retrieval.
Not surprisingly given the broad scope of the conference, the 41 articles included addressed a wide variety of topics including the impact of new technologies on online systems, marketing online systems, user responses to online systems, training of online users and statistical results of user and system surveys.

The main value of the book lies in the fact that the articles deal specifically with online systems in the Australian environment. This is particularly important because many of the articles discuss the implementation and marketing of online systems in industrial applications, for example in the business, health and legal industries. Obviously these articles would be of much less value if they were not set in the Australian context. Online users will also find useful the emphasis on the practical rather than the theoretical or technical aspects of online systems. Therefore this book would be helpful as a reference to library and information science specialists and in general to anyone involved with online systems. However, I doubt individuals would like to add this book to their personal library rather than consult it at their local library. Academics especially will find the coverage of technical issues unsatisfactory and many other people would object to those few authors who seem to be a bit too eager to promote the products of the companies they represent.

As usual the individual articles submitted to the conference varied substantially in quality. The conference opening and the keynote address deserve special attention because they provide a good introduction to information retrieval. Following is a list of the main sections of the book so that potential readers have an idea of whether or not they would profit from checking this book: electronic publishing; marketing online services, training and education, database critiques, business databases, health and medicine databases, videotex, legal databases, database developments, searching techniques, and microcomputer developments.

A. Laribi
Macquarie University


If you want to build your first expert system, don’t buy this book. It won’t help. However, if you’ve built an expert system and are dissatisfied with the limitations of rule-based technology then you might benefit from ‘reading it.

A cornerstone of expert system mythology is the ‘Knowledge Principle’: if you put enough knowledge into a system, no matter how simple its inference mechanism, then you will achieve expert performance in your selected problem domain. Slatter’s book is a review of some recent work drawn from cognitive psychology and expert system research. It emphasises problem solving strategies rather than knowledge representation or acquisition. As such it is a refreshing slant on the area.

Slatter attempts to outline the role of ‘Cognitive Emulation’ as “a strategy in expert system design which seeks to emulate human thinking”. As one might expect, this covers a large area of research and often only tentative conclusions can be drawn from the literature. The field is still young. Consequently the book aims to cover as much territory as possible at an introductory level. This is not a weighty text, but a light guided tour through some major issues. Whether this lack of depth will be the book’s best feature or worst failing depends entirely on your requirements. Workers in the field may find it frustratingly brief on important topics. Novices will find a well structured and different approach to the literature, which is referenced in abundance.

There are six chapters, five of them relatively brief. Chapter two, dealing with results from current cognitive psychology, I found the most disappointing. Much more detail would have been welcome here, as this is an area poorly covered by other books. The heart of the book is chapter 5, which reviews the literature in expert systems application from Slatter’s cognitive emulation perspective. I think this is probably the book’s selling point for people wanting an introduction to the area. I would hope that, rather than satisfying the reader’s appetite, this section will spur you on to seek out important work in the literature.

My other major criticism of the book is that certain key areas have been ignored. Current work on naive physics and qualitative reasoning are entirely ignored, yet are an important part of the book’s target field. A section on the current cognitive paradigms from machine learning for knowledge acquisition would also have been nice. Finally, I would like to have seen a more formal treatment of why rule based systems on their own may be inadequate for some tasks, as this seems to be one of the underlying themes of the book.

Overall, I think the book would be welcome as an addition for a library. I am unsure if the expected price will make it attractive for many individuals. Maybe this should have been released in paperback. After all, it will be well out of date in a couple of years.

E. Coiera
University of New South Wales


For the price this text book is well worth including in the library of a capacity planning department, although with a much higher Australian price I probably would not buy it for myself. There is a certain amount of padding and repetition which I found a bit annoying. However, within this padding are several very salient points which are not found in most other works.

There are six discrete sections to the book. The first section is an overview of capacity planning, the issues involved and the contents of the book. The second section details the tasks that a capacity planner must perform in order to arrive at a capacity plan. There are some useful practical examples and a few case studies in this section.

One third of the book is devoted to the third section which discusses in some detail the methods used in the component approach to capacity planning and the system modelling approach. There are many mathematical formulae presented and a detailed example. A discussion of the merits between the two methods is included. While the authors believe in the system modelling approach they recognise that it is a more complex and expensive method and not widely used.

The fourth section contains details of a survey of the FORTUNE 1000 companies (500 industries and 500 service). Three hundred and eighty eight companies responded and the results of these are given along with the survey questions and a discussion on the methodology used. The fifth section draws conclusions about the survey. It highlights differences between types of industry, which parameters they use and the basis of their capacity plans. The last section is a summary and makes recommendations on how capacity planning should be developed in the future.

The book contains many references throughout. These can be used by a reader to gain more information about specific aspects and demonstrate the professionalism of the authors. Although the book is written by academics they have attempted to write a practically oriented book. Their survey was conducted to determine what is actually practised throughout industry by people who have to get the job done.

In summary this book would be very useful for the capacity planner who is setting up or changing the capacity planning function within an organisation. From the survey he can ascertain the capacity planning methodology needed for his type of business based on what others do. The rest of the book spells out different approaches to capacity planning and some of the traps. Higher executives may find the book useful if they are looking for ideas on how to plan and control their Information Services Division. The book is in an easy to read format, is double spaced and contains high and low level information on what is and what should be performed by a Capacity Planning Department.

D. Smith
CSA, Systems Consultant (Capacity Planning)
(Continued from inside front cover)


INTERFLORA BLOOMS WITH TANDEM COMPUTERS

Interflora, the largest worldwide flower relay organisation, has signed an agreement to purchase a Tandem NonStop computer and terminals made by Tandem Computers in Australia, to streamline its local and international floral delivery service.

The agreement is for the supply and installation of 1700 intelligent terminals, a Tandem fault-tolerant computer and customised application software. The new system will provide a network to interconnect the 1660 Interflora members throughout Australia with gateways to the 55,000 members of Interflora in 140 countries throughout the world.

INTERNATIONAL POSTGRADUATE SCHOLARSHIPS

Postgraduate scholarship applications for places in a Masters Degree in Electronic Engineering Programme are once again being invited, by Philips, from young graduate engineers and scientists in Australia.

The study and training programme will be at Philips International Institute (PII) in Eindhoven, Holland, where 14 Australians have previously studied under this scholarship scheme.

The Netherlands University Foundation for International Co-operation accepts responsibility for allocating Masters Degrees to PII students who successfully complete a three semester degree program (17 months).

Financial support, including air fares and living allowances, is paid and other assistance is also provided. English is the language of tuition.

Competition for the 24 available places is worldwide and very keen.

Application forms and full information from Philips Australia. Phone (02) 925-3270.

GRIFFITH UNIVERSITY

Griffith University will soon have one of the most advanced administration computer systems on any Australian campus.

Griffith is installing two new computers and has chosen the UNIX operating system.

The university will outlay about $900,000 over several years for hardware, software and extra personnel for system redevelopment, but director of the University's Information Technology Centre, Mr Mike Steel, has estimated that savings will be about $69,000 a year.

The hardware, two Sun 4/280S computers, each with 32 megabytes of memory, will arrive late this month and are expected to take a week to install. Griffith will use a UNIX operating system, making it only the second Australian university, after Deakin, to install a UNIX system for administrative applications.

IEEE

ASSPA 89

1989 Australian Symposium on Signal Processing and Applications

17-19 April 1989

Adelaide, South Australia

ADVANCE NOTICE AND CALL FOR PAPERS

The South Australia Section of IEEE is proud to be able to host ASSPA 89. ASSPA 89 aims to provide a forum for Australian workers active in the pervasive field of Signal Processing and its applications to exchange views in both research and applications areas.

SCOPE

The following list is a guide to areas in which papers are invited:

* Signal Processing for Detection, Estimation and Tracking — Theory and Application
* Array Signal Processing — Theory and Application
* Image Processing
* Spectrum Estimation
* Remote Sensing Applications
* Pattern and Target Classification
* Communications Signal Processing
* Signal Processor Applications and VLSI
* Other Applications of Signal Processing

PAPER SUBMISSION

Prospective authors should request a Paper Description Cover Sheet from the Technical Subcommittee Convenor and submit it with a one-page summary of their paper for review. Submission of a summary implies a commitment to present the paper if accepted and to adhere to timescales and procedures given here. The schedule for paper submission to the Technical Subcommittee Convenor is as follows:

* Four copies of summary and cover sheet — 3 October 1988
* Notification of acceptance despatched — 1 November 1988
* Papers for final refereeing and print — 30 December 1988

In addition to invited papers a limited number of review papers will be included. Proposals for review papers should be raised with the Technical Subcommittee Convenor as early as possible.

EXHIBITS

An equipment exhibition will be part of the Symposium. Prospective exhibitors should contact the Symposium organisers.

FOR FURTHER INFORMATION

To ensure that you are on the mailing list for further information, forward your name and address to the Symposium Organisers.

Technical Subcommittee

Convenor

Mr G.C. Mountford

Maritime Systems Division

PO Box 1700 Salisbury SA 5108

Phone: (08) 259 6491

Fax: (08) 259 5139

Symposium Organisers

Techsearch Incorporated

GPO Box 2471

Adelaide SA 5001

Phone: (08) 267 5466

Fax: (08) 267 4031
The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

OFFICE BEARERS: President: J. Goddard; Vice-Presidents: P. Murton, A. Underwood; Immediate Past President: R. Christie; National Treasurer: R.G. Heinrich; Chief Executive Officer: A.R. Kelly, PO Box 319, Darlinghurst, NSW, 2010. Telephone (02) 211 5855.


SUBSCRIPTIONS: The annual subscription is $25.00. All subscriptions to the Journal are payable in advance and should be sent (in Australian currency) to the Australian Computer Society Inc., PO Box 319, Darlinghurst, 2010. A subscription form may be found below.

PRICE TO NON-MEMBERS: There are four issues per annum. The price of individual copies of back issues still available is $7.50. Some are already out of print. Issues for the current year are available at $9.00 per copy. All of these may be obtained from the National Secretariat, PO Box 319, Darlinghurst, NSW 2010. No trade discounts are given, and agents should recover their own handling charges.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining partway through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $7.50 per copy. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly, of any change of address.

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: GPO Box 446, Canberra, ACT, 2601. NSW: 6th Floor, 66 King Street, Sydney, NSW, 2000. Qld: Box 135, GPO, Aspley, Qld, 4034. SA: Box 2423 GPO, Adelaide, SA, 5001. WA: Box F320 GPO, Perth, WA, 6001. Vic: PO Box 98, East Melbourne, Vic, 3002. Tas: PO Box 216, Sandy Bay, Tas, 7005. NT: GPO Box 213, Darwin, NT, 0794.

CONTRIBUTIONS: All material for publication should be sent to Mr Rob Cook, Editor, Australian Computer Journal, Centre for Information Technology Research, University of Queensland, St. Lucia, Qld 4067. Prospective authors may wish to consult manuscript preparation guidelines published in the February 1986 issue. The paragraphs below briefly summarise the essential details.

Types of Material: Four regular categories of material are published: Papers, Short Communications, Letters to the Editor and Book Reviews. Generally speaking, a paper will discuss significant new results of computing research and development, or provide a comprehensive summary of existing computing knowledge with the aim of broadening the outlook of Journal readers, or describe important computing experience or insight. Short Communications are concise discussions of computing research or application. A letter to the Editor will briefly comment on material previously appearing in the Journal or discuss a computing topic of current interest. The term ‘computing’ is interpreted broadly to include descriptions of computer hardware and software, and major computer applications such as information systems, expert systems, computer networks and office automation.

Refereeing: Papers and Short Communications are accepted if recommended by anonymous referees, Letters are published at the discretion of the Editor, and Book Reviews are written at the Editor’s invitation upon receipt of review copies of published books. All accepted contributions may be subject to minor modifications to ensure uniformity of style. Referees may suggest major revisions to be performed by the author.

Proofs and Reprints: Page proofs of Papers and Short Communications are sent to the authors for correction prior to publication. Fifty copies of reprints will be supplied to authors without charge. Reprints of individual papers may be purchased from Associated Business Publications, PO Box 440, Broadway, NSW, 2007. Microfilm reprints are available from University Microfilms International, Ann Arbor/London.

Format: Papers, Short Communications and Book Reviews should be typed in double spacing on A4 size paper, with 2.5 cm margins on all four sides. The original, plus two clear bond-paper copies, should be submitted. References should be cited in standard journal form, and generally diagrams should be ink-drawn on tracing paper or board with stencil or Letraset lettering. Papers and Short Communications should have a brief Abstract, Key word list and CR categories on the leading page, with authors’ affiliations as a footnote. The authors of an accepted paper will be asked to supply a brief biographical note for publication with the paper.

This Journal is Abstracted or Reviewed by the following services:

Publisher
ACM
AMS
CSA
ENGINEERING
INDEX INC.
INSPEC
INSPEC
ISI
ISI
SPRINGER-
VERLAG

Copyright © 1988. Australian Computer Society Inc.


All advertising enquiries should be referred to the above address.

Printed by MAPS Litho Pty Ltd, Unit 3, 8 Aquatic Drive, Frenchs Forest, NSW, 2086.