CONTENTS

RESEARCH PAPERS
1- 9  Expert Systems in Crisis Management: Ambulance Dispatch
      A.N. HAWKE and C.D. MARTIN
10-18 Including an Explicit Memory Model in a Machine Description
      C.A. LAKOS
19-26 An And-Parallel Dataflow Model for Logic Programs Based on
      Mode Prediction
      T-C. CHUNG and J-W CHO
27-32 On the Analysis of Iterative and Recursive Programs
      F. MILI, B. ALAYA and A. MILI

APPLICATIONS STUDY
33-40 CSIRONET's Terabit File Store
      B.R. LEDERER, R.J. DAKIN, K.R. PARKER and A.L.V. COOK

SPECIAL FEATURES
41-48 Book Reviews
48  Letter to the Editor
LIBRARY ASSOCIATION OF AUSTRALIA TO CHANGE CONDITIONS FOR PROFESSIONAL MEMBERSHIP

The Library Association of Australia will introduce revised conditions for professional membership of the Association from 1 January 1988.

From this date, the Association will consider applications for professional membership from persons who have completed qualifications which are not currently recognised by the Association but where the course of study included a substantial component of information studies e.g. a major within a Bachelors' degree or UG2 diploma, or the greater part of a graduate diploma or higher degree. In the case of overseas qualifications, the studies should be equivalent in duration to the normal time for preparation of professional level library and information workers in the country in question.

Persons wishing to apply for professional membership of the Library Association of Australia under the new provisions will need to provide the following documentation:

i. details of degree or diploma studies, including:
   - course content
   - requirements for entry to the course
   - normal duration of course
   - dates of commencement and completion of course

ii. details of any further study undertaken and awards received since completion of the primary degree.

iii. details of employment history, including duty statement of the applicant's current (or most recent) position in library and information service.

iv. details of relevant publications or professional reports

v. membership of and/or activity in relevant professional groups and organisations

vi. details of any continuing education courses or programs attended/completed which relate to the field of library and information science.

vii. the name of a person familiar with the applicant's work who may be referred to for advice as to the nature and level of the applicant's work experience, in the information service field.

viii. any other relevant information or documentation which the applicant considers demonstrates that he/she meets the criteria for Associateship.

In addition, applicants who have completed their library and information studies overseas should include:

i. details of:
   - recognition of course by the relevant national library or information science association.
   - an outline of the pattern of education for professional level information work in the overseas country.

ii. attachments such as:
   - a certified copy of the original award
   - a translation of the award if the language of the original is not English
   - a transcript of results
   - a copy (in translation if appropriate) of the course syllabus, including details of coverage of individual subjects, and reading lists if possible.

A panel convened by the Course Recognition Committee of the LAA's Board of Education will interview applicants and make recommendations to the Board of Education.

Application forms and further information about the new procedures are available from the Library Association of Australia Head Office, 376 Jones Street, Ultimo NSW 2007. (02) 692 9233 or (008) 22 1481.

BINDERS FOR THE "AUSTRALIAN COMPUTER JOURNAL"

NOW YOU CAN STORE YOUR COPIES OF THE "AUSTRALIAN COMPUTER JOURNAL" CONVENIENTLY FOR EASY ACCESS AND REFERENCE

The "Australian Computer Journal" binder will hold twelve copies (3 years). Each magazine is held separately by a wire for ease of opening and removal.

Available in colour blue and clearly labelled on cover and spine, this binder will complement your library or office.

Price: $10.00 per unit, including postage and packing (add $5.00 for overseas orders).

Orders to:

ACS National Secretariat
PO Box 319, Darlinghurst, NSW, 2010

Please supply ................ ACJ Binders.

Payment of $A.................... is enclosed.

Name ........................................................................................................................................................................

Address .......................................................................................................................................................................

...............................................................................................................................................................................

Please allow three weeks for delivery within Australia.
Expert Systems in Crisis Management: Ambulance Dispatch

A.N. Hawke and C.D. Marlin†

Crisis management concerns situations where a human operator must make rapid decisions in an environment in which incorrect decisions have severe consequences. This paper describes an experiment in the application of expert system techniques to a particular form of crisis management: the allocation of ambulances to requests in a metropolitan area. The ambulance coordinator's task is described and the implementation of an expert system to advise coordinators is outlined. OPS5 is used to code the coordinator's expertise and Pascal is used to perform calculations, to interact with the operating system and to handle databases of information about suburbs and ambulances. Testing the expert system is also described.

Keywords and Phrases: Expert systems, crisis management, ambulance dispatch, OPS5, Pascal.


1. INTRODUCTION

The term crisis management is used in this paper to describe situations where a human operator is expected to make quick decisions based on many factors and where the consequences of an incorrect response are severe (including, for example, death). Such critical situations occur in applications such as air-traffic control, management of nuclear reactors and chemical plants, scheduling of emergency vehicles such as ambulances and fire appliances, and various military situations.

All these examples represent high-stress occupations and the application of expert systems as advisors in these areas has many potential benefits, including:

1. Helping to reduce the stress associated with such occupations by reducing the responsibility of the human operator to ensure that all relevant factors have been taken into account.
2. Facilitating training by comparing a trainee's response with the suggestion of the expert system, either in a real crisis or using stored data about situations that occurred in the past.
3. Reducing the problems caused by the retirement or reallocation of expert operators, by continuing to have their expertise available through the expert system.
4. Combining the expertise of several operators in the expert system, which might well mean that it will consider a combination of factors that any individual operator would not consider.

Despite these benefits, applications of expert systems to crisis management still appear to be rare. One such expert system is that described by Anderson, Cramer, Lineberry, Lystad and Stern (1984) to automate low-level tasks in the cockpit of an aircraft during an emergency; such an expert system is clearly operating in a crisis management situation. Similarly, the AIRPLAN expert system described by Masui, McDermott and Sobel (1983) operates under severe time constraints; this system assists with the launching and recovery of aircraft from an aircraft carrier. Another expert system involved in a crisis management application is the Ventilator Manager program (Fagan, Kunz, Feigenbaum and Osborn, 1983), which assists with the management of breathing apparatus in the intensive care unit of a hospital. Also relevant is the expert system developed for advising on the management of spills of oil and hazardous chemicals at the Oak Ridge National Laboratory (Johnson and Jordan, 1983).

Other expert systems have been designed to cope with the demands of real-time analysis and control. One example is the expert system described by Hood and Mason (1986), that interprets incoming radar signals, trying to identify the sources of the signals and infer their movement. Wright, Green, Fieg! and Cross (1986) describe Hexscon, a general framework for building expert systems for real-time control in military and industrial applications.

There are other expert systems that have been developed in application areas relevant to crisis management, but which do not have to operate under the severe time constraints that we regard as characteristic of crisis management. One example is the expert system developed by Ray, Lee, Morgan and Airth-Kindree (1986), that monitors sleeping patients and diagnoses sleeping disorders; the authors state that this system should ideally operate in real-time, although it does not do so at present. Cross (1985) describes an expert system to advise on air-traffic control, one of the crisis management applications mentioned earlier; this system looks for possible collisions in a
**Figure 1. The physical layout of the communications room at the St John Centre.**

The display of many aircraft, but troublesome situations are detected well in advance and the time constraints are hence not severe. Finally, two military expert systems for advising on target selection are described by Callero, Waterman and Kipps (1984) and by Slagle and Hamburg (1985). The first of these is called TATR and advises on what priorities to assign to possible air-strike targets (the latter being airfields and the aircraft on them); the second, BATTLE, advises on the allocation of a set of weapons to a set of targets, for maximum effect. Neither of these expert systems operates in real-time, both performing forward planning for later attacks, and so are not regarded as addressing a crisis management problem.

This paper describes the development of an expert system to advise on ambulance coordination in the Adelaide metropolitan area. The particular problem was chosen because, through the Road Accident Research Unit at the University of Adelaide, we had an introduction to the relevant personnel at the St John Centre in Adelaide and because it seemed that the problem was typical of others in the area of crisis management. Other advantages included the availability of the operators' log sheets, which could be used in testing the expert system, and the willingness of the St John staff to discuss the means by which they came to their decisions.

The remainder of this paper is structured as follows: Section 2 describes the ambulance co-ordination problem in some detail; Section 3 describes the implementation of the expert system to advise on ambulance coordination; Section 4 describes testing the system; and, finally, Section 5 presents some conclusions.

2. THE PROBLEM

The physical layout of the communications room, the part of the St John Centre where ambulance coordination takes place, is shown in Figure 1. A single switchboard operator controls calls coming into the building and routes them to their various destinations within the building. A request for an ambulance is referred to one of the two telephonists who look after such requests. The telephonist receiving the call then takes down the details of the call, such as the whereabouts of the incident, the condition of the patient, perhaps the patient's destination, and so on. The telephonists give a priority to the request using guidelines laid down by a body of medical specialists, entering the details of the call on a colour-coded card and placing the card on a belt-conveyor system. Depending on the slot in which it is placed, the card is then gathered in by one of the supervisors with responsibility for medical retrievals, clinic vehicles or city requests. The radio/telephonist (radtel) at the top of Figure 1 is only used in large emergencies, such as extensive bushfires, when it is necessary to make high-level decisions about the allocation of insufficient resources.

The *senior supervisor* is responsible for coordinating medical retrievals for all of South Australia; the concerns in this coordination task include the distance from the incident, the availability of medical staff and which of road vehicles, helicopters or aeroplanes to use. For example, a patient needing urgent transport from Port Pirie to a hospital in Adelaide may need a doctor on hand throughout the trip and the supervisor would need to consider whether a road vehicle would be quicker or smoother, or whether a trip by plane should be organised. The senior supervisor also supervises all the other staff in the communications room and undertakes some administrative duties.

The *clinic supervisors* coordinate a set of vehicles within the Adelaide metropolitan area for clinic patients, minimising the number of runs that these vehicles have to make; these supervisors are assisted by a radtel. Clinic patients are generally people who need to attend outpatient clinics at hospitals and are either too poor or too sick to use private transport, or too sick to use public transport; clinic cases involve moving some 300 to 400 people per day.

The *city coordinators* deal with requests for ambulances throughout the city. During the daytime from Monday to Friday, there are two of these coordinators, one for the Northern sector of the metropolitan area and one for the Southern sector; each coordinator has up to 17 vehicles under his or her control. At night and on weekends, there is only one coordinator for all of Adelaide; 75% of the 300 to 400 city requests for ambulances during a day occur between 7 am and 7 pm. Apart from the belt-conveyor system, two other kinds of communication occur between the personnel in Figure 1.

- When an urgent call is received, the call sets off an alarm on all desks and the coordinators are able to listen in on the call and thus get advance warning of the whereabouts and gravity of the incident, giving them a little extra time to choose a vehicle, and

- When things get desperate, a city coordinator may take a vehicle from the other coordinator's sector or from the pool of clinic vehicles.

Our preliminary investigations showed that the task of the city coordinators offered the most promise as an application of expert systems techniques to crisis management and so we soon restricted our attention to the subproblem. In fact, in the remainder of this description, the term coor-

**Dinat** will be used exclusively to refer to a city coordinator.

As mentioned above, requests are assigned a priority by the radio taking the details of the request. These accident priorities have the following meanings:

**Priority 1**
An ambulance should reach the incident within six minutes, even if it means borrowing a vehicle from the other sector; there are repercussions on the coordinator if the vehicle does not reach the incident in ten minutes. In responding to a priority 1 request, the driver is allowed to break traffic rules relating to speed-limits, red lights, median strips, etc.

**Priority 2**
Although these requests should be answered immediately if possible, there is a maximum of 20 minutes to reach the incident. In this case, there is no flouting of traffic rules.

**Priority 3**
These are more mundane cases, such as a request logged at 10 am for a vehicle to transport a patient from one hospital to another at 2 pm. These are generally answered only at the coordinator's discretion, when he or she feels sure that there are enough vehicles in reserve for priority 1 responses. Within this priority, requests are loosely ordered, with patients who are already at a hospital at the bottom of the list and those needing stretchers at the top.

Various factors are taken into account by a coordinator in deciding which ambulance to allocate. Although some of these factors relate to the route the ambulance may take from its present location to the incident and then to a hospital, it must be stressed that it is the vehicle's crew who actually choose the route taken. The major factors uncovered by interviewing coordinators were:

**The severity of the call**
This is represented by the priority categories described above.

**The type of vehicle needed**
Some vehicles carry advanced life-support equipment and this equipment may influence which vehicle is sent to a particular incident. For a priority 1 request, speed is paramount and so this factor is of little importance.

**Traffic conditions**
These conditions play a key part in the decision-making process. Aspects that are considered include the time of day (e.g., whether or not it is rush-hour, and the relative flow of traffic), special events (such as street parades) and road-works.

**The position in relation to the city**
If an accident occurs on one side of the city and there is a vehicle in such a position that it would have to pass through or close to the city, then it may not reach the accident any sooner than a vehicle further away which does not need to go through the city area.

**The crews available**
For instance, if a crew has been working flat-out all day, then they may deserve an easy job next time around, or if a crew is approaching a projected lunch-break and a long but relatively low priority request is in the offing, then another crew will be chosen — of course, if the request is priority 1, then this factor would not be rated highly.

**Major freeways in the area**
If there is a request for an ambulance in a suburb close to an arterial road and there is a vehicle which is also close to that road, then it would be far quicker for that vehicle to break the speed limit down the major road than for another to battle its way through adjacent suburbs.

After a vehicle has been allocated, the coordinator must generally ensure that the free vehicles in his or her sector are spread evenly over the sector to take care of sudden life-or-death requests, where response time is of particular importance. For instance, if two or three ambulances are busy in one part of the metropolitan area taking care of requests which must be handled immediately and their departure has left a large gap, then the potential for a poor response time in that area has been created and so the coordinator must detail other ambulances into the gap that now exists. However, if the depleted area is populated mainly by young (and hence healthy) people and if the area is not heavily industrialised, the coordinator may rely to a small extent on luck and wait until the vehicles finish their requests and return to their assigned areas without attempting to fill the gap. Such decisions are complicated because areas change in character over the years, so an area which one year was particularly notorious for urgent requests may be trouble-free the next.

The above description of the task carried out by the coordinator shows that the decision-making process involved is complex and many heuristics are involved. This led us to believe that expert system techniques, rather than the use of an algorithmic approach, would be suited to the implementation of a program to advise the coordinators.

The coordinator's job is learnt by experience and on-the-job training. They are not thought to have become proficient for at least three years and the usual routine is to gradually move them into areas of greater responsibility and pressure as their skills increase. On average, a coordinator lasts seven years before the shift-work, concentration and pressure take its toll and he or she resigns for some less gruelling task. This means only a four-year work span as a skilled coordinator, underscoring the potential utility of an expert system suitable for training and consultation.

### 3. IMPLEMENTATION

**3.1 Structure**
The production system approach was chosen for the construction of the expert system. The principal reasons for this were:
— It was the approach with which we were the most familiar.
— We had recently received the OPS5 implementation for VAX/VMS (Digital Equipment Corporation, 1983) and were looking for an opportunity to construct an expert system using it, and
— Preliminary discussions with the domain experts at the St John Centre suggested that we could expect forward chaining as a major control mechanism within the expert system.

As the expert system was developed, it became clear that OPS5 (Brownston, Farrell, Kant and Martin, 1985; Forgy, 1981) was not at all suitable for the efficient implementation of some aspects of the system. One such aspect concerns the maintenance of two key databases: the suburbs database, describing the location of suburbs within the metropolitan area, and the ambulances database, describing the current location and status of each ambulance.

To be able to search these databases efficiently (for example, the suburbs information is stored in a hash table), it was decided that another language would have to be used; the language chosen for this task was Pascal. Other aspects of the implementation that proved inconvenient in OPS5 concerned the calculation of distances between locations; the latter involves complicated trigonometry, which turned out to be inconvenient in OPS5.

The expert knowledge of the coordinators is principally contained within the Acc module. This knowledge was acquired from the coordinators through many hours of observation and questioning. The coordinators were extremely cooperative, motivated at least in part by the belief that we could not possibly put their hard-won expertise into a computer program. The only real difficulty with the knowledge acquisition aspect of the project was the fact that the coordinators frequently did not know how they did their job and, in particular, they were frequently not aware of all of the factors that they took into account in coming to a decision.

When a request arrives for an ambulance, each unassigned ambulance is retrieved from the ambulances database and its description is pushed into the working memory of the production system. The productions are then applied, gradually assigning to the vehicle a number of points depending on the suitability of the vehicle to respond to the request. At the end of this process, the three vehicles with the highest number of points are presented to the coordinator, with advice on which to choose. Sometimes, but only for a priority 3 request, the advice will be to not choose a vehicle at all.

```
(p check_life_support
 (wait ↑2 - 2)
 (dist ↑2 <dist> )
 (vehicle_descr ↑2 { <veh> <> EMPTY } )
    ↑points <pts> ↑life_support E )
 (closest ↑3 ( <x> = <dist> ) )
 (position ↑life_supp_needed Y)
  →
 (modify 1 ↑2 3)
 (modify 4 ↑2 <veh> ↑3 <dist> )
 (modify 3 ↑points (compute <pts>+3))
```

Figure 3. An OPS5 production from the Acc module.

The working memory of the production system within the Acc module contains several elements, including:
— `vehicle_descr`, which describes the ambulance currently being evaluated,
— `closest`, which describes the ambulance which, of those evaluated so far, has been found to be closest to the incident,
— `wait`, which is used to control the order in which some of the productions are permitted to fire,
— dist, which describes the distance between the ambulance being evaluated and the incident, and
— position, which describes the incident requiring an ambulance (location, request priority, etc.).

These working memory elements frequently have internal structure, in that they consist of attribute-value pairs. For example, the 'vehicle_desc' element contains attributes for the ambulance number, its location, when its crew started work, whether it has advanced life-support equipment, the number of points allocated to it in the evaluation process thus far, and so on.

The productions in the Acc module are predominantly concerned with the examination of the factors discussed in Section 2; one of these productions is shown in Figure 3. An OPS5 production consists of '(p, the name of the production, the left-hand side of the production, the symbol '->', the right-hand side of the production and ')'. The left-hand side of a production is a collection of patterns called condition elements, each of which is a pattern to match a working memory element. For example, the first condition element in Figure 3 is

\[(\text{wait} \uparrow 2 \ 2)\]

and evaluates to 'true' if the working memory contains an element 'wait' and the value of its second attribute is '2'. The second condition element succeeds if the working memory contains an element 'dist'; if it succeeds, the value of the second attribute of this element is to be denoted by the variable '<dist>'. The third condition element is an attempt to match a working memory element called 'vehicle_desc'; the second attribute of this element, which is to be denoted by '<veh>', is not to have the value 'EMPTY', its 'points' attribute is to be denoted by '<pts>' and the 'life_support' attribute is to have the value 'E'.

The next condition element specifies that there is to be a working memory element 'closest' and that its third attribute (which is to be denoted by '<x>') should have the same value as that denoted by the variable '<dist>' (i.e., the value found earlier in the second attribute of the working memory element 'dist'). The final condition element succeeds if there is a working memory element called 'position' whose 'life_supp_needed' attribute has the value 'Y'.

The right-hand side of an OPS5 production consists of an unconditional sequence of commands called actions. There are three such right-hand side actions in Figure 3. The first is

\[(\text{modify} \ 1 \ \uparrow 2 \ 2 \ 3)\]

which takes the first pattern in the left-hand side of the production, namely the pattern

\[(\text{wait} \ \uparrow 2 \ 2)\]

and creates a new working memory element whose contents are the same as the element associated with the pattern (i.e., 'wait'), except that its second attribute now contains the value 3; the previous element of this name is then removed from working memory. The second action on the right-hand side of the production in Figure 3 simply updates the second (ambulance number) and third (distance from ambulance to incident) attributes of the working memory element 'closest'. The third action takes the working memory element matched by the third condition element (i.e., the ambulance description) and increments its 'points' attribute by 3.

Thus, the effect of this production is to test whether the vehicle under consideration is as close to the incident as the closest vehicle so far and has advanced life-support equipment, and whether advanced life-support equipment is required for this request. If these conditions are met, this vehicle is identified as being the closest (by entering its number in the second attribute of the 'closest' element of working memory) and its distance from the incident is entered as the shortest such distance (by entering this distance in the third attribute of the 'closest' element). The points score associated with the vehicle is then incremented appropriately. (As mentioned earlier, the 'wait' element of working memory is concerned solely with controlling which productions will be permitted to proceed at any given time.)

The strategy of pushing an ambulance description into working memory and then gradually computing a point score for it works well for priority 1 and priority 2 requests; this is primarily because a vehicle will almost always be assigned to them immediately. In fact, for priority 1 cases, the closest vehicle is chosen almost without exception, the questions of overtime and life-support capabilities turning out to be largely immaterial in practice. Arriving at the site in question in the minimum amount of time is of paramount importance. For priority 2 cases, the same does not always apply. Sometimes, if the coordinators are running out of unassigned ambulances, they may decide to wait a short time before responding to such a request. The question of overtime has more importance and an inferior vehicle may be chosen for the task if it will relieve the pressure on another area or the end result is good positioning for another task. However, the major factor is still getting an ambulance there as quickly as possible.

Priority 3 requests turned out to be difficult to deal with in the expert system, because they are generally only responded to at the coordinator's discretion. In other words, they are only answered when the coordinator feels that answering the request will still leave enough vehicles in the reserve pool to respond to more serious requests within the required time limits. A coordinator may, in fact, ignore a priority 3 request for up to an hour before detailing an ambulance. This means the expert system had to include the ability to store priority 3 requests (unlike priority 1 and 2 requests), and to periodically search these requests to determine if a request could, at this later time, be serviced. As mentioned earlier, an ordering within priority 3 also exists, with those patients already in hospital generally waiting the longest, while those needing stretchers are attended to before those that can walk. On the other hand, the longer the patient waits, the higher the associated priority becomes.

To store priority 3 requests, a queue is used; each time the queue is scanned, requests already on the queue are
increased in priority, the amount of increase depending on whether the patient is already at hospital, walking or a stretcher-case. Requests are initially entered in the queue at different positions depending on their nature. One aspect of priority 3 requests which was not implemented is that a coordinator will sometimes send an ambulance in response to a priority 3 request whose time has not yet arrived, simply because an ambulance is free in the area at that time. This behaviour was not mentioned during the interviews with coordinators and was only noticed during testing, when it was realised that vehicles recorded in the log sheets were answering requests up to half an hour before schedule.

3.3 The Suburbs and Ambulance Databases

The questions of the representation of locations and the calculation of distances occupied a good deal of our attention in the early stages of the project. Observing the coordinators revealed that they primarily use the position of one suburb relative to another, normally taking the straight-line distance. This led us to the important conclusion that searching for the best path from the vehicle to the incident was not part of the problem. Distances between objects were thus implemented as straight-line distances in almost all cases. The only exceptions involve suburbs on a peninsula (where the straight-line distance between an incident on the end of the peninsula and an ambulance across the adjoining gulf is clearly misleading) and a particular suburb split by a railway line with only one crossing.

Adelaide has relatively small suburbs and so the suburb name was regarded by the coordinators as a sufficiently good description of location when computing distances. Consequently, the suburbs database was established as the source of information about the location of incidents. As mentioned previously, it is organised as a hash table and contains the name of the suburb and its latitude and longitude. On the other hand, the ambulances database contains four separate lists of vehicles:

1. Vehicles on duty. This list holds the vehicles on duty at that particular time, including the vehicles that are currently on lunch-break.
2. Vehicles unassigned. This is the list of free vehicles; it is searched for each request.
3. Vehicles assigned. This is the list of vehicles currently assigned to some task.
4. Vehicles to be evaluated. All the vehicles in the 'vehicles unassigned' list are copied to this list when a request occurs and each time a call is received by the production system for a new vehicle to be processed, a vehicle description is removed from this list and passed to the production system; when the list is empty, all the free vehicles have been examined and processed by the production system.

An element in any of the lists of vehicles contains the name of the vehicle, the position of the vehicle, the time it began work, the projected end of the working day, the projected time for lunch break, whether the crew have had their lunch-break yet, and whether the vehicle has advanced life-support capabilities.

3.4 The Interface between OPS5 and Pascal

As mentioned in Section 3.1, the Pascal routines contained in the modules Loader, Accid and Scheduler are called by the OPS5 modules Acc and Controller. Establishing the correct interface between the two languages turned out to be more difficult than expected and provides an interesting example of the practical difficulty that may arise when attempting to make use of an 'expert system language' such as OPS5. Some of the difficulties encountered were due to the standard of the documentation (Digital Equipment Corporation, 1983), whereas others were because of the tedious nature of the protocol required.

The way in which the interaction between OPS5 and Pascal must be accomplished is illustrated by the example in Figures 4 and 5. Figure 4 contains a fragment of OPS5 code, consisting of two productions and associated declarations. Among these declarations is the indication that there are three external routines known as 'distance', 'knocking_off_time' and 'area_empty'. The calls on these routines are deceptively simple and occur at the end of the production known as 'priority3-emergency'. Note that parameters to be transmitted to a Pascal routine, such as '<p>' and '<acc>' in the call on 'distance', are simply included in the call.

The Pascal side of the interface, on the other hand, reveals the tedious code which is required merely to effect the transfer of information into and out of the Pascal routines. This tedium is illustrated by the Pascal procedure 'distance' (one of the routines called by the OPS5 code in Figure 4), which is outlined in Figure 5. Incoming parameters have been packaged in a result element by OPS5 and these have to be extracted using calls on 'ops$parameter';
Figure 5. An outline of the Pascal procedure “distance”.

```pascal
procedure distance;
begin
  (Obtain the position of the ambulance.)
  vall:= ops$parameter(1);
  vall:= ops$parameter(vail, vehpos, 22);
  (Obtain the position of the incident.)
  vall:= ops$parameter(2);
  vall:= ops$parameter(vail, accidpos, 22);
  (Carry out the distance computations.)
  ...
  (Return results to the OPS5 module.)
  ops$reset;
  hill_job:= ops$intern(‘job_pos ’, 7);
  ops$value(hill_job);
  vall:= ops$intern(constr(acceptr1.typ), 4);
  ops$value(vall);
  vall:= ops$intern(constr(vehptr1.typ), 4);
  ops$value(vall);
  ops$assert
end (distance);
```

Another difficulty relates to restarting the system, which was frequently necessary during testing. For example, when new productions were added, new score increments for use in these productions had to be found. This was a matter of using trial and error: after consulting with the coordinators and examining how the other factors had been weighted, an educated guess was taken and testing was begun to determine the effects. Unfortunately, each time a weight was changed in a production rule, the problem would have to be laboriously set up again by keying in the location and status of all the vehicles. Such restarting of the system could take up to 10 minutes and had to be done many times before increments were obtained which gave accurate results. A final problem found with testing was that the log sheets had to be transformed from the request-by-request form in which they were supplied into an appropriate list of chronological events.

Because of these difficulties, another mode of running the system was devised: testing mode. This mode is a simulation of a day’s activities, conducted according to information from the log sheets for that day; data files corresponding to the log sheets for various days can be selected by the user. In this mode, the system prompts the user after each user action, specifying the next event that will occur, when it will occur and the response made by the coordinator. This mode also enables the system (and the associated real-time clock) to be halted and permits the user to instruct the system to jump ahead to the next event.

The expert system was extensively tested and performs well in relation to the behaviour of experienced coordinators; this is especially true of responses to priority 1 and 2 requests (the most critical). As discussed earlier, the han-
dling of priority 3 requests involves considerable discretion on the part of the coordinators and hence it is not surprising that the expert system did not always recommend the same actions as the coordinators for these requests. Note that our measure of the performance of the expert system relates only to how well it performed in relation to the coordinators; no attempt was made to address the question of how optimal their performance was and whether the expert system could do better.

5. SUMMARY AND CONCLUSIONS
An experiment in the application of expert system techniques to the area of crisis management has been described. The problem chosen was the task of ambulance coordination in a metropolitan area and the implementation was carried out in OPS5 and Pascal. This work was carried out as the project component of the first author's Honours degree in Computer Science at the University of Adelaide. As such, it was never intended that the system would be used 'live' by the St John coordinators; the primary motivation for the project was to gain some experience with expert system techniques and with the OPS5 language. Although the project is finished and the system is never likely to be developed further by St John, it has shown them what is possible with an approach that they had never considered.

Initially, it was expected that the coordinators used a chain of logical reasoning that would lend itself to implementation with forward chaining in the expert system. However, later work revealed no such reasoning and the resulting expert system contains no chaining. The coordinator comes to a decision based on consideration of various factors and the expertise of a coordinator seems to lie in knowing how much weight to give to each factor for different situations and, also, in being aware of special cases. This was implemented in the expert system by examining each unassigned vehicle in turn and computing a point score for the vehicle, based on consideration of the factors identified in conversations with coordinators. This strategy gave acceptable results for priority 1 and 2 requests, where it can be expected that a vehicle will be assigned immediately. Priority 3 requests required a different implementation strategy, involving the maintenance of a queue of such requests.

Although planned partly to gain some experience with OPS5, the project has made extensive use of an algorithmic language (Pascal). The principal reasons for this were the need to search two databases efficiently and the limited arithmetic facilities of OPS5. It was also necessary to use Pascal for access to the real-time facilities of the operating system.

The language OPS5 was found to be convenient for some aspects of the implementation, particularly the encoding of the expertise of the coordinators. For example, we found it easy to add further such expertise incrementally as testing forced us to return to the coordinators and refine the knowledge contained in the expert system. We also found that the OPS5 implementation was sufficiently efficient to operate within the real-time constraints imposed by the need to respond quickly to priority 1 requests; we believe that the small number of rules (41 in the Acc module) was a major factor in meeting this real-time constraint.

The interface between OPS5 and Pascal proved difficult to master at first and tedious once mastered. Furthermore, the kind of side-effects that the Pascal modules must have on the working memory of the OPS5 modules is error-prone and makes the entire expert system difficult to understand.

A special mode of execution of the expert system was introduced to facilitate testing. Using this testing mode, it was shown that the system performs well when compared with the behaviour of experienced coordinators. Thus, we conclude that expert systems could be successfully employed as advisors in crisis management applications such as ambulance dispatch; using them in these applications would help to reduce the stress in such occupations and, because of the present long training period and short working life, are likely to prove cost-effective.

ACKNOWLEDGEMENTS
The authors would like to express their thanks to the staff of the St John Ambulance Centre in Adelaide and to Don De Giglio, the Assistant Superintendent of the Communications Division, in particular. This project would not have been possible without their willing co-operation. We are also grateful to Stephen Hood, Alex Sims and the referees for their helpful comments on earlier versions of this paper.

REFERENCES
JOHNSON, C.K. and JORDAN, S.R. (1983): 'Emergency Management of Inland Oil and Hazardous Chemical Spills: A Case Study in...

BIOGRAPHICAL NOTE
Andrew Hawke received his BSc degree in Mathematics and Computer Science from the University of Adelaide in 1984, and his Honours degree in Computer Science the following year. Currently he is employed at the University of Adelaide as a research assistant, where he is working on various aspects of an integrated incremental programming environment, including concurrency.

Chris Martin has been with the Department of Computer Science at the University of Adelaide, where he is now a Senior Lecturer, since 1984. He completed both his Honours degree in Computing Science in 1973, and his PhD in Computing Science in 1979, 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, USA. His research has primarily been concerned with programming language design and implementation, especially in relation to coroutines and parallel processes, and various aspects of integrated incremental programming environments.
Including an Explicit Memory Model in a Machine Description

C.A. Lakos†

This paper argues for the inclusion of an explicit memory model in machine descriptions used for retargetable code generation. The widespread approach of leaving the model implicit though parameterised at certain key points, is seen to be excessively limiting. By contrast, the inclusion of an explicit model allows for the elegant expression of several otherwise awkward machine features, including limited span branches, the presence of hardware stacks and memory segmentation even at the machine code level.

Keywords and Phrases: Retargetable code generation, machine description, memory model.
CR Categories: D.3.4.

1. INTRODUCTION

Recent years have seen significant growth in the research on retargetable code generation (Lunell, 1983). This work seeks to expand that of earlier compiler-compilers by automatically producing a code generator from a machine description. The gains to be made are a simplification of the porting or retargeting process and the general application of powerful code selection algorithms. The form of the machine description determines the class of languages and machines to which the particular scheme may be applied as well as the ease of use. We argue that many schemes suffer in this respect because they pay little attention to the modelling of the target machine's memory. They either have the memory model completely implicit (and thus predetermined), or they parameterise it at certain key points, the parameters possibly being part of the machine description and possibly specified separately. In this paper, we develop an explicit memory model and examine the implications of incorporating the model in a machine description language.

2. MACHINE DESCRIPTIONS FOR RETARGETABLE CODE GENERATION

A retargetable code generator (RCG) produces, from a machine description (MD), a translator from an intermediate representation (IR) of a program to some target language (TL), which is usually machine code, but not necessarily so. The translator is called a code generator (CG). As a first approximation, we can represent the above process as a pair of mappings:

\[
\text{RCG: MD} \rightarrow \text{CG} \\
\text{CG: IR} \rightarrow \text{TL}
\]

This representation, while it is adequate for most purposes, is not precise enough for others since it suggests a macro-like expansion of IR, i.e. for each IR element, corresponding TL element(s) are produced. In most implementations, however, CG will maintain some internal state which will be modified during the generation of code, and will, in turn, affect the generation of code. If we denote the information pertaining to the state as the domain INF, then the above mappings would more precisely be specified by:

\[
\text{RCG: MD} \rightarrow \text{CG} \\
\text{CG: IR} \rightarrow \text{INF} \rightarrow (\text{TL} \times \text{INF})
\]

though this refinement will not be of major concern in this paper. More importantly for our purposes, the representation as mappings is also deficient in that it suggests that languages MD, IR and TL have three distinct domains, while, in fact, there is a significant overlap between them. We now consider these overlap relationships in order to motivate the machine description constructs we introduce later. Since programs in IR and TL will be in some linear form such as prefix, postfix or infix notation, we restrict our attention to the operators and operands which make up these linear forms. For illustrative purposes, the observations we make are tied to the diagram in Figure 1 which depicts the languages MD, IR and TL as three intersecting sets of operators and operands. Initially, however, we restrict our attention to the operators, reflecting the balance of emphasis in the code generation literature.

![Figure 1.](image-url)
The relationships we observe are then as follows:

(a) There is a certain amount (hopefully large) of common ground between IR and TL. In any reasonable code generation scheme, with a flexible machine description language, this common ground will be expressed with a common notation in MD. This is what Fraser refers to as a comfortable description (Fraser, 1977). So, for example, a machine instruction implementing addition should be described using the same plus operator as occurs in IR. In other words, the overlap between IR and TL (as far as operators are concerned) will be completely contained in MD.

(b) We expect that there will be parts of TL which will not be covered by MD, since they will never be relevant to generating code from the given IR. For example, there is generally no need to describe the machine’s floating point instructions if IR does not cater for floating point arithmetic. This part of TL will henceforth be ignored.

(c) Similarly, we expect that there will be some parts of IR which will not be covered by MD. These will be aspects of IR which will always be transformed (machine independently) into some other IR construct or constructs before code generation. This arises since the retargetable code generator is often used with existing IRs, rather than IRs specifically designed for the purpose. For example, a stack-based IR might be converted into prefix form before code generation, or assignments occurring nested within expressions in IR may always be coded prior to the enclosing expression. We will ignore such possibilities in the future discussion.

(d) We will normally find that there are aspects of IR not found in a given TL. These need to be mapped into TL, and because these transformations are specific to a particular machine, we contend that they should be included in its MD. The mapping to TL may be specified via other IR constructs before code generation. For example, a machine with instructions to code boolean AND, OR, and NOT operators but with no instruction to code equivalence (EQV), would require a transformation of the form

\[ T_1 := t_2 \text{EQV} t_3 \rightarrow T_1 := (T_2 \text{AND} T_3) \text{OR} \text{NOT}(T_2 \text{OR} T_3) \]

where the semantic understanding of \( T_1, T_2, T_3 \) is left unspecified until Section 6.1.

(e) Some machine features or instructions (from TL) may not be common to IR but may still be required to code parts of IR (such as those in part (d)). For example, to implement an IR zero-fill right shift (RSHIFT) on a given machine may require the use of a sign-extended shift instruction (which we might denote by the operator RSSHIFT) and a mask instruction (denoted by the unary operator MASK) neither of which would be IR operators:

\[ T_1 := \text{t}_2 \text{RSSHIFT} t_3 \rightarrow T_1 := (t_2 \text{RSSHIFT} t_3) \text{AND} \text{NOT}(\text{MASK} t_3) \]

This transformation specifies how an operator in subset (d) is converted into operators in subset (e).

(f) It is often useful to have available aspects of MD which occur neither in IR nor in TL. These could then be used to simplify the specification of the transformations from subset (d) to subset (e) (as above). In the example quoted in part (e), it may be easier to use the given transformation even if the target machine had no instruction to produce a generalised mask. In this case, the MASK operator, at least in this general form, would occur neither in IR nor TL.

Given the observations above, we advocate that MD should not be one fixed interface for all intermediate representations and all target languages, as was intended by UNCOL (Steel, 1960, 1961) and as has been assumed in the Amsterdam Compiler Kit (Tanenbaum et al., 1983). Rather, the actual operators, operands and commands in a specific MD will be determined by IR and TL chosen. We distinguish between operators which return a single result, and commands which may alter more than one machine location but which do not return a result.

It is our purpose in this paper to concentrate on the memory model, but we note from the above that each machine description will need to contain declarations for the operators and commands appearing in IR, TL and MD (regions, a,d,e,f in Figure 1 above) together with their functionality. The only commands and operators for which the semantics are defined in MD are assignment, a padding command (see Section 5), and the operators required for the specification of addressing, namely arithmetic operators: +,-,*,// (remainder); boolean operators: &J,~; shift operators: <<<>>>; and addressing operators: @ (address of) and ! (indirection). Where notation in IR for these operators differs from the ones assumed above, the IR form is specified as an alias. With the declarations above, together with the declaration of the memory model considered below, it will be possible to generate automatically a symbolic IR recogniser provided we assume some standard form of IR such as prefix notation.

While the discussion above has emphasised the relationships between the various sets of operators, we now observe that similar relationships will also hold for the operands which occur in the languages IR, TL and MD. Hence it is more appropriate to picture the language relationships as in Figure 2.

---

**Figure 2.**
It could be said that many of the distinctive issues of code generation arise from this very focus on operands and memory restrictions. For example, the limited number of registers available on a machine needs to be carefully managed; some instructions may reference literals but only of limited magnitude or precision; certain addressing modes may give access only to memory locations within certain bounds; and jumps or branches may have limited spans. Such limitations make optimal code generation difficult (Aho et al., 1977; Szymanski, 1978; Robertson, 1979).

3. PREVIOUS APPROACHES
In view of the above considerations, it is surprising how little attention the modelling of memory receives in the various retargetable code generator systems. Many schemes either have the memory model completely implicit (and thus predetermined), or else parameterise it at certain key points. Fraser (1977) is an exception in that he declares the memory model in ISPS notation (Siewiorek et al., 1982) and then uses the encoded rules of the underlying expert system to deduce the necessary properties as well as the required operator transformations.

Ganapathi bases his approach on attribute grammars (Ganapathi, 1980; Ganapathi and Fischer 1985). The IR is a prefix form decorated with attributes and the MD is primarily given as an attribute grammar. By appropriately limiting the use of synthetic and inherited attributes, the grammar can be used in a one-pass code generation scan of IR. On the right hand side of each production may appear disambiguating predicates and action symbols. Disambiguating predicates are used to limit the applicability of productions by requiring them to be satisfied before the production can match. Typically, these predicates are used to check the size or value of an operand and perhaps even its location. Action symbols perform some semantic action such as emitting code, determining an instruction address, etc. As well as the attribute grammar there are a number of other components which make up the machine description. Thus the disambiguating predicates and action symbols, while they are referenced in the attributed grammar, are defined elsewhere. The binding of IR variables to address expressions is performed as a prepass of IR and is separately specified. The actual form of the address expressions is not machine-specific but some choices may be affected by the target machine architecture. Ganapathi considers such possibilities as the access to non-local variables via a static chain or via a set of display registers. The storage binding is further parameterised by a table of constants which specify the direction of stack growth, the offset and alignment of certain memory areas, the literal strings used for allocating various data items in the assembly language, and the registers used for particular purposes (the top of stack pointer, the current stack frame pointer, the display pointer, and the argument pointer). Yet another table specifies the data types available on the machine, in terms of their alignment restrictions and range of values. Finally, one more table specifies the capabilities of each addressing mode and interfaces with the addressing mode productions (in the attribute grammar). The addressing modes are given in terms of a bit-encoded field which indicates the number of levels of indirection (0,1,2 or 3 levels), whether a base register, displacement field and/or index register are used, and whether auto-increment or auto-decrement is allowed. The above fragmentation of the MD together with incomplete specification of the various tables in the available documentation makes it difficult to determine the exact class of machines this approach will handle, but it still seems strongly biased towards register architectures.

Cattell's memory model (1980, 1982) is highly structured, perhaps even rigidly so. The memory (including registers) is declared as a set of storage bases which are parameterised by the word length, the array length (one dimensional) and type, which is chosen from: program counter, primary memory, temporary location, reserved location or general-purpose location. Thus the primary memory is declared as a monolithic whole. The instruction fields are declared as contiguous sequences of bits, characterised by a fixed bit offset and bit length within an instruction word, a fixed word offset within the instruction and a type field specifying one of: op-code, operand selection control, or data expression. Since the word offset is fixed, Cattell caters for instruction fields whose position is variable by adopting the ad hoc rule that the position is conditional on earlier instruction words being generated. If not, they are omitted, and subsequent fields move up. In defining instruction semantics, values are associated with the instruction fields at the addressing mode level (operand class productions) or at the instruction definition level. Operator transforms are encompassed by including IR axioms which are then used to deduce (by means-end analysis) the appropriate operator transformations.

The Graham-Glanville-Henry scheme (Agrain et al., 1984; Henry, 1984) is divided into several phases: a prepass performs IR transformations; a special LR parser selects virtual instructions to match this transformed IR; and a postpass handles certain machine idioms. The IR transformations are provided as a set of options, each of which may be selected or not as required. The primary memory is not declared as such but certain properties are indicated by specifying the type correspondences between IR and TL. Address calculation is assumed to be explicit in IR, the machine's addressing modes being defined by a subset of the grammar fed to the LR parser table generator. The registers are declared as isolated entities parameterised by type, mapping to component registers, status (allocatable/dedicated), etc. The register manager which interfaces with the LR parser via the semantic actions supervises the allocation of registers together with the handling of register spills.

Lakos (1979) included declarations for a partial memory model even though it was specially tailored to the language BCPL (Richards et al., 1979). The memory was declared as monolithic, but the retargetable code generator assumed that three memory segments were used (to match a predefined intermediate representation of BCPL). Multiple register sets were allowed, and stacks could be
declared and referenced with a different notation to indicate automatic pushing and popping of the stack. (A subscripted reference to a memory area declared as a stack would be assumed to be a fixed reference, while an unsubscripted reference would be assumed to be a reference to the top of stack, with the appropriate popping or pushing implied.) As in ISPS, subfields and concatenations of the above memory areas could be declared and one could also declare variables for auxiliary purposes such as the specification of transforms. The notation was somewhat ad hoc and awkward, but with this partial model it was possible to generate object code as well as assembly code.

In many of the above cases it is not easy to determine the precise memory models implied or allowed by the machine descriptions. However, they generally assume a single set of general-purpose registers, and where memory references are expanded in IR it suggests a monolithic memory (as opposed to a segmented one). They do not therefore appear to cater for machines where the memory is segmented even at the machine code level (Burroughs, 1972), for those with multiple register sets, or for those with a hardware stack. Some of these coding issues can be relegated to the assembler-memory can be segmented by including location counter directives; and limited span branches may be processed as a standard feature. Of course, if the assembler does not handle these matters, it may not be possible to define the machine. The problems become even more acute if one desires to generate object code, for then the assembler facilities are no longer available and additional issues will surface, such as the need to pad instructions to word boundaries and the need to cater for rigid page limits.

For reasons such as the above, we argue that machine limitations intrude at too many points for the generation of object code to be completely ignored. Some code generation decisions do not make sense except in this context and perhaps that is why the memory model has received more attention in retargetable code optimisers (Davidson et al., 1980; Giegerich, 1983).

4. THE BASICS OF THE PROPOSED MEMORY MODEL

Having argued in general terms for the inclusion of an explicit memory model, we now propose a specific formulation. We observe that in the context of retargetable code generators the memory model needs to be flexible (to be widely applicable) and relatively easy to analyse (so facilitate the production of the CG). It is on this latter ground that we eliminate ISPS (Siewiorek et al., 1982). It is certainly flexible, as indicated by its wide usage, but it is a procedural notation and hence difficult to analyse. We note, for example, that the rigid page boundaries of the PDP-8 do not even need to be declared explicitly in ISPS (Siewiorek et al., 1982) since they can be hidden in the address calculation process.

4.1 Memory classes

The first aspect of our model is to distinguish three memory classes:

(a) The Register class

Each element of this class will have its current contents tracked by the CG in an attempt to optimise register usage. Individual registers may be flagged as reserved or dedicated, in which case they will not be automatically allocated for holding temporary results.

(b) The SMemory class

Memory areas declared to be of this class will be statically initialised by the CG. A location counter will be associated with each memory area in this class, indicating the next free location to be initialised, and thus also the amount of code or data already generated. During code generation space will need to be allocated for these memory areas, and the initialised portions will constitute the object code (if this is to be produced as output by the CG). Normally, we would preclude the generation of code which would alter these memory areas during run time but due to the primitive nature of some machine instruction sets, it may be necessary to override this check with a code generator option.

(c) The DMemory class

Memory areas declared to be of this class will be dynamically modified at run time. A location counter may be associated with each of the memory areas of this class, in which case it is assumed that the location counter acts like a stack pointer, delimiting the free space from the used part of the area. Because memory areas of this class are not statically initialised by the CG, space does not need to be allocated for them during code generation, apart from the appropriate descriptors.

4.2 Segmented memory areas

The second aspect of our memory model is to cater for the declaration of multiple memory areas of each class, the number (together with their names) being determined both by IR and TL. The declarations will relate to one compilation unit whether this is an Ada package (United States Department of Defense 1983), a Modula-2 module (Wirth, 1982), or some other division. This approach gives a segmented view of memory which matches commonly used intermediate representations and loader interfaces much better than a monolithic model.

In addition to allowing multiple memory areas we allow the areas to be multiply-dimensioned (at least in the case of SMemory and DMemory). We distinguish between uniform and non-uniform dimensions. In terms of a two-dimensional structure, such as a matrix, uniform dimensions would entail each row having an identical number of columns. By contrast, non-uniform dimensions would allow each row to have a distinct number of columns. Since space is allocated for the SMemory class at code generation time, the dimensions of areas in this class are assumed to be uniform and are used to indicate hard boundaries — word boundaries, page boundaries, etc. With the DMemory class, however, it is convenient to allow the dimensions to be non-uniform. In this way, the
4.3 Aliases

The third aspect of our memory model is to allow the explicit declaration of aliases. So far (in Section 4.2) we assumed that all the memory areas are disjoint, and hence there are no aliases except for the ones we now explicitly introduce. Aliases serve two roles — they either define an identifier to serve as a simple reference to a single memory element (thus simplifying the pattern matching required during code generation), or they define a different representation of data. In both cases, aliases imply that the memory is held in common but only in the former case does the CG assume the relationship between the original and derived representations. Aliases thus cater for the definition of subfields and concatenations, as well as providing the ability to define different data types in the same width field. We do not allow aliases to combine elements from different memory classes. The implication of aliases is considered further in Section 6.

5. EXAMPLE AND DISCUSSION OF THE MODEL

As an example, we declare the memory model required for implementing BCPL (Richards et al., 1979) on the PDP-11 (Digital Equipment, 1971). Both the language and machine have been deliberately chosen to be simple for illustrative purposes. The predefined IR of BCPL (viz. OCODE) implies four memory areas for the program code, statically initialised data, a local stack frame, and a global communication area (which stores globally accessible variables and procedure addresses). The usual scope rules are followed but local variable access is restricted to the stack frame of the current procedure, not the enclosed procedures. The data types encompassed by BCPL are the word and the byte. BCPL thus serves as a simple subset of C (Kernighan et al., 1978). Accordingly, the declaration of memory might then be:

```
Register R0..7(15..0) % 8 basic registers of 16 bits
Field R(0..7)(7..0) (R(0..7); R(0..7)), % Byte form of register
Md(Nb:0..7)(31..0) (R(Nb)*2,R(Nb)*2+1)), % Double-length registers
SMemory Prog,Prel(PC:0..2000)(15..0), % Program code + relocation
Stat,Encl(SW:0..200,SW:1..11)(7..0), % Static data + relocation flags, defined in terms of bytes
Field Op = Prog(PC), % Op-code may overlap other fields
Sm = Prog(PC)(11..9), % Source addressing mode field
Sr = Prog(PC)(8..6), % Source register field
Sw = Prog(PC)(4..2), % Max. word for source addressing
W(15..0) = (Stat(SW,Sb), Stat(SW,Sb+1)), % Next static data word
SMemory Global(1..401), % Global communication area
Local(DP:0..5000)(15..0), % Local stack frame
Reserve R(5),R(6),R(7), % Reserved registers
```

In the memory declaration, non-contiguous ranges are allowed for each dimension. Thus

```
Prog(PC:0..1000,2000..30000)
```

would have been acceptable. Dimensions omitted earlier in the list of memory areas are assumed to be the same as the following one. Note also that one location counter, PC for example, can be used to record the current position in more than one memory area.

In declaring the byte and double-length forms of the R registers the use of the variables Nb and Nd, respectively, make the aliases explicit and easy to determine.

Rather than require the user to specify rigid bounds for the memory areas (especially the SMemory areas), we propose to allow the parameterisation of the MD and hence of the CG, but the maximum range for the possible values of the parameters must be known, so that range checks can be incorporated into the CG tables (cf. Section 7).

We generalise the concept of an instruction field (while retaining the term) to include any field which is defined to be a fully indexed SMemory component. The concept can apply equally to executable code and data. Code (or data) is then generated by associating values with the instruction fields during code selection and these values are then recorded in the relevant SMemory area (see further in Section 6.2).

Having made the memory model explicit, some interesting questions arise which may not have been apparent before. In the first place, if a register can be viewed both as a composite whole or as a set of subfields, one is led to ask which should be declared as primary and which as secondary (i.e. aliases)? For example, should the PDP-11 registers above be primarily declared as 8-bit, 16-bit or 32-bit units? We assert that the appropriate choice is determined by the independence of the subfields. If the subfields are not independent, as in the case of the PDP-11 register bytes, then the concatenated structure is primary. But if the fields are independent and cover the whole structure, which is true in the case of the PDP-11 register words (as opposed to the double-length registers) then the fields deserve to be declared as primary. Of course, if different sets of independent fields can cover the same register, then the decision is not so clear-cut. Our convention above has the attractive property that the CG will never have to track two simultaneously active subfields of a register.

A second, somewhat similar question arises in declaring the static data area above. We might consider all the following alternatives as possible:

```
Stat(0..2000..11)(7..0)
Stat(0..401)(7..0)
Stat(0..2000)(15..0)
```

Here our preferred option is determined by the hardness of the word boundary which, in turn, will be determined by the instruction set. Note, however, that if several hard boundaries are appropriate, e.g. byte, word, double word, quad word, etc., then the second alternative might be the simplest to work with. The choice is not as critical as that for registers, since the decision there impinges on register allocation.

The possibility of multiple dimensions for these memory areas raises a subsidiary point. If an alias makes use of word boundaries, it may arise during code generation that a subscript goes out of range. In the definition of the field W in the example above, if the location counter SB...
has the value 1, the alias will fail since there is a reference to \text{Stat}[SW,SB+1]. We propose that in this case, the code generator will seek to code the predefined, one-operand command `padding`, the argument of which specifies the memory location to be padded. Thus different padding can be supplied for data, for code words, and even for page boundaries.

A third question which arises with an explicit memory model is whether labels should be explicitly declared or left implicit. In response, we note that labels are usually implicit in object code whereas they are normally explicit in assembly language, almost like a label table appended to (or at least interspersed with) the code. The label table can thus be viewed as an initialised segment of memory, and hence we advocate the explicit declaration of labels in the MD. The label table can be declared as a memory area of the SMemory class (since it is statically initialised) with the setting of the location counter of the area determining how much is initialised. We have opted for this solution for various reasons: if the labels are not explicit in the machine description then it would be difficult to generate label definitions in assembly language without awkward conventions; and by making the labels explicit, the user can define the associated memory areas in such a way as to take advantage of the particular properties of labels in the given IR and TL.

6. THE MACHINE DESCRIPTION LANGUAGE AND ALIASES REVISITED

The above memory model is to be incorporated in a machine description language where expressions and commands are defined using an extended form of condition with three components:

\[
guard \rightarrow \text{pattern} \rightarrow \text{result}
\]

The guard is a condition on SMemory components, usually instruction fields, which then restricts the applicability of the pattern for code selection. The pattern specifies the expression (or command) which this alternative can be used to code. The result is a formatted string or a replacement expression (or command) indicating assembly output or an IR transformation respectively.

Thus, for example, the PDP-11 source addressing mode might be declared by the following function:

\[
\begin{align*}
\text{Src}() = \text{cond} &\{\text{Sm}=0 \rightarrow \text{R}[\text{Sr}] \rightarrow \text{"RNN"}[\text{Sr}], \quad \text{Register direct} \\
&\text{Sm}=1 \rightarrow \text{R}[\text{Sr}] \rightarrow \text{"RNN"}[\text{Sr}], \quad \text{Register indirect} \\
&\text{Sm}=6 \quad \text{& Sk=S} \rightarrow \text{Local}[\text{Sw}/2] \rightarrow \text{"W[(N)(N)]"}[\text{Sw},\text{Sr}], \quad \text{Local cells}\}
\end{align*}
\]

Note that on the PDP-11, addressing mode 0 provides direct register access, mode 1 provides indirect register access, and mode 6 specifies indexed addressing, i.e. the specified register is added to the next instruction word (denoted above by \text{Sw} for the source mode). The indexed addressing mode is given an interpretation above which allows access to Local variables.

The ADD instruction and the transform for EQV given in Section 2 could then be declared:

\[
\begin{align*}
\text{cond}(\text{Op}=1000000) \rightarrow \text{Dest} := \text{Dest} \oplus \text{Src} \rightarrow \text{"ADD \#S, \#S"}[\text{Src},\text{Dest}] \\
\text{cond}(= T1 \rightarrow \text{T2 EQV T3} \rightarrow T1 := [T2 \& T3] \& -[T1 \& T3])
\end{align*}
\]

Note that the same conditional form serves to specify both assembly code output and IR rewrites, the object code output being specified by the setting of instruction fields. Note also that the conditional notation is extended with the definition of functions which thus enables a neat factoring of the machine description. In matching the patterns, a reference to a function will result in the selection of one of the alternatives. Subsequent reference to that function in the result component of the conditional will use the corresponding result component from the function. We also cater for the parameterisation of functions (by instruction fields) and hence one function definition with two instantiations would be adequate for defining the source and destination addressing modes of the PDP-11.

6.1 Pseudo-memory components

In the above example, Local[Sw/2] occurs in a pattern component of function \text{Src}. It simplifies the matching process to require simple indices to memory components and to provide for this we allow for the definition of pseudo instruction fields. These are SMemory fields declared as above except that they are not aliases for a primary memory area. During code selection they behave just like genuine instruction fields but when it comes to generating object code, the field value is simply discarded. Thus, in the above case, the SMemory Field declaration would be augmented by

\[
\text{Sn}(14..0),
\]

and then the addressing of Local cells could be specified by

\[
\text{Sm}=6 \quad \text{& Sw=S} \rightarrow \text{Local}[\text{Sw}] \rightarrow \text{"W[(N)(N)]"}[\text{Sw},\text{Sr}],
\]

Now, the pattern specifies a simple memory index and the setting of the field \text{Sw} is easily determined.

The concept of pseudo-memory, once admitted, finds application in other contexts. It serves as a convenient conceptual stage between IR and TL even though it actually occurs in neither. It is akin to the operators in region \text{f} of Figure 1. Location counters are assumed to be pseudo-memory, unless otherwise declared, and pseudo fields can be used to specify the position of actual instruction fields. For example, the position of the \text{Dw} field in the PDP-11 could be specified as

\[
\text{Dw} = \text{Prog}[\text{PC+Sk+1}]
\]

where \text{Sk} is a pseudo field set to 0 or 1 in the relevant source addressing function depending on whether the \text{Sw} field is absent or present.

The concept of pseudo-memory would also prove valuable with the Register class of memory. In the specification of the transform for EQV above, \text{T1}, \text{T2}, \text{T3} can be considered as pseudo registers. They are declared as register fields but, as with pseudo instruction fields, they are not aliases of any primary memory area. In instruction coding, any expression (of appropriate width) can be moved into the pseudo register at zero cost, and any subsequent refer-
ence to the contents of the pseudo register will result in the expression being returned.

6.2 Location Counters, Instruction Fields and Stacks

As we have already noted in Section 5, the concept of an instruction field is generalised to include any fully indexed SMemory component. The concept thus applies equally to executable code and data. A value is bound to an instruction field either directly in a guard, or indirectly through matching the pattern component of a conditional. This value is then used to augment the initialisation of the primary SMemory area (provided the field is not a pseudo field). If, as in the normal situation, the instruction field is an alias of the SMemory area at a position relative to the appropriate location counter(s), then these location counter(s) are automatically incremented after the generation of the instruction or data item. If this convention leaves an uninitialised gap in an SMemory area then it is set to zero, although one could argue that this situation should be flagged as an error.

The use of location counters is applicable to DMemory areas as well as to SMemory areas. Here, the concept of a current location is relevant to the definition of stacks and the top-of-stack element. We specify that a fully indexed DMemory component specified relative to the appropriate location counter(s) will be called a stack element. The autoincrement convention we introduced for instruction fields above is now augmented by a corresponding one for autodecrement. The stack-like behaviour can be used by the CG to determine the free locations in DMemory areas and hence the temporary storage where registers may be spilled. Unlike instruction fields, stack elements are assigned values as a result of code generation (not by the processing of guards). We adopt the obvious convention that references to the top elements of a stack in the current location counter(s) will result in a push of the stack, while references on the right hand side of assignments will result in pops. Absent stack elements should be reported as erroneous.

7. IR REWRITES

Agrain et al. (1984) distinguish two classes of IR rewrites required in their system. They refer to these as tree-to-tree and tree-to-forest rewrites. We note that the former are easily accommodated in our machine description (cf. the examples of EQV and RSHIFT in Section 2). In accommodating the latter, we note that tree-to-forest rewrites lie at the heart of code generation since the one assignment statement will generally need to be coded by several instructions, each one being a tree in its own right. We thus observe that any subtree is automatically pulled as another tree in the forest if the operator which is its root does not occur as an internal node of any instruction pattern. (If it does so occur, it may or may not be pulled, depending on the context at code generation time.)

8. PROCESSING THE MEMORY MODEL IN THE MACHINE DESCRIPTION

While we intend to consider the detailed processing of the above style of machine description in a future paper, we here indicate what is required with respect to the memory model. In processing the three-component conditionals the first step is to analyse the guard into a sequence of conjuncts, where each conjunct specifies the condition to be satisfied by an instruction field (referred to as a field condition), or the condition on a more general SMemory component. Each field condition is indicated by a set of possible values, an optional auxiliary constraint dependent on other fields or SMemory values, and the list of those fields on which it depends. Since a field used as a subscript in the pattern implies a restriction on the set of possible values adopted by the field, the pattern will also contribute to the conjuncts of the processed guard.

Although we allow fields to be mutually dependent (and hence the need for the auxiliary condition and dependency list above) we anticipate that this will rarely occur, except where pseudo fields are used. Thus, in function Src() of the PDP-11 example, the alternative expressing the access to local memory cells, viz.

\[ Sn \in \{0..5000\}, Sm \in \{6\}, Sr \in \{5\}, Sw \in \{0..10000\} : Sw=Sn*2 \]

would be analysed into the following list of conjuncts

\[ Sn \in \{0..5000\}, Sm \in \{6\}, Sr \in \{5\}, Sw \in \{0..10000\} : Sw=Sn*2 \]

The conjunct for Sw specifies that Sw assumes a value in the set \(\{0..10000\}\), being also constrained to satisfy the condition \(Sw=Sn*2\), which depends on the field \(sa\).

Having transformed the guards into this form, it is a simple matter to check the compatibility of two guards. This is of crucial importance in eliminating dead alternatives and hence reducing the size of the CG tables and the time spent in code selection. This feature can be exploited in the machine description in the expression of type crossing. Consider, for example, the PDP-11 where many instructions are applicable to either bytes or words. To ensure that the source and destination addressing functions are compatible, we would normally define one version of these functions for addressing bytes and one for words. The above compatibility analysis means that we can get away with one version of each without any code generation time overhead, provided that the byte/word alternatives are distinguished by an extra conjunct, say \(=0\) for words and \(=1\) for bytes where \(B\) is another pseudo field.

The compatibility analysis also allows us to determine accurately the set of side effects associated with an instruction. Considering the destination addressing for the PDP-11, we might have

\[
\text{Dst(1)} = \text{cond(Dm=0) \rightarrow R[Dr] \rightarrow "R[M]"(Dr),} \\
\text{Dm=1 \rightarrow IR[Dr] \rightarrow "R[M]"(Dr),}
\]

Since the guards are mutually exclusive we know that only one destination is active at a time and hence there are no side effects (at least for the alternatives noted above).

The above processing of guards is a necessary precursor to the production of coding tables, whatever the algorithm chosen. In this regard, our preference lies with a
Including an Explicit Memory Model in a Machine Description


modified form of the bottom-up tree matching algorithm of Hoffmann and O'Donnell (1982). Hatcher and Christopher (1986) present one adaptation of that algorithm for use in code generation. Neither algorithm can be used directly with our machine description language because of the presence of guards, and because the one addressing function or memory reference may occur more than once in a given pattern, thus making the operands mutually dependent. A serious drawback of this style of algorithm is the possible exponential explosion in the size of the matching tables, a problem which is exacerbated in our scheme by the proliferation of possible operands given our segmented memory model. To overcome this problem, the effects of independent patterns must be contained (see Hatcher and Christopher, 1986), and the machine description will have to be factored (as in Henry, 1984). This latter aspect is already encouraged by the ability to declare functions, but may be limited by the lack of orthogonality in the machine design.

9. CONCLUSIONS AND OPEN QUESTIONS

The above memory model has a certain affinity with ISPS, which is not undesirable considering the wide currency of that language. On the other hand, our notation is declarative in contrast to the procedural nature of ISPS, making it more amenable to processing. It caters for segmented memory and hence multiple register sets. This means that memory references to different areas (such as the local stack frame) are readily identified as such, and hence it will be possible to optimise register references across such memory references. By a simple extension of the location counter concept, the model caters for the specification of hardware stacks. Since object code can be generated, limited span branches can be specified in the machine description. The ability to define pseudo-memory components means that one can require memory references occurring within patterns to have simple indices (thus simplifying pattern matching). Pseudo-memory components also have a simple application to the specification of IR rewrite rules.

The fact that the memory model is systematic and explicit in the machine description means that many code generation issues can be handled in a clean, regular way, in contrast to previous approaches. Thus, the direction of stack growth and the method of access (whether by display register or static chain) is indicated by the declaration of the appropriate memory area and by suitable conditions on the instruction fields. This contrasts with the approach of Ganapathi where these issues need to be specified in separate tables (Ganapathi and Fischer, 1985). Similarly, the guards which limit the applicability of the patterns and the result components which produce the output are completely specified in the one machine description, rather than being disguised in the separate coding of disambiguating predicates and action symbols. In the same way, the location of instruction fields within an instruction can be precisely defined even when this depends on the settings of other instruction fields. It is not necessary to include ad hoc rules such as in Cattell (1980, 1982).

At this stage, the prepass which analyses the machine description has been written. This analyses the guard and patterns, and determines the coding alternatives provided by each instruction description. Preliminary results indicate that the proposed machine description with an explicit memory model can be used to describe traditional architectures such as the PDP-11 (Digital Equipment, 1971), ad hoc architectures such as the Z80 (Zaks and stack machines such as the B6700 (Burroughs, 1972) (though we admit that the use of RCGs for stack architectures is somewhat questionable). It was interesting to observe with OCODE that the transforms required in the PDP-11 description mainly revolved around IR operators not present in TL, whereas the B6700 description primarily required transforms for memory access.

The model, as expounded above, leaves some questions unresolved. One question relating to IR rewrite rules concerns the compensation required for tree-to-forest rewrites which pull operators with side-effects, e.g. increment and decrement in C (Kernighan et al., 1978). Here we observe that operators with side effects are not strictly operators in the sense assumed by our MD, and hence we would not allow such side effects except perhaps as internal assignments. We have not considered how to cater for these during code generation. We also concur with Agrain et al., who noted, concerning the IR of the portable C compiler (PCCIR), that some of the transformations we use in our implementation alleviate problems peculiar to PCCIR and may not be required in different IRs.

A second question concerns whether we wish to use declaration of aliases to cope with the multiplicity of source language types (as in C, Pascal, for example), or whether we wish to have some machine dependent reduction performed first, such as is done in the Portable C Compiler (Johnson, 1977). At stake is a compromise between including machine dependencies in the first phase of the compiler, or overburdening the code generation phase.

A third question relates to the complexity of the guards which restrict the applicability of the patterns. This complexity does much to determine the coding algorithms which can be employed. The guards may all be quite simple, with each field being associated independently with a set of values. In this case the description would qualify for input to syntactic matching algorithms (Henry, 1984; Agrain et al., 1984). Alternatively, the guards may be complex enough to specify the patching of label references in object code. In this case, a syntactic matching algorithm would be inadequate and the detection or processing of dynamic blocking situations would be important (Glaville, 1977). Ganapathi’s use of disambiguating predicates would not be sufficient since that assumes there is a fall-through alternative with no constraints for coding each operator (Aho et al., 1985). The problem is that some blocking situations (such as the inability to redefine a label) may not need to be handled since they can be guaranteed never to arise on the basis of assumptions concerning the IR.

The range of issues noted above suggest that it may be
more appropriate to specify several descriptions, with specially-tailored processors and algorithms for each phase. Thus IR rewrites might be specified with one description, code selection with another, register allocation with another, etc. However, we assert that whether a single description or a family of descriptions is used, there is much to be gained by having the one machine description language with an explicit memory model so that all code generation issues can be clearly identified as such in the self-contained description(s).

ACKNOWLEDGEMENT
The author wishes to acknowledge the care with which the reviewers examined this paper. This is reflected in a significant improvement in the organisation and description of the material.

REFERENCES
JOHNSON, S.J. (1977); A Tour through the Portable C Compiler. Bell Laboratories.
ROBERTSON, E.L. (1979); Code Generation and Storage Allocation for Machines with Span-Dependent Instructions. ACM TOPLAS 1, 1.
SZYMANSKI, T.G. (1978); Assembling Code for Machines with Span-Dependent Instructions. CACM 21, 4.
ZAKS, R. Programming the 280, Sybex Library.
An And-Parallel Dataflow Model For Logic Programs Based On Mode Prediction

T-C. Chung and J-W. Cho†

Logic programming offers several kinds of parallelism for executing programs. It is natural to associate logic programming with a dataflow architecture to cope with the high parallelism. Since the input/output relationships of arguments are not fixed in such programs, run-time support to exploit the AND-parallelism of logic programs in a dataflow model requires control of the dynamic token flow, which induces much overhead. In order to remove the run-time reordering of subgoals, that is, to avoid using dynamic arcs, and to exploit AND-parallelism to some extent in a dataflow model, we adopted the method of predicting the dependency relation between subgoals at compile time.

Hitherto there has been no evidence that the dataflow model has been successfully employed to exploit AND-parallelism for pure logic programs. In this paper, we propose a model named the 'dataflow model for predicted AND-parallelism (DAMPA)' in which dataflow graphs supporting AND-parallelism for logic programs can be produced at compile-time through a refined static data dependency analysis and a graph generation procedure.

Although the dataflow graph in the model does not provide optimal AND-parallelism, there is no run-time overhead such as dynamic arcs between the literal nodes and no join operation between the literals. OR- and argument-parallelism are also considered.

Keywords and Phrases: logic programming, AND-parallelism, argument-parallelism, mode prediction, literal dependency, dataflow graph.
CR Categories: C.1, I.2, D.3.

1. Introduction

PROLOG is now considered as an attractive logic programming language with its high potential for parallel execution. A program is assumed to be a set of Horn clauses where a Horn clause is a restricted form of general clause of the first order predicate logic. A program can also be described as a set of procedures and a goal statement. A procedure stands for a set of Horn clauses which have the same head. In general, a clause is written in the form

\[ P : A_1, A_2, \ldots, A_m, \ (m \geq 0), \]

where \( P \) and \( A_i \)'s are literals (or atoms). \( A_1, A_2, \ldots, A_m \) are connected by a logical conjunction. \( P \) and the \( A_i \)'s are called a head literal and body literals, respectively. A clause without a head is called a goal. A literal has the form of \( R \ (t_1, t_2, \ldots, t_n) \), where \( R \) is an n-place relation name and, for \( i = 1, 2, \ldots, n \), \( t_i \) is a term (or an argument). A term may be a constant, a variable, or a function.

The execution of a Horn clause logic program is conceptually the search of an AND/OR tree based on the resolution proof procedure. In executing the logic programs, several kinds of parallelism such as OR-, Stream-, Argument-, Search-, and AND-parallelism have been found (Conery, 1983). AND-parallelism is the major concern in this paper.

There are two typical parallel execution models for the logic programs which do not assume any underlying architecture. They are the reduction model (Haridi and Ciepielewski, 1983) and the AND/OR process model (Conery, 1983). The reduction model deals only with OR-parallelism, since a goal is rewritten for each literal. But new goals are independent of one another. The AND/OR process model solves a logic program using a set of AND- and OR- processes that communicate via messages. Thus the process model deals with AND- and OR-parallelism, but each process is not independent of the AND relation of subgoals.
There are some parallel execution models which assume particular architectures and some of them consider dataflow architecture as the underlying architecture. It is natural to associate logic programs and the dataflow architecture because the parallelisms in both logic programs and dataflow architecture are very high. There are two types of execution model for dataflow implementations: the procedural model (Ito, Shimizu, Kishi, Kuno, and Rokusawa, 1985) and the proof tree model (Hasegawa and Amamiya, 1984, and Umeyama and Tamura, 1983). From the viewpoints of parallelisms treated, and of advantages and disadvantages, they are similar to the AND/OR process model and the reduction model, respectively.

It is difficult to judge the superiority of the procedural model (or AND/OR process model) to the proof tree model (or reduction model). There may be a problem with the tradeoff between the benefits of exploiting AND-parallelism and the control overheads incurred. In this paper, the procedural model of AND-parallelism is considered.

AND-parallelism involves the simultaneous execution of two or more subgoals in a clause. This raises the binding conflict problem. Suppose two subgoals share a variable and that they execute in parallel. If they attempt to bind the variable to two different values, a binding conflict results. Two major approaches have generally been taken to solve this problem: the join technique and the reordering technique. In the former, each subgoal generates solutions independently, and then these values are joined so that each shared variable can have the same value. It has a major disadvantage in the loss of performance by the join operation if the subgoals have many answers. In the latter case, the literals sharing the same variable are divided into a generator literal and consumer literals of the variable at run-time. Only one generator literal is chosen for each variable. Then the generator literal produces the values for the shared variable, which are tested and consumed by the consumer literals. All literals not containing any shared variables are executed in parallel. This technique, adopted by Conery (1983), has a major disadvantage in run-time reordering overhead. Although the latter has reordering overhead, the join operation has been avoided in AND/OR process models (DeGroot, 1984, and Chang, 1985).

Since the dataflow graph is produced by the compiler, adopting the reordering technique in dataflow model requires dynamic arcs which induce heavy overhead in processing time and required space. Therefore, the procedural model which adopts the join technique may be efficient as shown by Ito et al. (1985). But, the procedural model with the join technique inherits the disadvantage of the join operation. In this paper, a procedural model is proposed which avoids the join technique and does not use dynamic arcs. In order to avoid the latter, (1) the input/output nondeterminism of a clause in the logic programs should be reduced, or (2) the subgoals in a clause should be resolved sequentially. In our model, mode is detected first at compile-time to reduce the input/output nondeterminism. And then remaining nondeterminism is resolved by another procedure, which is also proposed here. This, however, sacrifices the exploitation of AND-parallelism somewhat.

The mode is the information which restricts the nondeterminism of the logic programs. There are two approaches for obtaining the mode for AND-parallelism: the annotation approach and the non-annotation approach. The former explicitly describes the mode information in the logic program (Ito et al., 1985). The latter obtains the mode information in compile-time (Chang et al., 1985, and Mellish, 1981). The annotation approach requires users to have some knowledge of the particular computational model. In this paper, a non-annotation approach is taken. If the mode generated by a compiler is not restrictive for the nondeterminism of the logic programs, the non-annotation approach may restrict the AND-parallelism. Therefore, the restrictive mode should be generated.

For mode detection by the non-annotation approach, Chang et al. (1985) proposed the static data dependency analysis and showed that it can derive producer/consumer relationships between literals from each clause at compile-time. Working on the static relationships eliminates the overhead of ordering or reordering of literals at run-time. Instead, a somewhat restricted AND-parallelism can be achieved. In order to provide the starting point of static analysis, the programmer must provide only one declaration for the properties of arguments of the procedures which may be a query. Chang et al. (1985) and Mellish (1981) claimed that this task is not unreasonable for programmers. We refine Chang's static data dependency analysis to define a more precisely predicted mode, and apply the generated mode to the generation of dataflow graphs which exploit AND-parallelism without dynamic arcs. The model that we propose is called the dataflow model for predicted AND-parallelism, or DAMPA. Figure 1 shows the differences between DAMPA and other models. Until now, there has been no evidence that the procedural dataflow model could be successfully employed to exploit AND-parallelism for pure logic programs.

In the following sections, we first describe the refined static data dependency analysis and then present the procedure for generating the dataflow literal dependency graphs. Transformation from a dataflow literal dependency graph to a dataflow graph is described next. Finally, we compare DAMPA with other models.
An And-Parallel Dataflow Model For Logic Programs

2. Refined Static Data Dependency Analysis and High Level Dataflow Graph

2.1 Static Data Dependency Analysis

In the ordering algorithm proposed by Conery (1983), whether a variable has a ground value, or is independent of other variables or not at run-time, is very important. The variables in a clause can be classified into three groups at compile-time: ground, independent, and coupled. This means that the variables will be bound to grounded terms, remain as independent variables, and be coupled with other variables at run-time, respectively. A variable belongs to only one group, yet it can change from one group to another as they are processed. The group to which a variable belongs may be restricted beforehand using the user-supplied mode.

Now let us consider the static data dependency analysis proposed by Chang et al. (1985). They insist that the mode of a clause can be obtained using the static data dependency analysis at compile-time, if a programmer supplies only a little information, i.e. the mode of each procedure which may become a query. In other words, static data dependency analysis can earn the exit mode of a procedure from an activation mode. The mode of an entry point is declared as a form of entry (procedure-name, [model, mode2, ... , mode.m]). The modes may be represented for each argument or each variable, and they are represented as ? (ground), # (independent), % (coupled), and %N (coupled in N-th coupled group).

An activation mode and the relationship between an activation mode and the exit mode produced by the static data dependency analysis for a clause or a procedure are represented, respectively, as follows: activate (clause or procedure-name, activation-mode), and mapping (clause or procedure-name, activation-mode, exit-mode).

Now, let us explain how to obtain the exit mode of a clause for an activation mode. If two procedures \( h(X, Y) \), \( g(X, Y) \), \( g(X, a) \), and \( g(a, a) \), and an activation mode of the clause activate(\( h \), [\#, \#]) are given, the analyzer can find that the exit mode of the clause is mapping(\( h \), [\#, \#], [\#, ?]) because the clause \( g \) makes the variable \( Y \) from independent to ground. The exit mode of the variable \( X \) is not ground because the exit mode of a procedure is decided by choosing the worst mode among the exit modes of the clauses of the procedure. (By worst we mean that the exit mode that contributes least to exploiting AND-parallelism. The ground mode contributes best and the coupled mode worst.)

2.2 Refined Static Data Dependency Analysis

In order to obtain the exit mode of a clause from an activation mode, it is necessary to select one literal at a time for processing. Chang et al. employed the left-to-right literal selection strategy for selecting each literal for processing. Such a strategy is simple but may miss some opportunities for AND-parallelism. Figure 2 shows such an example.

In Figure 2, if the left-to-right strategy is applied, the exit mode of the clause is mapping(\( f \), [\#, \#], [\#, \#]). But, if one selects the literal which generates the largest number of ground terms as the exit mode, then the exit mode of the clause becomes mapping(\( f \), [\#, \#], [\#, ?]). In view of its contribution to AND-parallelism, a ground term is superior to the terms in the other group, and an independent variable is superior to the coupled variables. Therefore, the static data dependency analysis itself may overlook some opportunities for exploiting AND-parallelism. We propose a refined static data dependency analysis to improve AND-parallelism. Although the refined
static data dependency analysis requires additional
time for compilation, this overhead can be justified if
the number of compilations is much less than execu-
tions.

In the refined static data dependency analysis, a
maximum grounding literal selection strategy is used
to select the next literal. The maximum grounding
literal selection strategy selects a literal first whose
activation mode of each variable is grounded, and
then selects the literal which has the maximum
grounding factor of mapping. When these strategies
give rise to more than one literal as a candidate, the
leftmost literal is selected. The grounding factor can
be defined as:

\[(\text{the number of ground variables in exit mode} - \text{the number of coupled variables in exit mode}) - (\text{the number of ground variables in activation mode} - \text{the number of coupled variables in activation mode})\]

There are many PROLOG systems with extended
core facilities. The proposed control primitives
and heuristics have been devised to minimize the size
of the search tree, or to detect failure quickly, or to
avoid the creation of failure branches (Naish, 1985).
The maximum grounding literal selection strategy is
also a heuristic rule to get the above benefits.

The algorithm for the refined static data depen-
dency analysis has been omitted here. The outputs of
the refined analysis consist of the exit mode of the
clause for an activation mode and an ordered subgoal
list with the mapping relation of each subgoal literal.
For the example of Figure 2, the analyzer outputs the
exit mode of the clause, \([\text{mapping}(\text{ft}, \text{?}, \text{?}), \text{mapping}(\text{ft}, \text{?, ?})])\), and
the ordered subgoal list, \([\text{mapping}(\text{h(Z, Y)}, \text{?}, \text{?}), \text{mapping}(\text{g(X, Y)}, \text{?, ?})])\). In the next sec-
tion, the procedure for generating the dataflow literal
dependency graph from the ordered subgoal list of a
clause is presented.

2.3 Dataflow Literal Dependency Graph
In order to run a logic program on a dataflow
machine, the dataflow graph must be generated.
Therefore, the literal dependencies between the literals
and the relation between arguments or variables must
be identified to represent them in a proper fashion.
For this purpose, the dataflow literal dependency
graph is constructed from the ordered subgoal list
produced by the refined static data dependency
analysis. It consists of the literal nodes and the arcs
between the literals, each of which represents the
dependency relationship for a variable.

Informally, the procedure for generating the
dataflow literal dependency graph is as follows:

Procedure make-graph
1. The graph is constructed incrementally by adding a
literal which is selected one literal at a time from
the ordered subgoal list. The head literal becomes
both the first and the last node of the graph.
2. A selected literal is added to the partial graph by
drawing some arrow arcs from some appropriate
nodes in the partial graph for each variable in the
literal. An appropriate node is selected for each
variable in the literal as follows:
   2.1) Process from the variable whose activation
       mode is less grounded.
   2.2) If the activation mode of the variable is cou-
       pled or independent, select the graphically last
       node which includes the variable.
   2.3) For the variable whose activation mode is
grounded, if there is a node which has been con-
    nected already to the literal for other variables
and includes the variable, then select the node,
else select the graphically first node whose exit
mode for the variable is grounded.

End
In a clause, the head is used as a producer literal
which supplies a token to subgoals, and is also used
as a consumer literal which needs tokens from
subgoals. We split the head literal into a producer-
head and a consumer-head literal in the dataflow
literal dependency graph. Figure 3 shows a dataflow
literal dependency graph.

2.4 Transforming A Dataflow Literal Dependency
Graph to A High-level Dataflow Graph
For a dataflow literal dependency graph to be useful
for the dataflow machine, it must be transformed to a
high-level dataflow graph. In order to transform into
a high-level dataflow graph, we introduce two con-
cepts: coupled node and predicate arc.

The producer head literal, the subgoals, and the
consumer head literal should be transformed to a
Unify head-literal, Solve subgoals, and a Clause-
return head literal, respectively. A Unify node unifies
the input token and the head literal to produce the
unified output. A Solve node solves the subgoal and
outputs a stream of solutions for the input token.
A Clause-return node makes a stream of solutions for
the clause and returns it to the merge node in the

\[\text{p(x,J,J,[b,L]):q(j,L,r(L)),w(M,N),x(a,M)}\]

\[
\text{[easy(p, [%, %, ?]), mapping(q, [%, %], [?, ?]), mapping(x, [?, ?], [?, ?]), mapping(w, [?, ?], [?, ?]), mapping(u, [?, ?], [?, ?]), etk(p, [?, ?])]}\]

(a) A clause and the ordered subgoal list

(b) A dataflow literal dependency graph

Figure 3. Generation of a dataflow literal dependency graph
An And-Parallel Dataflow Model For Logic Programs

OR-graph. These nodes will be precisely described in the later section.

A coupled node more clearly shows the processing of an arc which represents a coupled variable. All the literals in a dataflow literal dependency graph which are in the coupled group become coupled nodes, and they are serialized in the high-level dataflow graph. In Figure 4, such a transformation is shown where the node denoted by a doubled circle is a coupled node. When serializing the literals, a coupled node whose exit mode is more grounded is positioned first, and then all the arcs for the variables in any the coupled literals are connected to the coupled node. In Figure 4, q and r are coupled, and then q is laid first because the exit mode of the variable l is grounded so that r can remain as a normal node.

Any node in high-level dataflow graph should have at least one output arc. For this purpose, there is a special arc called a predicate arc, which connects a node, which does not have an output arc, to the consumer head. A predicate arc carries only a token that represents true or false. In Figure 4, predicate arcs are represented by dashed lines.

3. Dataflow Graph and Token Format

3.1 OR-graph

In order to exploit OR-parallelism, all of the clauses in the procedure should be solved concurrently in a nondeterministic manner. In this case, multiple solutions may be returned to the goal in the order in which they are obtained. This is called don't-know nondeterminism.

An OR-graph corresponds to a procedure. Four special nodes are introduced for an OR-graph as shown in Figure 5. The copy node duplicates the input to each output arc. The merge node moves the tokens from its input arcs to its output arc in the order they arrive. There is a pair of nodes represented by enter and return which are used as an entrance and exit of a procedure, respectively. They receive a goal token and return a stream of solution tokens respectively, and are also used to save and restore the context of a token when a procedure invocation occurs.

3.2 Token Format

A token is a message between two directly connected nodes in a dataflow graph. It consists of a literal data and tag information for its flow control. A token is a tuple that consists of colour, destination, and operand. Colour has all the flow-control information, destination is the address to which the token should flow, and operand represents an operand or a list of operands.

There are three kinds of operands: atom, variable, and structured data. Operands are distinguished by the data type field. For a coupled variable, a shared bit of the variable in a token indicates whether the variable is shared or not. The scope of the shared bit is always limited to one clause.

3.3 Colour Field and Related Operators

In order to execute a dataflow graph, the interpreter needs to allocate a unique identifier to a token for each procedure instance and to maintain the history of the procedure invocations. In addition, AND-parallelism requires the functions of combining the solutions of the subgoals. Therefore, colouring (or token labeling) is used to portray the information, and the nodes to manipulate the colour are introduced in the dataflow graph. The colour field of a token consists of three subfields: context field, alternative solution field, and split tag field.

The context field has the unique identifier for the procedure instance to maintain the history of the procedure invocations. This field is manipulated by the enter and return nodes. The enter node generates and assigns a new unique tag value to the context field, and saves the context field of the calling token. The return node uses the context data constructed by the enter node to establish the proper tag and the destination of the returned token. The output arc therefore is a dynamic arc.

Since AND-parallelism permits the subgoals to be executed independently, their solutions should be combined in order to obtain the solutions which satisfy all the independently executed subgoals. The

Figure 5. An OR-graph
solutions should contain information on which this combination is based. The split tag field has this information, and split and comb nodes are introduced to manipulate the field. A split node is added at the AND-parallel execution point, and its function adds the information of the corresponding AND-parallel point to the split tag field of the input token. A comb node makes new tuples by combining solutions from the producer literals based on the value of the split tag field.

The number of solutions may not be one because DAMPA also exploits OR-parallelism. Therefore, each of the solutions has to be distinguished from each other by the alternative solution field. This field is also managed by split and comb. Figure 6 shows an example of this.

Figure 6. Dataflow graph for Figure 4

3.4 Share-variable and Shared-Variable

A share-variable is the variable which occurs at two or more places, whose binding value at one place should be checked with the values of the variable in other places. In general, there are three places where the share-variables may occur in logic programs: inter-subgoal level, intra-subgoal level, and head-literal level.

If two subgoals share a variable, the variable becomes an inter-subgoal level share-variable. As an example, variable X in p:- q(X), r(X) is an inter-subgoal level share-variable. An intra-subgoal level share-variable on the other hand appears in more than one place of a subgoal. A subgoal :- p(I, J), J, J) has the share-variables I and J in this level. A head-literal level share-variable is a variable that appears in many places in its arguments such as the I in a clause head p(I, J):-.

A share-variable at any place should have one binding value. One method is to solve subgoals (or unify arguments) concurrently first and then to check whether the binding values for the share-variable are unifiable. The other is to solve a subgoal (or unify a argument) first and to pass the binding value to the other subgoals (or arguments) and solve the subgoals (or unify arguments). Let us call the former method eager-evaluation and the latter lazy-evaluation, respectively, for each level. Eager-evaluation gains more parallelism than lazy-evaluation, but it suffers from join or consistency check overhead. There is a trade-off between parallelism and consistency check overhead.

Let us define shared-variable as the share-variable when the subgoals (or arguments) which include a share-variable are solved (or unified) concurrently. A share-variable in eager-evaluation becomes a shared-variable, while a shared-variable in lazy-evaluation remains a simple variable. Therefore, the shared-variable is the concept at run-time while the share-variable is the concept in view of the syntax. A shared-variable is distinguished by a shared-bit from a simple variable. In this paper, lazy-evaluation is adopted at the inter-subgoal level and the head-literal level, and eager-evaluation at the intra-subgoal level. The inter-subgoal level has been described already, but the other levels are treated in the next section.

3.5 Dataflow Graph Details

The detail representation of each node in a high-level dataflow graph and the management of streams are presented in this section.

3.5.1 Unify node

Among the parallelisms in the logic programs, argument parallelism is also adopted in this paper. Argument parallelism means that all arguments are unified concurrently and the bindings for each share-variable are checked to see whether they are consistent. The activation mode and the lazy-evaluation of the share-variable in the head-literal level greatly influence the construction of the dataflow graph for literal unification. Figure 7 shows an example graph for a head literal. If the predicted modes of arguments are coupled, the graph has CC nodes and sub nodes for the variables in the arguments such as the 4th, 5th, and 6th arguments in Figure 7. The lazy-evaluation for the head-literal level causes the unification nodes of the 5th argument and 6th argument to be serialized, and places the share node after body invocation as shown in Figure 7. This reduces the number of shared-variables whose shared-bits are set so that this reduces the cost of CCs.

The nodes in Figure 7 have the following functions:

(i) A unify node receives two tokens; an input token from the goal and the head argument token from the head argument of the clause. It produces a result token and a substitution token. A result token is the...
An And-Parallel Dataflow Model For Logic Programs

result of unifying two input tokens, and a substitution token is the binding pairs of the shared-variables. If one input is a shared-variable and the other is a term other than a simple variable, the node outputs the shared-variable to the r-port and a set of pairs of the shared-variable and its instance (i.e. the term) to the s-port. In other cases, the node outputs its instance to r-port and 'nil' to s-port. When the head argument is a simple variable, the unify node for the argument can be omitted.

(ii) A CC (Check Consistency) node checks for consistency among the bindings of the shared-variables. A CC node has two inputs and produces one output. Its output value becomes 'fail' if any of inputs is 'fail'. If one of the inputs is 'nil', the other input value becomes the output. If both inputs are lists, select one shared-variable and its binding instance from one list and then search the other list for the binding instance of the selected shared-variable. If the search fails, hold it temporarily and select the next pair. If such a binding instance exists, then call the unify primitive in order to check whether both instances are unifiable. If they are not unifiable, then the output value becomes 'fail' and the CC operation finishes, otherwise the bindings of the unify are added again to the binding list. After all the bindings for each variable in the input lists are completed, CC outputs the holding list and finishes.

(iii) The share node tests its input token to see whether there are simple variables and then changes all the simple variables in the instance into shared-variables. The substitute(sub) node tests the binding list; if it is 'fail' the node outputs 'fail' or if 'nil', the node outputs the instance directly; otherwise, the node tests the shared-bit of the instance and replaces the shared-variable by the binding instance of the variable. The AND node performs a logical 'and' operation.

3.5.2 Solve node
A Solve node solves the subgoal and obtains a stream of solutions each of which consists of a pair of solution instances and bindings of the shared-variables in the input token. The solution instance and bindings of the shared-variables are output to r-port and s-port, respectively. The bindings of shared-variables are used to substitute the variable in other subgoals. The details of the Solve node are shown in Figure 8.

The goal literal expression, \( L \), associated with a given Solve function, is a constant argument of the Activate node. This node, when enabled by the arrival of a token of input environment for the \( L \), constructs the goal token and sends it to the subgoal procedure and the Extract-output node. The second primitive operator is the Extract-output function. Its task is to construct a solution environment and the list of bindings of the shared-variables for each solution of the returning tokens from the subgoal by comparing them with the goal token, and outputs them to r-port and s-port, respectively. The Activate node and the Extract-output node also changes a variable in the intra-subgoal level into a shared-variable and into a simple variable, respectively. The shared-bit of the shared-variables in intra-subgoal level is set and reset by the Activate node and the Extract-output node, respectively.

The token from the s-port of a Solve node is used only at the coupled nodes. Therefore a coupled-Solve node consists of a Solve and a comb-sub node as shown in Figure 9. A comb-sub node combines the incoming tokens and substitutes the shared variables with bindings of corresponding shared-variables from the s-port output.
3.5.3 Stream and Storage Arcs
The Solve node generates a stream for an activation token. A stream is a list of solutions. The solutions are the same tokens except for the operand field and the alternative solution field. They are treated the same way at any node. The new streams are created at the Copy node for each input token and are merged into one stream at the Merge node in the OR-graph. In the AND-graph, streams are created at the split node for each input token and are merged into one stream at any Comb or Comb-sub node.

There are problems in handling a stream in comparing a single token with a stream of tokens for the case of an Extract node, and in comparing streams of tokens for Comb or Comb-sub nodes. The stream handling requires the concept of a storage arc (Halim, 1986). The storage arc is the arc on which a token is preserved upon firing. This is an alternative to recirculating the token upon firing. This, of course, represents a departure from the pure dataflow concept, but the use of storage arcs avoids token recirculation, thus reducing token traffic.

When a stream is managed by an Extract-output node, it also carries out the following functions in order to reduce the overhead for multiset, which means a set of identical solutions obtained from a goal. If the instances of all the variables which are required by the consumer subgoal(s) are grounded and they already have the ground values when they entered the subgoal, then only one solution is sufficient to the consumer. In such a case, the rest of solutions are cancelled.

4. Comparisons
DAMPA has both advantages and disadvantages compared to other models. The refined static data dependency analysis generates more restrictive mode than the static data dependency analysis (Chang et al., 1985). But it needs more compile-time. DAMPA proposed in this paper supports AND-, OR-, and argument-parallelisms, but the graph proposed by Umeyama and Tamura (1983) or Halim (1986) treats only OR-parallelism. The model proposed by Ito et al. (1985) requires costly join operations. Since the modes are predicted in DAMPA, even though it can avoid the join operations, it still exploits AND-parallelism.

5. Conclusion
A dataflow model conceptually requires a deterministic graph, but a logic program is nondeterministic. This conflict can be resolved by predicting the dependency relationships between subgoals at compile-time. We use a refined static data dependency analysis to predict the mode of arguments and suggest the method of generating a dataflow graph which exploits the AND-parallelism based on the predicted mode. The suggested model has no run-time overhead from dynamic arcs in spite of the absence of any user-supplied control information for AND-parallelism. OR- and Argument-parallelism are also considered, which are useful for the dataflow model. This paper also proposes a method of handling the multiset problem which appears frequently in the model that does not have a demand-driven facility. What remains to be done is to evaluate the refined static data dependency analysis and the proposed dataflow graph, and to construct a machine to run the graphs.

References
On the Analysis of Iterative and Recursive Programs

F. Mili†, B Alaya* and A. Mili*

Strongest invariant functions have proven useful in analysing some functional aspects of iterative programs. In this paper we investigate their relationship to fixpoints of recursive programs.

Keywords and Phrases: Iterative programs, recursive programs, invariant functions, strongest invariant functions, fixpoints, least fixpoints, program semantics.


1. INTRODUCTION: POSITION OF THE PROBLEM

Mili et al., (1985) introduce the notions of invariant functions and strongest invariant functions, and show how these two notions can be used to analyse iterative programs. Among their main findings, we mention:

— The observation that invariant functions are more expressive than the perennial loop invariants (Hoare, 1969; Manna, 1974; Gries, 1981) as a means to analyse iterative programs: in particular, one can derive a loop invariant from any given invariant function.

— The observation that strongest invariant functions of iterative programs can be derived systematically, given special forms of their loop bodies.

— The observation that strongest invariant functions can be used to generate most general loop invariants (Caplain, 1975), which are known to be most interesting in analysing iterative programs (Amy and Caplain, 1978; Dunlop and Basili, 1982).

— Finally, a constructive formula that yields the function computed by an iterative program from one of its strongest invariant functions.

Among the questions that were left open for further investigation in Mili et al. (1985), one appears to be particularly intriguing: elucidating the relationship between invariant functions, as a tool to analyse iterative programs, and fixpoints, as a tool to analyse recursive programs. Specifically, given an iterative program Pi and an equivalent recursive program Pr, can we define a mapping between invariant functions of Pi and fixpoints of Pr? Does the same mapping link strongest invariant functions of Pi to least fixpoints of Pr?

The purpose of this paper is to discuss this question, as well as some related issues. But first, and for the sake of readability, we introduce some elements of mathematical notation, as well as the main results of Mili et al. (1985).

2. ELEMENTS OF MATHEMATICAL NOTATION

The emphasis in this section is on readability: hence we may sometimes sacrifice rigour for the sake of making the definitions and notations easier to follow. A more formal treatment is given in Mili et al. (1985).

2.1 States and Spaces

Let P be a program on variables a, b, c, say of type integer. Any triplet of integer variables that (a,b,c) can take is called a state of program P. The space of P is the set of its states. Given a state s of program P, we denote by a(s), b(s) and c(s) (respectively) the a—, b— and c— component of s.

2.2 Sets and Relations

Let S be a set (usually a set of states, i.e. a space). A relation on set S is a subset of SxS. Among the constant relations on S, we define: the identity relation, I = {(s,s): s ∈ S}, and the Universal relation U = SxS.

2.3 Operations on Relations

In addition to the set theoretic operations of union and intersection, we define the following operations on relations.

The inverse of relation R is R^⁻¹ = {(s,s'): (s',s) ∈ R}.

The domain of relation R is dom(R) = {s: ∃ s': (s,s') ∈ R}.

The range of relation R is rng(R) = dom(R^⁻¹).

The product of relation R by relation R' is the relation R * R' = {(s,s'): ∃ t: (s,t) ∈ R & (t,s') ∈ R'}. Let A be a subset of S. The sub-identity of A is the relation I(A) = {(s,s): s ∈ A}. If A can be written as {s: t(s)} for some predicate t on S, then I(A) may also, for convenience, be written as I(t).

The prerestriction of relation R to subset A is the relation A ∘ R = I(A)^⁻¹ R.

The postrestriction of relation R to subset A is the relation R ∘ A = R * I(A).

†Department of Informatics, Faculty of Sciences, University Campus, 1002, Belvedere, Tunisia. *School of Engineering and Computer Science, Oakland University, Rochester, MI, 48063, USA. Manuscript received February 1987.

2.4 Properties of Relations
A relation $R$ on $S$ is said to be an equivalence if and only if $(IUR^a UR^*R)CR$. A relation $R$ on $S$ is said to be a function if and only if $R^*a R C R$. One can deduce from this definition that for a function $f$,
\[(s, s')e^f \land (s, s'')e^f \implies s'' = s',\]
i.e. any argument has a unique image.

Let $f$ be a function. The nucleus of $f$ is the relation $f^a f^a$. We have $f^a f^a = \{(s, s'): f(s) = f(s') \}$ if and only if $s$ and $s'$ have the same image by $f$.

A function $f$ on $S$ is said to be more-injective than $f'$ if and only if
\[\text{dom}(f) = \text{dom}(f'), \quad \text{and} \quad f^a f^a C f'^a f^a.\]

Intuitively speaking, $f$ is more-injective that $f'$ if and only if it discriminates more amongst its arguments; i.e. fewer arguments have a common image. At the limit, a function is said to be injective if and only if $f^a f^a C I A$.

Then each image has a single argument.

A functional on space is a function on the set of functions on space $S$.

2.5 An Introduction to Invariant Functions
We use a simple example to illustrate invariant functions and strongest invariant functions.

Space, $S = \text{natural}$;
Program, $w = \text{while } s < 108 \text{ do } s := s + 6$.

An invariant function is a function that remains invariant between any two consecutive iterations through the loop (hence between any number of iterations). Examples of invariant functions for program $w$ are:
- \[C = \{(s, s') : s' = 0\},\]
- \[f_1 = \{(s, s') : s' = s \mod 2\},\]
- \[f_2 = \{(s, s') : s' = s \mod 3\},\]
- \[f_0 = \{(s, s') : s' = s \mod 6\}.\]

Function $C$ is invariant vacuously, since it is constant.

Functions $f_1$ and $f_2$ are invariant, since the loop body adds six at a time, hence does not modify the parity of the state (mod 2) nor its remainder by three (s mod 3). Function $f_0$ is invariant, since the loop body adds six at a time, hence does not modify the remainder by six of the state.

It is not difficult to see that amongst these four functions, $f_0$ is the one that carries most information about the iterative program at hand. It is what we call a strongest invariant function of the program at hand. Other strongest invariant functions are:
- \[f_0 = \{(s, s') : s' = s \mod 6 + 2\},\]
- \[f_2 = \{(s, s') : s' = s \mod 3\},\]
- \[f_0 = \{(s, s') : s' = s \mod 6 \land 6 \leq s' < 114\}.\]

Among all these strongest invariant functions, $f_0$ is (almost) equal to the function of the loop: it coincides with the function of the loop over the domain $[0..108]$; this will be discussed further in Section 5. It is this property of strongest invariant functions that makes them most useful in the study of iterative programs.

More information on invariant functions can be found in Mili et al. (1985).

3. SEMANTICS OF ITERATION AND RECURSION
In this section we consider an iterative program $Pi$ and a recursive program $Pr$, that we choose to be equivalent, in a sense to be defined. Then we use them to discuss the relationship between invariant functions and fixpoints.

3.1 Functional Abstraction
Let $P$ be a program on space $S$. We define the functional abstraction $\phi$ of program $P$ as the function on $S$ denoted by $[P]$ and equal to:
\[\phi = \{(s, s') : \text{if program } P \text{ is executed on initial state } s \text{ then it terminates on final state } s'\}.\]

From this definition, it stems that:
\[\text{dom}(\phi) = \{s : \text{if program } P \text{ is executed on initial state } s \text{ then it terminates}\}.

3.2 Invariant Functions and Iterative Programs
Let
\[P_1 = \text{while } t \text{ do } b\]
be an iterative program on space $S$. According to a theorem due to Mills et al. (1986) the functional abstraction of $P_i$ is the (unique) function $f$ that verifies the following conditions:
1. \[\text{dom}(f) = \text{dom}(\phi)\]
2. \[I(t)' f = I(t)\]
3. \[I(t) f - I(t)' [b] f.\]

An invariant function of $P_i$ is a function $f$ on $S$ such that:
- \[\text{dom}(f) = \text{dom}(\phi)\]
- \[I(t)' f = I(t)([b] f).\]

Intuitively, $f$ is invariant if and only if for all $s$ such that $(s)$ holds, one finds the same result by applying function $f$ to $s$, or by applying to $s$ function $[b]$ followed by function $f$.

A strongest invariant function of $P_i$ is a function $f$ on $S$ such that:
- \[f\] is an invariant function of $P_i$,
- \[f\] is the most injective function $I(t)' f$.

Intuitively, a function is a strongest invariant function if and only if it is an invariant function that discriminates most amongst its arguments in $[s : t(s)]$.

Mili et al. (1985) proves that function $[P]$ is a strongest invariant function for program $P_i$.

3.3 Fixpoints and Recursive Programs
Let
\[P_r := \text{if } \neg t(s) \text{ then } s \text{ else } P_r(B(s))\]
be a recursive program on space $S$, written in an Algol-like notation. Following Broy et al. (1980) we call this format of recursion tail recursion. From this program, we derive the function $K_{Pr}$ defined by
\[K_{Pr} = \{(f, f') : f = I(t) \cup I(t)' [b] f\}.

A function $f$ on $S$ that verifies $K(f) = f$ is said to be a fixpoint of $K_{Pr}$ (or: a fixpoint of $Pr$). The smallest (with respect to inclusion) of all the fixpoints is called the least fixpoint of $K_{Pr}$ (or: a fixpoint of $Pr$).

It is well-known (Manna, 1974; Livercy, 1979) that the functional abstraction of $Pr$ is the least fixpoint of $K_{Pr}$. 

Analysis of Iterative and Recursive Programs

Example 1

S = natural.

Pr(s) := if s=0 then s else f(s).

Then KPr = {(f,f') : f' = I(s=0) U I(s#0)*f}. A fixpoint of Pr is a function f that verifies

f = I(s=0) U I(s#0)*f

<=>

f = I(s=0) U I(s#0)*f U I(s=0)

because I(s=0) = (I(s=0) U I(s=0))

<=>

f = I(s=0) U (I(s#0)UI(s=0)) * f

by distributivity

<=>

f = I(s = 0) U f

because (I(s ≠ 0) UI (s = 0)) = I

a set-theoretic identity.

Any function containing I(s=0) = {(0,0)} is a fixpoint.

The least fixpoint is, I(s=0) itself. It is left to the reader to check the validity of this result by observing the operational behaviour of program Pr.

4. INVARIANT FUNCTIONS VERSUS FIXPOINTS: MAPPINGS

We consider an iterative program Pi and an equivalent recursive program Pr, and we use them as a basis for our comparisons.

Theorem 1

Let

Pi = while t do b

and

Pr := if ~t then s else Pr(b(s))

be two programs on space S, such that [b]=B. Then [Pr]=[Pi].

Proof

We consider the theorem of Mills, which is invoked in Section 3.2, and we show that function [Pr](substituted for f) verifies all the clauses (i), (ii) and (iii).

Condition (i)

We use an informal argument here, referring to the operational semantics of iteration and recursion, to convince ourselves that Pi terminates for initial state s if and only if Pr terminates for initial state s. Hence dom([Pi])=dom([Pr]).

Condition (ii)

Because [Pr] is a fixpoint of Pr, we have

[Pr] = I(~t) U I(~t)*B*[Pr].

Whence we deduce

I(~t)*[Pr] = I(~t) U I(~t)*I(~t)*B*[Pr]

by distributivity and because

I(~t)*I(~t) = I(~t)

Condition (iii)

We deduce from equation (0) above

I(t)*[Pr] = I(t)*I(~t) U I(t)*I(t)*B*[Pr]

= I(t)*B*[Pr]

= I(t)*[b]*[Pr].

Hence [Pr]=[Pi].

Remark 1

Programs Pi and Pr are equivalent, not only from the viewpoint of having the same functional abstraction, but also from the viewpoint of having the same structure, i.e. the same condition t and the same step function B=[b]. Hence we could in principle talk of a fixpoint of the iterative program Pi or of an invariant function of the recursive program Pr. For the sake of clarity, however, we will do so only when necessary.

In order to organise our comparative effort, we pose in turn four cardinal questions.

IS AN INVARIANT FUNCTION OF Pi A FIXPOINT OF Pr?

The answer is: not necessarily.

The reader can easily convince himself/herself of this answer by considering the definitions:

An invariant function of Pi is characterised by:

dom(f) = dom([Pi])

I(t)*f = I(t)*[b]*f,

whereas

A fixpoint of Pr is characterised by:

f = I(~t) U I(t)*B*f.

In order to further illustrate our claim, we give below an example of invariant function which is not a fixpoint.

Example 2

S = integer.

Pi = while s≥0 do s:=s—12,

Pr = if s<0 then s else Pr(s—12).

Let f be {(s,s'): s' = s mod 3}. Then dom(f)=S=dom([Pi]). On the other hand,

I(t)*f = {(s,s'): s>0 & s' = s mod 3},

whereas

I(t)*[b]*f

= {(s,s'): s≥0 & s' = s mod 3}.

Hence f is an invariant function of Pi.

It is not a fixpoint of Pr since, e.g. (—5,—5) e I(~t) U I(t)*B*f while it is not in f.

IS A FIXPOINT OF Pr AN INVARIANT FUNCTION OF Pi?

The answer is:

(a) Not necessarily,

(b) We can map a fixpoint of Pr into an invariant function of Pi.

The reader can convince himself/herself of the validity of
Analysis of Iterative and Recursive Programs

clause (a) by considering, again, the definitions above. As further evidence, we give below an example of fixpoint that is not an invariant function.

Example 3
S=natural.
Pr = if s=0 then s else f(s+5)
Pi = while s≠0 do s=s+5.

Let f be
{(s,s'): s' = s mod 5}
I(s=0) U I(s≠0) * [(s,s'): s' ≠ s mod 5]
= I(s=0) U [(s,s'): s≠0 & ∃t: t=s+5 & s'=t mod 5]
= I(s=0) U [(s,s'): s≠0 & s'=s mod 5]
= [(s,s'): s=0 & s'=s mod 5] U [(s,s'): s≠0 & s'=s mod 5]
= [(s,s'): s'=s mod 5]
= f.

Hence f is a fixpoint.

Now, f is not an invariant function for Pi since its domain equals S while the domain of [Pi] is {0}.

As for clause (b), we can gain some intuition for it from the following observation. The reason why f was ruled out as an invariant function in the example above is fairly minor: its domain is too large. It seems that with proper prerestriction, we can map a fixpoint f into an invariant function. Hence the following proposition.

Proposition 1
If f is a fixpoint of Pr then
f' = I(dom([Pr])) * f
is an invariant function of Pi.

This proposition will be proven later as a corollary of proposition 3.

Remark 2
The functional which is defined by
{(f,f'): f' = I(dom(([Pr])) * f}
is denoted by H0.

IS A STRONGEST INVARIANT FUNCTION OF PI A LEAST FIXPOINT OF PR?
The answer is:
(a) Not necessarily,
(b) We can map a strongest invariant function of Pi into a (the) least fixpoint of Pr.

The reader can convince himself/herself of clause (a) by observing that there is a unique least fixpoint of Pr, namely [Pr], while there are typically several strongest invariant functions of Pi, of which [Pi] (= [Pr]) is only a special element. The example given below further illustrates this clause.

Example 4
S = natural.
Pi = while s<s<21 do s=s+5.
Pr = if s≥21 then s else Pr(s+5).

Let f be
{(s,s'): s'=s mod 5}.

According to the formulas given in Mili et al. (1985), f is a strongest invariant function for Pi. Yet, (0,0) is not an fixpoint of [Pr]. Hence f is not the least fixpoint of Pr, which is [Pr].

As for clause (b), we have the following proposition.

Proposition 2
Given a strongest invariant function f of Pi, the function
g = I(¬t) U I(t)*f & I(¬t)*I(rng(f(t)*[b]))
is a (the) least fixpoint of Pr.

Proof
The least fixpoint of Pr is [Pr]. By theorem 1, [Pr] = [Pi]. By a theorem due to Mili et al. (1985),
[Pi] = I(¬t)
U
{(s,s'): t(s) & f(s)=f(s') & ¬t(s') & ∃x: t(x) & s'=s[b](x)}.

We have
{(s,s'): t(s) & f(s)=f(s') & ¬t(s') & ∃x: t(x) & s'=s[b](x)}
= I(t)*f & I(¬t)*I(rng(I(t)*[b])).

Now,
q(s')
<=>
(∃x: t(x) & s'=s[b](x))
<=>
(∃x: (x,s')∈I(t)*[b])
<=>
s'∈rng(I(t)*[b]).

Hence I(q) can be replaced by I(rng(I(t)*[b])) in the formula above, yielding the result sought.

The Australian Computer Journal Vol. 20, No. 1, February 1988
Analysis of Iterative and Recursive Programs

Figure 1.

It is left to the reader to convince himself/herself that \( g \) is indeed the least fixpoint of \( Pr \), i.e. \([Pr]\), which is also \([Pi]\).

**Remark 3**
The functional which is defined by
\[
\{(f,f'): f' = I(\sim t) \cup I(t) \cup f \cdot I(\sim t) \cup I(rng(I(t) \circ b))\}
\]
will be denoted by \( H1 \).

**PROPOSITION 3**

Let \( f \) be the least fixpoint of \( Pr \). Then \( f \) is a strongest invariant function of \( Pi \).

**Proof**

Because \( f \) is the least fixpoint of \( Pr, f = [Pr] \). From theorem 1, \( f = [Pi] \). By Mili et al. (1985), \([Pi]\) is a strongest invariant function.

We now prove proposition 1, as a corollary of this proposition. Let \( f' \) be a fixpoint of \( Pr \). Then \([Pr] \subseteq f' \subseteq [Pi] \). Hence \( I(\text{dom}([Pr])) \subseteq f \subseteq [Pr] \).

Now, by proposition 3, \([Pr]\) is a strongest invariant function of \( Pi \).

\[
f_0 = \{(s,s'): s' = s \mod 6\},
f_1 = \{(s,s'): s' = s \mod 3\},
f_2 = \{(s,s'): s' = s \mod 2\},
\]

\[
f = H1(f_0) = I(s \geq 108) \cup \{(s,s'): s < 108 \& s \mod 6 = s' \mod 6 \& 108 \leq s' < 114\}.
\]

5. INVARIANT FUNCTIONS VERSUS FIXPOINTS: APPROACHES TO SEMANTICS

Let us recapitulate the results of the comparisons.

Relationship between invariant functions and fixpoints:
- An invariant function is not necessarily a fixpoint.
- A fixpoint is not necessarily an invariant function,

- A fixpoint can be mapped into an invariant function by means of functional \( H0 \).

Relationship between strongest invariant functions and least fixpoints:
- A strongest invariant function is not necessarily a least fixpoint.
- A strongest invariant function can be mapped into a least fixpoint by means of functional \( H1 \).
- A least fixpoint is a strongest invariant function.

For us who thought we could prove invariant function to be equivalent (modulo some mapping) to fixpoints and strongest invariant functions to be equivalent to least fixpoints, these results are rather disappointing. In fact fixpoints and invariant functions are best viewed as orthogonal approaches to the semantics of recursion and iteration, as illustrated below (see also Figure 1).

The set of fixpoints (vertical line in Figure 1) is structured in a lattice-like fashion, by the ordering relation of inclusion. For functions, being a subset of another function is equivalent to being less-defined than the other function (Manna, 1974). Any fixpoint can be mapped into the least fixpoint of \( Pr \) by means of functional \( H0 \) (see Figure 1).

We are not sure how the set of strongest invariant functions (horizontal line in Figure 1) is structured: we do know, however, that any of its elements can be mapped into \( f = [Pr] = [Pi] \) using functional \( H1 \). Also, each strongest invariant function is the top element of a lattice of invariant functions. For example, for

\[
S = \text{natural},
\]

\[
Pi = \text{while } s < 108 \text{ do } s := s + 6,
\]

we could have (see Figure 1)

\[
f_0 = \{(s,s'): s' = s \mod 6\},
\]

and

\[
f_1 = \{(s,s'): s' = s \mod 3\},
\]

\[
f_2 = \{(s,s'): s' = s \mod 2\},
\]

and

\[
f = H1(f_0) = I(s \geq 108) \cup \{(s,s'): s < 108 \& s \mod 6 = s' \mod 6 \& 108 \leq s' < 114\},
\]

which is the functional abstraction of \( Pi \).

By virtue of remark 1, the invariant function approach and the fixpoint approach can be used interchangeably on an iterative program or a recursive program. We compare these two approaches from the standpoint of practical feasibility; i.e. we assess their usefulness in deriving the functional abstraction of \( Pi \) (or \( Pr \)).

In the fixpoint approach, the least fixpoint is defined as the smallest (with respect to inclusion) function amongst the set of fixpoints. It is hardly ever feasible to use the definition as a basis for finding the least fixpoint in practice: to find the least fixpoint, one does not first define all the fixpoints, then determine their minimum. Rather, one can use the operational approach (see Manna (1974) for a recent reference), which provides that the least fixpoint is the limit (so to speak: in fact the infinite union) of the sequence of functions defined as

\[
F_0 = \phi,
\]

\[
F_{i+1} = K(F_i) \text{ for all } i \geq 0,
\]
Analysis of Iterative and Recursive Programs

where

\[ K = \{ (f, f') : f = [I^{-1}U(t)B*f'] \} \]

The strongest invariant function approach is quite radically different. It starts with building a strongest invariant function, then applying functional HI to it to find function f; the application of functional HI is a straightforward step. As for finding a strongest invariant function, it may be fairly difficult, though it is both intuitive and instructive. In Mili et al. (1985), we had given a set of systematic formulas for deriving strongest invariant functions, when the space of the program and the structure of the loop body have specific forms. These formulas have such a narrow domain of application that they can only be used in very few cases; work is under way to attempt to generalise them.

Apart from these formulas, which are totally constructive when they are applicable, one can consider a stepwise approach to the construction of least fixpoints: starting from an arbitrarily weak invariant function, carrying arbitrarily little information, and strengthening it progressively until it becomes strongest; on the graph of Figure 1, this process consists of starting from a point in a lattice of invariant functions, and working one's way up to the top of the lattice. This is an intuition-rich process, for which we have no systematic, general, approach for the moment. We will content ourselves with making a specific guideline concerning this process, based on the following remark. In our study of invariant functions, we hardly ever lend any importance to the value that an invariant function takes; rather, we lend importance to the way in which the function partitions its domain. In other words, f is not important; f*fA is. Hence the process of defining f and strengthening amount to defining an equivalence relation, then narrowing it by intersection with other equivalence relations.

6. CONCLUSION: A SUMMARY

In this paper, we have compared fixpoints, a traditional tool for analysing recursive programs, and invariant functions, a tool for analysing iterative programs. The comparison highlights two orthogonal approaches to defining the semantics of iterative programs, and of tail-recursive programs. These two approaches are orthogonal means to determining the functional abstraction of a program.

7. ACKNOWLEDGEMENT

The authors wish to acknowledge the kind assistance of Dr Janusz Laski from Oakland University, in reviewing an earlier version of this paper and commenting on it. They are also grateful to the anonymous reviewers for their insightful feedback.

8. BIBLIOGRAPHY


BIOGRAPHICAL NOTES

Fatma Mili earned the doctoral de troisieme cycle from the University of Paris in 1984. She is an assistant professor of computer science at Oakland University in Rochester, MI. Her research interests are in structured programming, programming heuristics, computer science education and decision support systems.

Mustapha Ben Alaya earned the computer engineering from the University of Tunis in June 1985. Presently he is a PhD candidate at the University of Tunis, and a consultant with the computing consulting firm SEISEG of Tunis. Mr Alaya's research interests are in structured programming and software maintenance.

Ali Mili is a PhD candidate in the history department of the University of Tunis, and an associate professor in the computer science department, which he chairs. His research interests are in structured programming, program fault tolerance and programming heuristics.

CSIRONET's Terabit File Store

B.R. Lederer†, R.J. Dakin†, K.R. Parker‡, and A.L.V. Cook‡

The Terabit File Store is a large scale storage facility based on a Braegen Automated Tape Library and disc cache. The software controlling this facility is designed to support backup and archiving of files by users operating in an environment including communicating multi-vendor mainframe hosts on a local area network. Hosts currently serviced include a Fujitsu M180 computer, on which the control program runs, two Control Data Corporation Cyber 800 series computers, and a Fujitsu M380 computer. All hosts communicate via Control Data Corporation's Loosely Coupled Network. Incoming small files up to 30M bytes in size are buffered to tape via the disc cache; large files up to 1.2 G bytes are transferred directly to tape. A special purpose catalogue records file locations on disc or tape and provides user facilities for managing large holdings of files including hierarchical subdirectories, subdirectory comments, multiple instances of the same named file distinguished by time and date of creation, keep limits to control the number of instances of a file retained, retention periods, user group control over storage allocation, and sharing of files between users and hosts.

Keywords and Phrases: Archive, file server, local area network, mass storage system, tape library.

CR Categories: C.2, C.3.

1. INTRODUCTION

CSIRONET is a publicly unlisted company owned jointly by IDAPS Australia and the Commonwealth Scientific and Industrial Research Organisation (CSIRO). CSIRONET provides bureau computing services for CSIRO, Government and other users. Mainframe hosts at the central site in Canberra include a number of Control Data Corporation (CDC) and Fujitsu computers linked by CDC's local area network called the Loosely Coupled Network (LCN); for a list of acronyms used in this paper see the Appendix). The CDC hosts include the Cyber 205 supercomputer and two Cyber 840 computers running the NOS 2 operating system. The Fujitsu computers include an M380 and M180 both running OSIV/F4-MSP, an IBM MVS-like operating system. Users Australia-wide connect with these hosts via the CSIRONET packet switched network.

The Terabit File Store (TFS) is a large scale storage facility based on a Braegen Automated Tape Library (ATL) (Section 2). The TFS is designed to provide central file archival and backup facilities for users operating in a communications environment such as the above. Currently there is a local service to the M180, the computer to which the ATL is attached and on which the TFS control program runs, and a remote service to the NOS hosts and to the M380 (Figure 1). A uniform command interface permits users of these hosts to save, retrieve, audit and delete files, and change file description parameters. Remote files are transferred on the Loosely Coupled Network by CDC's Remote Host Facility (RHF) communications software (CDC 1, CDC 2). Small incoming files, up to 30M bytes in size, are buffered to tape via a disc cache; larger files are transferred directly to tape. File locations on disc or tape are recorded in a special purpose TFS catalogue. The catalogue provides convenient facilities for managing large holdings of files including user-defined subdirectories for grouping related files, subdirectory comments, retention periods, limits on the number of files automatically kept, mechanisms for sharing files or transferring file ownership, etc.

The TFS service has been in production since 1983. Currently there are over 70,000 files in the system belonging to more than 500 users.

The software on which this service is based includes as core component, the TFS control program, a product designed and developed at CSIRONET, as well as non-CSIRONET components. The main CSIRONET contribution was to supplement the ATL with the disc cache and to manage access to the combined system by the control program for handling file movements and for recording, via the catalogue, file locations. The non-CSIRONET components include XYTEX, the Braegen supplied software which controls the ATL machine and tape database...
CSIRONET's Terabit File Store

Figure 2. The Braegen Automated Tape Library (ATL).

Table 1. TFS Approach to the ATL.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
<th>TFS Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Storage Capacity</td>
<td>2 Terabits</td>
<td>Write full tapes</td>
</tr>
<tr>
<td></td>
<td>(2000 tapes x 150 Megabytes/tape)</td>
<td>Tidy near empty tapes</td>
</tr>
<tr>
<td>High Transfer Rate</td>
<td>1 Mbyte/sec</td>
<td>Do I/O in big blocks</td>
</tr>
<tr>
<td></td>
<td>(6250 bpi tapes x 200 in/sec)</td>
<td>(19 K bytes)</td>
</tr>
<tr>
<td>Redundancy</td>
<td>4 arms and drives</td>
<td>Timeout and retry on tape mounts</td>
</tr>
<tr>
<td>Slow Access Time</td>
<td>40 sec (mount)</td>
<td>Batch writes</td>
</tr>
<tr>
<td></td>
<td>180 sec (skip)</td>
<td>Prefetch reads</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optimise file location</td>
</tr>
<tr>
<td>User Access</td>
<td>Conventionally by IBM style JCL</td>
<td>Hide tape handling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Make uniform across network</td>
</tr>
</tbody>
</table>

(Section 2), and CDC’s Remote Host Facility, the Communications software which controls file transfers across the Loosely Coupled Network (Section 4). The original design concepts underlying the TFS control program were described by Dakin, Lederer and Parker (1985). Also described in that paper was an initial service to a CDC Cyber 76 via an NSC HYPERchannel local area network. This service went into production in 1984. Subsequently the Cyber 76 was decommissioned and the TFS service converted to the Loosely Coupled Network. The product has evolved with these environmental changes and in this paper we describe its present state. In Section 2 we review design strategies. In Section 3 we describe the control program proper. In Sections 4 and 5 we describe the communications interface and user interface respectively. In Section 6 we view the product from a number of user and installation perspectives including cost, performance, reliability and security. Finally in Section 7, we place the TFS solution in the context of alternative approaches to the backup and archival problem.

2. FILESTORE APPROACH TO THE ATL

Figure 2 shows the ATL with its four attached tape drives. The ATL houses over 2000 conventional 2400 ft tapes each of 150M byte capacity. Total data storage capacity is 2.4 Terabits (2000 x 150 x 10^6 x 8 = 2.4 x 10^12 bits). The tapes are automatically picked from internal shelves by a Reel Selector robot and passed to ARM (Automatic Reel Mount) robots, one per drive, for mounting. Average time for this is about 40 seconds. The ATL comes with a vendor-supplied program called XYTEX for controlling machine movements and a database for recording tape locations.

Table 1 lists our strategies for exploiting the ATL. The disc cache plays a central role. Incoming small files are accumulated on the disc cache and retained there until insufficient space forces a tape write (Figure 3). A tape write creates a full tape of files. To reduce tape skipping at retrieval time the files are sorted so that active ones are placed near the front of the tape and so that files belonging to a particular user are grouped together on the tape. If the user subsequently requests several files from the same tape all files in the group are read back to the cache, i.e. prefetched, in a single operation. As tapes become empty due to file migration, expiration or deletion we periodically tidy near empty tapes via a process that reads back the live files to the cache and scratches the tapes.

These strategies help avoid operational problems such as tape storage underutilisation or tape drive contention that might arise were users allowed to backup their files directly to the ATL. Tape underutilisation occurs in CSIRONET’s conventional tape library where most tapes are only about 10% full, a figure near the industry average. Tape drive contention would be likely to occur were all files (and not just the large ones) transferred directly to tape via the Remote Host Facility (RHF). This is because RHF requires a separate ATL tape mount per file transfer.

3. CONTROL PROGRAM

Table 2 lists components of the control program. It consists of a number of subtasks (processes) for handling file movements into or out of the store (File Reader, File Writer), for migrating files between cache and tape (Tape Writer, Tape Reader(s)), and for performing housekeeping activities (Tape Tidy, Backup, etc.). It also includes routines for accessing the catalogue and the queue system. In this section it is convenient to restrict the description of file transfers to those that occur in the local M180 service: viz transfers of small files directly via the disc cache. The handling of remote files, including large files, will be considered in the next section. Before focussing on the sub-
Table 2. TFS Subtasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Host-disc I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td></td>
</tr>
<tr>
<td>Host Inquiry</td>
<td></td>
</tr>
<tr>
<td>File Reader</td>
<td></td>
</tr>
<tr>
<td>File Writer</td>
<td></td>
</tr>
<tr>
<td>Audit Writer</td>
<td></td>
</tr>
<tr>
<td>Tape Writer</td>
<td></td>
</tr>
<tr>
<td>Tape Reader(s)</td>
<td></td>
</tr>
<tr>
<td>Tape Tidy</td>
<td></td>
</tr>
<tr>
<td>Operator Interface</td>
<td></td>
</tr>
<tr>
<td>Backup</td>
<td></td>
</tr>
<tr>
<td>PUFF (pid/uid registration)</td>
<td></td>
</tr>
<tr>
<td>Retention Period Enforcer</td>
<td></td>
</tr>
<tr>
<td>Catalogue Control</td>
<td></td>
</tr>
<tr>
<td>Queue System</td>
<td></td>
</tr>
</tbody>
</table>

For the tasks we describe the layout of the cache, the catalogue structure, and the queue system.

The disc cache is organised as a set of buffer files. In the particular configuration shown in Figure 4 these are six OSIF/F4 direct access datasets of blocksize 19K bytes, equal to the disc drive's track size. Each buffer file contains 3900 blocks or 75M bytes, constituting half a tape load. In Figure 4, buffer files 1 to 4 have already filled up and the incoming user file is being written to successive blocks of the 'current' buffer file 5. The writing out of the buffers to tape is cyclical. Were the next incoming user file to exceed the space available on 5, a tape write would be started on 1 and 2, and the 'current' buffer would switch to the 'next' one (6 here), usually empty in advance. The direct access character of the buffer files facilitates the sorting of user files during the tape write.

User file locations on disc or tape and other file details are kept track of by the TFS catalogue. To speed up searching, the catalogue records are arranged in a logical hierarchy shown in Figure 5. The location data, including buffer file number or tape number and starting block number, is stored in the file records (circles) at the lowest levels in the hierarchy. The file records are pointed at by higher level name subdirectory records (boxes) which are generated from components of the TFS file name, e.g. INSECT.DATA generates the INSECT and DATA subdirectories (Section 5). In user initiated requests, such as file retrievals, the search for a file record begins with the user index which, for each user name, contains a pointer to the user subdirectory at the topmost level. The user subdirectory contains pointers to the host subdirectories which in turn contain pointers to the name subdirectories. In TFS system initiated activities, such as tape writes, the search begins with the tape index which contains a pointer to the file record for the last file on the tape, which in turn points to the second last file, and so on. Since the user initiated searches are time critical the following strategies are used to minimise disc I/O traffic in accessing the catalogue: the
catalogue dataset is taken to be a direct access one with large blocks; records for a particular user are allocated to one or more adjacent blocks; and buffers are employed to make it likely that blocks of current interest are retained in core.

User requests to the TFS are queued in a database on the M180 called the request file. There are separate disc queues for each type of request such as send, fetch, audit etc. The request file is polled by the Host Inquiry subtask which reads requests and, if valid queues them to in-core send, fetch and audit queues. Together with the separate queue server subtasks (File Reader, File Writer, Audit Writer) this introduces useful parallelism into the system: retrievals work off the fetch queue and need not be held up by a long send queue. The request file provides for recovery from TFS or M180 failures since a restart is sufficient to cause reprocessing of incomplete requests.

We can now describe the dynamics of the system of subtasks in Table 2. The File Reader reads an incoming file from a user disc and stores it on the cache, as indicated in Figure 4. It is started by the Host Inquiry subtask putting a user request on its queue. The File Reader asks the catalogue to reserve space on the current buffer file. If there is none it triggers the Tape Writer and switches to the next buffer file. It then copies the file block by block and, at the end, catalogues it. Outgoing files are handled as follows. The File Writer is passed a request by the Host Inquiry subtask. The File Writer interrogates the catalogue for the file's location. If the file is on the cache then the File Writer copies it to the user disc. Otherwise it queues the request to the Tape Reader and temporarily waits. A Tape Reader wakes up, issues a mount request, waits for the ATL to service the request, skips to the part of the tape containing all of the user's files, and reads one or more files back to the cache in accordance with the prefetch strategy (see Section 2).

An Automatic Backup subtask periodically copies to tape the current buffer file together with the TFS catalogue dataset and other system datasets, thereby permitting recovery in the event of disc failure. The catalogue is supported by a conventional transaction log which, in conjunction with a recent backup tape, enables complete recovery following a disc failure. Last resort recovery is provided by writing, into the first block of each user file, file headers which contain sufficient file details to enable the catalogue to be reconstructed. A utility has been developed which reads the headers from tape and logs them against an empty or backup catalogue.

Several of the subtasks in Table 2 tidy the TFS file inventory. The Retention Period Enforcer runs each night and purges files where the expiry date has been reached. Tape Tidy compacts file holdings (see Section 2). Periodically a garbage collection utility removes catalogue file records that contain no active files.

New users are entered into the TFS user index via a subtask which processes changes to CSIRONET's user registration and validation database called PUFF. The changes of interest to the TFS are those to do with allocations of space and connections between user and project names. Space allocation on the TFS is controlled by a nominated member of a group of users (Section 5), the group being defined by all those six character user names with the same first three characters.

### 4. COMMUNICATIONS INTERFACE

Figure 3 shows TFS data flows. Local and remote files are indicated, but not large files.

Local files (M180 datasets) are transferred directly between user discs and the TFS disc cache by the File Reader and File Writer (see Section 3). The interface to M180 OSIV/F4 datasets, of which there are a wide variety, is provided by a package of low level routines developed at CSIRONET called the OS Toolkit.

Remote files (NOS files, M380 datasets) are transferred via the Loosely Coupled Network using the Remote Host Facility, RHF (CDC 2). RHF provides two kinds of file transfers: permanent file transfers between disc and disc or between disc and tape; and queued file transfers between disc and input queue or output queue. Both kinds of transfer may be initiated from either source or destination host and both are exploited by the TFS. A queued file transfer is used in the TFS for queuing a user request to the M180 request file (Section 3). It is initiated by a user on a remote host issuing a TFS command (Section 5). The effect of this is to cause an M180 job file containing the command parameters as input to be submitted to the M180 as an RHF queued file. When this job runs on the M180 it executes a program which reads the command parameters and enters them on the request file. Permanent file transfers are used for transferring user files to the M180 and are initiated by the TFS control program from the M180. Incoming small user files are first transferred to intermediate M180 disc datasets by RHF before being transferred to the disc cache by the File Reader. A reverse sequence applies to outgoing files. The intermediate datasets are allocated in disc work areas (Figure 3) prior to the transfer and deleted at the completion of the transfer. Separate work areas for incoming and outgoing files help avoid undue space competition. The overall transfer rate via the work area is about 100K bytes/sec, dependent on block size.

The double handling inherent in transfers via the work area results in a performance loss which it would be desirable to avoid by communicating with the remote hosts directly. The original TFS communications interface was indeed built around a direct link, via an NSC HYPERchannel link to a CDC Cyber 76 (Section 1). A CSIRONET development provided a communications access method enabling the TFS application to perform block I/O directly between the HYPERchannel and the disc cache (Wolfendale, 1981). This path to the TFS was phased out with the decommissioning of the Cyber 76 in late 1985. Currently, in the replacement path via RHF, files have to be transferred in their entirety to M180 discs before the TFS can access them. But, as CDC have recently announced a block access method for RHF, it is possible that a direct path may again be provided.

Another way of avoiding staging via the work area is to
use RHF's capability to transfer files directly between remote host and tape. For small files this is not a desirable option since a separate tape mount is required for each file transfer. However, for large files above 30M bytes the performance gain from a transfer rate of 200K to 400K bytes/sec, dependent on block size, is very attractive. We have exploited this capability to provide support for large files up to 1.2G bytes in size. A large file appears as the second dataset on a tape, the first being the standard TFS file header.

Staging via the work area is not a problem where communications are slow, e.g. to off-site hosts and work stations on the wide area network. Transfers of this kind are handled by CSIRONET's File Transfer and Spooling system (FTS). The transfer rates are dependent on line speed and are typically of the order of 1K bytes/sec. Integration of this system with the TFS would be quite feasible since both systems have a common user interface (Section 5).

5. USER INTERFACE

Table 3 lists commands and parameters for accessing the TFS. To save a file one types NSEND, to retrieve a file NFETCH, to list file holdings NAUDIT, to delete a file NDELETE, and to change parameters NCHANGE. There are basically two kinds of parameters: file and subdirectory. Subdirectories allow for multiple instances of the same file name and for grouping of related files. Subdirectory parameters such as retention period or keep limit are stored in the subdirectory records (Section 3) and apply to all instances of a file. File parameters such as creation date and time are stored in the file records and are instance specific.

The subdirectory concept lends itself to backup and archiving applications. In backup one typically has a set of files on disc and wants to back them up on a regular basis, say weekly. At any one time there will be several instances of each file in the backup store. By means of the keep parameter one can limit the number of instances retained. In Figure 5, for example, there are three instances of the file INSECT.DATA. Had the keep limit been set to three then the effect of backing up a new instance would be to cause the oldest one to be automatically deleted. This feature is similar in spirit to IBM's generation dataset group. By contrast there are some new features aimed at facilitating the management of large holdings of files. One is the ability to nest subdirectories and so impose a hierarchical structure on file holdings. Another is the subdirectory comment, up to 45 characters of text, which provides a means of recognising archived files when they are subsequently audited.

The subdirectory parameters such as keep limit or comment are normally specified on the command line when the first instance of a file is saved, but may be changed at any time via NCHANGE. The period separator in the file name is the means whereby the user defines his subdirectory structure. Thus the subdirectories INSECT and DATA in Figure 5 would have been automatically created the first time a file with the TFS name INSECT-.DATA was saved. Subsequent saves simply add new file records.

To retrieve one of a number of instances of a file it is necessary to specify the time and date of creation otherwise the latest instance of a file is returned. Host specific parameters, stored in the TFS file header (Section 3) at save time, are automatically used to recreate the file in its original format at retrieval time. On OSIV/F4 hosts these include the record and block size, dataset type, space allocation and disc volume location. On NOS they include the permanent file type and permit category. Some of the file header defaults such as volume location may be overridden by explicit settings on the command line.

To permit other users to access a particular subdirectory one could either grant them the appropriate read access or give them the file password, if any. The default read access restricts access to the owner only, but there are settings that allow users with the same project or group name (Section 3), or else all users, to access a file. If a user wishes to change his project or user name then there are facilities enabling him to assign ownership of his files to another project or user name.

The sharing of files between hosts is currently limited to

Table 3. User Interface.

<table>
<thead>
<tr>
<th>Commands</th>
<th>File Parameters</th>
<th>Identification</th>
<th>Space Management</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsend</td>
<td>Local file name</td>
<td>Remote file name</td>
<td>Retention period</td>
<td>Access for read</td>
</tr>
<tr>
<td></td>
<td>Cycle</td>
<td>Comment</td>
<td>Keep limit</td>
<td>Password</td>
</tr>
<tr>
<td>nfetch</td>
<td>Local file name</td>
<td>Remote file name</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td></td>
<td>Cycle</td>
<td></td>
<td></td>
<td>Password</td>
</tr>
<tr>
<td></td>
<td>Creation date, time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ndelete</td>
<td>Cycle</td>
<td>Remote file name</td>
<td></td>
<td>Password</td>
</tr>
<tr>
<td></td>
<td>Date range</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>naudit</td>
<td>Local file name</td>
<td>Remote file name</td>
<td>Retention period</td>
<td>Access for read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Comment</td>
<td>Keep limit</td>
<td>Password</td>
</tr>
<tr>
<td>nchange</td>
<td>Local file name</td>
<td>Remote file name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Comment</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
hosts of the same kind — the two NOS hosts can share files and M380 users can retrieve M180 files (but not vice-versa). Differences in file structure will probably restrict sharing of files between dissimilar hosts to simple text files. For example, NOS files comprise a number of 64 word physical record units (PRUs) with imbedded short PRUs indicating record and file boundaries. To archive such a file to the M180 it has been necessary to imbed in the file control words describing the PRU structure so as to ensure that the original structure can be recreated at retrieval time. The control word processing has been implemented by modifications to RHF and does not affect performance. However the presence of control words in TFS files saved from NOS hosts renders them unfit for subsequent retrieval to non-NOS hosts.

TFS transfers are asynchronous: command execution simply queues a request to the M180 request file with control being returned to the user. To coordinate a file transfer with subsequent processing a facility exists whereby a user can nominate a file containing a job to be submitted upon completion of the transfer. Typical applications are to retrieve a file and spin off a job to process it or to free a lock on a database upon successful completion of a backup. The asynchronous nature of the transfers also shows up in the handling of user messages. On the M180 (and M380) these are written back to the user terminal upon completion of the transfer. On remote hosts they are stored in a history file of messages.

A number of facilities enable files to be managed by a group of users (see Section 3). A nominated Group Control Officer can audit all files in a group by specifying the group identifier on the audit. If a user is leaving the group then the Group Control Officer can delete any remaining files. To avoid overcommitment of TFS storage the groups are assigned a TFS space allocation whose size in TFS 19K byte blocks is controlled by the Group Control Officer.

6. PERSPECTIVES
In this section we view the TFS service from a number of perspectives including cost, usage, performance, security, reliability and adaptability.

The TFS provides an interesting alternative to conventional disc or tape storage, both in terms of cost and functionality. TFS charges include a storage charge of 60 cents per M byte per month and an access charge of $1.74 per M byte. The storage charge per M byte is about 40 times cheaper than CSIRONET public disc, about ten times cheaper than private pack, and about ten times dearer than magnetic tape. The access charge reduces by a factor of four for large files above 30 M byte and by an additional factor of three for large files saved outside of prime time (8 am to 5 pm weekdays). Applications where cost or functionality considerations favour the TFS over the other media include the backup or archival of numbers of small files and the backup of large databases. TFS users with small files would usually achieve a much better space utilisation, and so a lower overall cost, than the 10% typical of tape users (Section 2); this is because TFS users allocate space by block rather than by volume as with tape or private pack. At the same time the TFS users are better placed to exploit the reduced job processing rates that apply in non-prime time, when CSIRONET runs the machines in unattended mode. Database owners are attracted to the TFS since it enables them to backup their database in unattended mode and so avoid interrupting their database users.

The TFS went into production in May 1983. Early growth in usage is indicated by the following figures:

<table>
<thead>
<tr>
<th></th>
<th>July 1984</th>
<th>Nov 1984</th>
<th>April 1985</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users with files</td>
<td>199</td>
<td>375</td>
<td>536</td>
</tr>
<tr>
<td>Total files</td>
<td>10,251</td>
<td>23,832</td>
<td>43,212</td>
</tr>
</tbody>
</table>

File holdings currently exceed 70,000 files belonging to over 500 users. Storage totals 40 gigabytes or about 20% of effective ATL capacity. Daily traffic, measured over a three week period in 1986 averaged about 250 M bytes:

- Files MB
- Saved/day 284 132
- Retrieved/day 148 106

During this period there were about 1 to 2 tape writes per day and about 30 tape reads per day. This indicates the effectiveness of the disc cache in reducing the frequency of ATL access.

Response time is good. It takes about 1 to 10 minutes to retrieve a file depending on location, size and length of queue. Up to four Tape Readers, one per tape drive, simultaneously serve the retrieval queue. Retrievals proceed in parallel with other activities such as sends or audits. All activities have the priority of system tasks.

Resource utilisation is quite tolerable. The Fujitsu M180 II-AD is a 2.7 MIP machine with 12 M bytes of memory. Working set and CPU requirements for the TFS when active are about 1.5 M byte and 5-10% respectively; for RHF they are about .5 M byte and 5-10%. Whilst idling the TFS takes up less than 1% of the CPU. Peak CPU usage occurs at night when housekeeping activities such as accounting or catalogue validity check occur; during prime time the CPU requirements are currently about 10 to 20 minutes dependent on traffic.

Security of file holdings is a major consideration. One vital component here is the automatic backup scheme (Section 3). There have been four head crashes on the disc cache drives in about as many years. In two of these failures the TFS catalogue disc dataset was corrupted. It was successfully restored from backup tape without loss of files. In one of the failures a disc buffer file was corrupted. It was restored from backup tape with the loss of only a few files. Another vital component is the check against tape overwriting. After a tape is open but before writing commences the volume serial number is available. At this point we dump the OSIV/F4 system catalogue containing all volume serial numbers of all TFS permanent tapes and search it for the volume serial number in question; if found we alert the operator to a potential case of overwriting. Usually exceptions of this kind are detected in advance by an overnight check on the consistency of the OSIV, XYTEX (Section 2), and TFS catalogues.
Large files are more secure than small ones in two senses. Large files are written directly to tape and do not depend on the flushing of the disc cache buffer files. They appear as a standard OSIV/F4 tape dataset (see Section 4) and are independently recorded in a UCC-1 Tape Management System (TMS) catalogue. This makes it possible for users to audit their large files independently of the TFS and retrieve them directly via RHF.

Reliability of operation, especially during unattended mode, is another major consideration. We experienced a lot of early problems with tape mount failures, particularly with the automatic threading of tapes. One device for getting around this problem was the introduction of a timeout and retry on the tape mount: in the case of a write a new scratch tape would be mounted on a new drive. However recovery in the case of a read usually requires manual intervention. Recovery in the case of M180 failure is easily handled: the user requests remain on the request file and are simply reprocessed following a restart (see Section 3).

We also experienced a lot of early problems with communications hardware and software (some of which was described in Dakin et al., 1985), with the XYTEX software which had to be converted to run under OSIV/F4, with the ATL which went through a move, an expansion, and a settling in period, and with bugs in our own software. These problems, which have now all been overcome, delayed the project: although the project team was formed in early 1980 a production release did not occur until mid-1983.

One practical point illustrated by this project is the need to build in flexibility. Archived files, by definition, tend to stay in the system for long periods of time. Meanwhile the environment changes. Replacement of the HYPERchannel link by the Loosely Coupled Network, associated with the decommissioning of the Cyber 76 (Section 2), was a case in point. Differences in the communications software rendered existing Cyber 76 and NOS files totally incompatible with the new link. However, it was found possible to resolve these incompatibilities by introducing conversions that are automatically performed by the TFS at retrieval time. The conversions were of a similar kind to those independently required to migrate conventional Cyber 76 tapes to NOS. Factors contributing to the adaptability of the TFS software are the fact that most of it is written in a high level language, PL/1, and that many of the internal control blocks contain room for expansion to include new parameters.

7. CONCLUSION
The TFS represents a solution to the problem of how to provide user driven backup and archival facilities in a multi-vendor bureau environment such as exists at CSIRONET. It avoids installation problems such as tape underutilisation inherent in conventional approaches. It overcomes user problems such as lack of tape access out of hours. It is cheap enough to attract users whilst still satisfying commercial requirements such as cost recovery or even profitability. Its cheapness derives from the project having achieved one of its design goals: viz. to pass on to users the economies of scale that result from providing access to a common mass storage system on a local area network.

Recent hardware developments such as cartridge tapes or optical discs do not, we believe, obsolete the TFS solution. Cartridge tapes of the IBM 3480 variety in combination with autoloading devices offer improvements over conventional tape backup both in terms of speed, availability and data compression. However these devices are oriented to volume dumping and do not address the problem of user driven file backup and retrieval free from operator intervention. Optical discs are suited to some kinds of archival applications such as the storage of image data. However their error rate is currently such as to make them generally unfit for conventional digital applications. StorageTek, for example, have reported the abandonment of a project to build a 4 G byte laser disc because the media manufacturer was unable to deliver a sufficiently low error rate to satisfy the requirements of clients, particularly in the banking sector.

Continuing interest in mass storage technology, such as that described in this paper, is evidenced by commercial developments, both at the hardware and software level. StorageTek have recently announced their Automated Cartridge Store, based on standard IBM 3480 cartridge tapes. This device represents the latest generation of jukebox technology in the tradition of the ATL. At the same time industry acceptance of the file server approach is indicated by the take-up of the following software products. In the pure IBM-compatible environment there is Hierarchical Storage Manager (HSM) and DASD Management System (DMS/OS). In the multi-vendor environment there is MASSTOR. All of these products support explicit user driven facilities, such as the TFS provides. In addition HSM and DMS/OS support implicit facilities including automatic archiving of files initiated by the installation and automatic recall upon reference by the user. MASSTOR is very similar in functionality to the TFS but is based on IBM 3850 wide cartridge technology and on the NSC HYPERchannel (Ewing and Peskin, 1982). Miller and Collins (1985) have approached the formulation of a reference model for such systems and given a number of examples.

ACKNOWLEDGEMENTS
This research formed part of the Facom-CSIRO Joint Development Project and was supported in part by Storage Technology of Australia. The project was launched under Dr. P. Claringbold, former Chief General Manager of CSIRONET, and it is a pleasure to acknowledge his support. We would also like to acknowledge the following contributions. D.R. Cameron and S. Brandstetter have, at one time or another, been part of the design and development team. B.J. Austin, G.L. Wolfendale, P.P. Hanlon and G. Havas developed the communications software for the original HYPERchannel local network. Subsequently B.J. Austin developed an archiving mode of transfer over the Loosely Coupled Network. W.S. Ford, H. Kadetz, S. Yong, A.C. Edington and G. Tonzing developed software for the
CSIRONET’s Terabit File Store

FTS system which we were able to use for the OSIV/F4 interface to TFS. H. Kadetz developed the OS Toolkit. P.P. Hanlon and H. Kadetz have provided invaluable assistance on OSIV/F4, E.H. Kinney and B.P. McDowall on NOS. J.M. Hudson developed the backup facilities on NOS. E. Barry provided invaluable operational assistance.

REFERENCES


BIOGRAPHICAL NOTES

Brian Lederer is currently a Senior Research Scientist at CSIRONET Canberra responsible for managing the Information Storage System Section. Before joining CSIRONET in 1979 he worked as a programmer for Caltex Oil (Australia) and prior to that as a Professional Officer in the School of Mechanical and Industrial Engineering, University of New South Wales. He has a BSc (Hons I) in Applied Mathematics from the University of New South Wales, a PhD in Theoretical Physics from the University of Sydney (1975) and a MEngSc in Operations Research from the University of New South Wales (1980).

Robert Dakin is currently Assistant General Manager (Products) at CSIRONET in Canberra. He previously was a Principal Scientific Officer at the UK Atomic Energy Authority’s Culham Laboratory, a Senior Lecturer in Mathematics at the University of Papua New Guinea, and founded the Operations Research Section at Cadbury’s in Claremont, Tasmania. He has a BSc degree in Physics, a first class honours BE and MEngSc in Electrical Engineering and PhD in Computer Science from Sydney University.

Kenneth R. Parker received his BSc (Hons) in Mathematics and Physics from the University of Melbourne in 1975 and a PhD in Theoretical Elementary Particle Physics from the University of Oxford in 1979. He researched elementary particle physics at CERN as a University of Surrey Research Fellow and then Information Storage Systems as a Research Scientist at the CSIRO Division of Computing Research. Currently at the CSIRO Division of Information Technology, his research interests include Protocol Engineering and the specification, verification and testing of upper-layer OSI communications protocols.

Anton Cook is currently a CSIRO Experimental Scientist seconded to Network Research Pty Ltd, Canberra. He joined the CSIRO Division of Computing Research Section in 1966, was Officer-in-Charge of the Melbourne and Canberra branches between 1966 and 1977, and was Controller (Operations) at CSIRONET Canberra from 1977 to 1984. He received a BSc degree majoring in Mathematics from the University of Melbourne.

APPENDIX

Here is a list of acronyms, meanings and section numbers, in brackets, in which the acronyms either are introduced or defined.

ARM Automatic Reel Mount (2)
ATL Automated Tape Library (2)
CDC Control Data Corporation (1)
CSIRO Commonwealth Scientific and Industrial Research Organisation (1)
DASD Direct Access Storage Device; ie disc (7)
DMS/OS DASD Management System; IBM environment archiving software (7)
Dataset IBM terminology for a file
FTS File Transfer and Spooling System; CSIRONET software (4)
HSM Hierarchical Storage Manager; IBM environment archiving software (7)
HYPERchannel NSC local area network (1,4,6)
LCN Loosely Coupled Network; local area network (1, 4)
MVS Multiple Virtual System; IBM operating system (1)
M180, M380 Fujitsu M Series mainframe hosts
NOS Network Systems Corporation (1)
NAUDIT TFS command to audit files (5)
NCHANGE TFS command to change file parameters (5)
NDDELETE TFS command to delete files (5)
NFETCH TFS command to retrieve a file (5)
NOS Network Operating System;
CDC software (1)
NSC Network Systems Corporation (1)
NSEND TFS command to save a file (5)
OSIV/F4 Operating System; Fujitsu software similar to MVS (1)
PRU Physical Record Unit; NOS concept (5)
PUFF Project/User File; CSIRONET software (3)
RHF Remote Host Facility;
CDC communications software (1, 4)
Terabit 10^12 bits (2)
TFS Terabit File Store; CSIRONET software (1)
TMS Tape Management System; UCC-1 software (6)
VSOS Virtual Storage Operating System;
CDC software (1)
XYTEX Vendor software supplied with the
ATL (2)
The Preface notes that this book is intended for 'second or final year students of computing science or related disciplines who have acquired an appreciation of traditional file processing techniques'. There is no comment on whether the book aims to provide a general background for such readers or whether it is to furnish an adequate introduction for those who will work extensively with data bases. Bowers succeeds admirably in the former role and the book can be strongly recommended; in the latter case, the treatment will not be found to have adequate depth, though here too it can be recommended for the overview it provides.

There is an introductory chapter that looks generally at file processing and data bases and which mentions features of the ANSI/SPARC architecture. The next two chapters are about data and its analysis and too it can be recommended for the overview it provides.

The data base area has much jargon and a number of subtle issues. The treatment provided by Bowers introduces both. The type of presentation is very readable and his book can be strongly recommended.

As is appropriate for an introductory book, the discussion is based on examples. Mention is made of theoretical bases and representations but we do not see them. The expression flows well and related examples are used for different parts of this tapestry-type presentation. The author succeeds in integrating the data analysis, normalisation and synthesis topics.

The data base area has much jargon and a number of subtle issues. The treatment provided by Bowers introduces both. The type of presentation is very readable and his book can be strongly recommended.

Attempts are made throughout the book to emphasise a systematic approach to the topics. Thus we have in Chapter 6, for example, information under the following topic headings:

6.10 Testing Strategy — including seven important guidelines

John Hiller
University of New South Wales

The aim of this book is to present programming as a discipline beginning with a specification, and then transforming it into a program. The authors' material evolved from first-year undergraduate courses and short industrial courses. As such, it is in good tutorial style, and is — for a formal treatment — easy to read. The authors avoid overwhelming the non-mathematical reader with unfamiliar symbols. They first introduce an operator with a name such as 'IS_DEFINED_AS', and after a chapter or two, they use a mathematical symbol for it instead. This works quite well.

Using specifications to derive programs is certainly an advance on not using specifications. However, it would be wrong to suppose that current specification technology can be immediately applied to everyday problems in industry. The problem that bedevils all specification methods is the same — one must have a soundly based language in which to express specifications. Such specification languages are similar to programming languages in that they start with a few basic concepts that can be tailored to deal with specific problem domains. For example, the specification language 'Z' is based on sets and maps. The user of such a language is therefore faced with expressing every problem in terms of sets, maps, or whatever, which may not be obvious or easy. The problem is analogous to the traditional problem of systems developers, who do not have the inclination to read this topic in depth. Tutorial questions are not included, although a small bibliography is described.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

Watts does make some good points in the text. In Chapter 6 for example, he observes that cost/effective use of an application generator may require significant changes in procedures for example, prototypes may be used more extensively, less manual documentation may be necessary and additional user involvement may be feasible. However I found difficulty in accepting some of his other assertions, such as (in Chapter 3)

(i) total project elapsed time will be at least halved, or
(ii) coding and testing will be reduced in the ratio 1:7.

The book does not include a case study, or a user survey (although general expressions of user experiences are discussed). I could not decide from this book whether there were any circumstances in which an application generator would be preferable to a full fourth-generation language.

SIMULA has an interesting and ongoing history — not only because of its belated recognition as an important theoretical language, but also because of its steadily increasing popularity. For devotees, a problem has forever been the unavailability of a high quality text dealing with the unique features of the language. A responsibility rests upon authors of SIMULA texts (or any language with unique features) to address the methodological and philosophical implications as well as the breadth and butter syntactic and semantic issues. Perhaps this language more than any other has suffered in its acceptance as a general purpose language because of the lack of a comprehensive and integrated exposition of the features it provides!

This text covers all aspects of SIMULA as specified in the 1986 SIMULA Standard. The method adopted by the author is to introduce a feature of the language and to demonstrate its applicability by means of small example programs. Nuances of semantics are well presented. The syntax of the language is not indicated and the reader is expected to glean this from the sample programs. It is clear that recent developments in the language have increased its self-documentation capability. The book is written in a clear, comprehensive style. Most of the examples are complete programs. Exercises are mainly based on these and the reader is encouraged to learn by modifying a given program. Clear guidance is given, but no solutions are presented in the text. SIMULA is a general purpose language; the specific application of SIMULA to discrete event simulation problems is not emphasised.

It is unfortunate that computer science educators have been slow to grasp SIMULA's potential: it has facilities for data abstraction and modularity; its powerful coroutine feature, prefixed block, hierarchial abstraction, separate compilation capability, provision of excellent compilers and debugging aids, all make it a unique and highly expressive language useful for demonstrating most modern approaches to software development. With such enormous potential, authors have an obligation to convey the necessary programming design and structuring skills so that the specialised features of the language are natural to apply. Persistent illustration through good problem selection and the application of modern problem solving methodologies are lacking in this text. The presentation of a plethora of programming delights or language features in isolation does not indicate how to integrate these jewels into a problem solution. Undoubtedly this consideration would considerably lengthen the book.

Nonetheless, at just over $50 for a soft cover, I would recommend this as a text for SIMULA users. It certainly beats hunting through a line printer manual. The explanations and examples are just what any programmer needs when clarifying some hazy point. Perhaps the text would also be of use to those who have met similar (sorry) features (like coroutines, class hierarchies, prefixed blocks, ...) in other languages. However, this book could not be considered seriously as a text for a novice or for an introductory programming course.

E. Saltman

University of Queensland


As the title suggests this is a collection of articles from the pages of the famous *Dr Dobb's Journal of Software Tools (DDJ)* around the common theme of programming the Motorola 68000 microprocessor family. (The programmers themselves are referred to as 'sixers' as opposed to the 'eighters' who have to suffer the Intel 80XX microprocessor series.)

Fifteen different authors have contributed the nineteen chapters of the book and the variation in style is as much as could be expected in the circumstances. Some new material and a reworking of several chapters has improved the flow of the book as far as the reader is concerned.

The book is structured into three sections, the first of which is an excellent introduction by Daniel Appleman to the hardware design philosophy, instruction set and machine architecture of the 68000. I thoroughly recommend these three chapters to anyone who wishes to gain an overview of the 68000 family. It is the best such description I have seen. The age of the book dictates that discussion stops with the 68020 and so the newer members of the family such as the 68303 are not mentioned.

In typical DDJ style the book is aimed squarely at those whose main interest is at the hardware board level but who need some practical software to drive the boards. This is summed up by Steve Passe in chapter 13: 'Being a "hardware" type I naturally had to have a 16 bit as soon as I could manage it. After letting the dust settle a bit I decided to go with the 68000 processor from Motorola. The board I chose has Motorola's MACSBug monitor in PROM, but other than that I had no software to run my new friend.'

The book aims to fulfil that need for software in the second section. Over half of the pages in the book are program listings in classic DDJ fashion. Some of the chapter titles give an indication of the fare on offer:

- Tiny BASIC
  - Comfort: a faster Forth
  - A 68000 native-code cross-compiler
  - A 68000 assembler
  - A 68000 cross-assembler
  - All the source texts are available in machine-readable form in a variety of disk formats (Amiga, Atari 520ST, CP/M, Mac, MS-DOS and Osborne) from the publishers. Most are in assembly language but there is a smattering of Forth too. The 68000 cross-assembler is written in Modula 2 — a binary version is also obtainable.

The final section is entitled 'Useful 68000 routines and techniques' and is an ad hoc mixture of assembler programs with a few gems including two multitasking kernels, a random number generator and an ingenious memory test which 'worms' its way through the main memory. Mac users with access to the MDS assembly language development system are given code which produces screen images of the Mandelbrot set.

For those interested in the 68000 at the board level prepared to put in some development effort this book is full of useful software and coding samples in assembly language. Educators, too, will find plenty of examples and exercises for their courses ranging from simple code sequences to working operating systems. Even serious software developers may find some use for the book as an insight into the philosophy of the 68000 processor family.

Michael J. Rees
University of Tasmania


This book addresses a part of the computing science curriculum which has so far received less than sufficient attention. Perhaps the word 'discrete' could be omitted from the title because there are concepts also from continuous mathematics such as limits and asymptotic complexity.

The presentation of the book is of a high standard and it is apparent that much attention has been paid to graphical design and layout. The review copy was bound in hard covers and one would expect the price to be fairly high. On the other hand, since the book is aimed at students in the first two undergraduate years, relatively large print runs should help to keep the price down.

The authors state in the preface that the purpose of the book is 'to introduce discrete mathematics and its computer science applications to students of computer science so that they will have an adequate set of mathematical tools for problem solving in their advanced courses'. Indeed, the list of chapter headings looks impressive. The strength of the book lies in the fact that it introduces the reader to a great variety of mathematical topics connected with computer science and it is therefore useful for students of computer science as a reference which also permits self study. Each chapter is followed by an application/example of the material covered. There are many exercises which make the book suitable for self study.

The main weakness is that only a superficial introduction is given to each of the topics in the book. Apparently, this has been done in the

Book Reviews

interest of readability, but it has the effect of obscuring the power and beauty of the subject. The reader remains unaware of many important issues such as the connection between logic and data bases or the use of many-sorted algebras in data abstraction. The treatment is altogether a little too simplistic and therefore occasionally boring. The application examples are sometimes trivial and sometimes misleading. For example, the application at the end of the third chapter has the title 'data base management system' but it describes on two and a half pages what is meant by selection, projection and join operations.

Clearly, this book must be considered a first introduction to each of the topics covered and as such it will be useful.

F. Hille
The University of Wollongong


This beautifully published book presents the theory of designs, a rapidly developing and important part of discrete mathematics. The work is a very comprehensive monograph and includes much material that has only previously appeared in research papers.

In the classical Euclidean Geometry of the plane two points determine a line and two lines are either parallel or meet in a unique point. The Euclidean plane can be completed to "The Projective Plane" by adding a line at infinity where parallel lines meet. A question looked at over the past fifty years is whether finite projective planes exist, in which two points determine a unique line and two lines always determine a unique point. These yield examples of designs. Another area which motivated the study of designs was the need to devise sound methods of performing agricultural experiments which did not involve trying every treatment on each variety. This led to the development of Balanced Incomplete Block Designs.

The theoretical development involves the study of Incidence Structures which include the finite projective planes, balanced block designs as well as solutions to elementary but interesting problems such as Kirkman's school girl problem and the problem of the thirty six officers. The book emphasises actual constructions and all sections of each chapter include suitable examples and exercises as well as references to the very extensive bibliography. The material presented is not however just a collection of research papers --- in many situations the authors have reproved the results and provided simpler or more illuminating approaches. The authors' impressive scholarship is evident throughout the book.

The authors claim that the book will provide some of the necessary material for anyone working in Communications Engineering, Optimisation, Statistical Planning, Computer Science, and Signal Processing. Indeed, on the fly leaf it is claimed that applications in these areas are mentioned. This is not true --- indeed, the book is striking for its lack of connection even to other parts of Combinatorial Mathematics. Nonetheless I must agree with Zentralblatt fur Mathematik in that, "This book is an excellent and up-to-date treatise... It will be a highly useful reference book."

David C. Hunt
University of New South Wales


The book is an introduction to logic and its applications for undergraduates in Computer Science. It covers prepositional calculus using truth tables, combinatorial logic and digital circuit design, predicate calculus and reasoning about programs, proof theory, models, clausal form, resolution theorem proving, and logic programming.

The strong points of the book are its readability and relevance. The book is rigorous, complete and concise. Instead of the usual dry abstract approach to logic, the authors use poignant examples from computer science to explain the concepts of formal logic. They also stress the relevance of the concepts and theorems to applications in computer science.

Unfortunately, the book does not suffice for the usual courses on Digital Logic, Program Verification, Automatic Theorem Proving, or Logic Programming offered by Australian Computer Science departments. While it is a very good introduction to each of these areas, additional material would be required in the courses. This breadth also leads to another criticism. We would have preferred more examples and exercises (particularly self-study exercises with answers) to reinforce comprehension of the concepts and applications. The number of examples and exercises is adequate but not abundant.

The book is highly recommended to students, engineers, programmers, analysts and even managers, as an introduction to the applications of formal logic in computer science. It is by far the most readable way for these people to learn formal logic.

Greg Butler and Judy Kay
University of Sydney


This book would be useful mainly for managers who are first time or casual users operating in a Pick environment, or for analyst programmers needing an overview of the system. It can serve also as a non-technical text, to introduce the principles of operating systems using a top-down approach.

Pick unlike other operating systems is totally integrated, having its own DBMS called ACCESS and its own version of the Basic language. Accordingly the system comprises a broader range of elements than are associated with other operating systems.

Topics common to all operating systems such as file design, are well presented at the introductory level. As the author states 'Data investigation is as important in Pick as in any other operating system'. The book gives a good example of database normalisation, showing the first, second and third normal forms. Brief descriptions are given of reference and BOMP files (BOMP = Bill of Materials Processing). ACCESS is fully covered with a variety of examples to illustrate its features. Similarly, the use of Pick Basic is well described, with full program listings.

The index lacks thoroughness, e.g. Reference to Q — pointers occurs from page 39 onwards, yet page 79 is the first mentioned in the index. However an extensive Pick glossary is provided and some of the chapters are summarised.

Pick, like Pascal, was designed by one person and consequently has idiosyncrasies, which the book does not identify. Pick provides the database utility ACCESS but does not provide facilities for users to create their own simple databases. An item on a Pick file is 32k and a 900-1000 line Basic program is bordering on this limit. Thus to save space one has to reduce documentation or indentation.

The author points out in the introduction that Pick runs on IBM systems and McDonnell Douglas/Information Systems, but fails to mention that it does not run on the world's second largest computer vendor machine, Digital/VAX. This could be just as well since the latest literature indicates that third parties will supply Pick for the VAX.

Pick and Unix both have an interesting past and a speculative future, so a chapter on the two would not have been out of place. Some of the shortcomings of Pick could have been mentioned here that are now being addressed such as networking, word processing and the lack of security features which have hampered its penetration into the mainframe market.

The book is well written, in user friendly style, easier to read than the manual and recommended for the first time user.

Peter Radovici
Glebe, NSW

HERSKOVITS, A. (1986): 
*Language and Spatial Cognition*, Cambridge University Press, x+208 pp., $84.00 (hardcover).

A what is truth?* asked Pilate. He is most unlikely to have had any inkling of the linguistic significance of his flippant but insightful remark. So many of our truthful statements are not logically true but are mediated by factors such as metaphor, extension and shared knowledge. Consider *there is an idea in the air*, *there is a man in the armchair*, *there is no water in the refrigerator*. None of these statements is ever likely to be literally true. Suppose a loaf of bread is enclosed under an inverted bowl. It is misleading to say either 'the bread is in the bowl' or 'the bread is not in the water'. None of these statements is ever likely to be literally true. Suppose a loaf of bread is enclosed under an inverted bowl. Unfortunately insufficient technical detail is provided to make this book a reference text and the serious reader would be obliged to follow up the references provided at the end of each paper for a detailed understanding of the contents. The emphasis on the computer diagnosis of ECG signals is also to some measure unfortunate as large-scale implementation of such systems have been notably few. The more the less, a wealth of statistical detail on the variability and thus the predictive value of different parameters in the ECG makes this a very valuable text for those interested in extracting the maximum diagnostic information from the ECG using Inductive Inference methods or other techniques in Artificial Intelligence.

Since the publication of these Conference Proceedings a great deal of progress has been made in the hardware and software used for recording, filtering and interpreting of the surface ECG and real time measurements of ST-segment depression, QT intervals and a host of other parameters is now almost routine. These methods and those using the Frank Lead Vector Cardiograph represent exciting new opportunities for non-invasive diagnosis of silent ischaemia and other preclinical manifestations of coronary artery disease. This book, though expensive at $US57, is a useful introduction to an exciting and rapidly developing interdisciplinary area of study.

Branko G. Celler
University of New South Wales

**Book Reviews**


This book presents the proceedings of the 4th Conference of the IFIP-IMIA series on Computer ECG processing held in Leuven, Belgium in June of 1985. The objectives of the Conference were to present state of the art developments in the use of computers in electrocardiography and to review developments in the field since the previous conference held in 1979. Special emphasis was given in this conference on efforts to develop international standards for computer based methods of ECG analysis, and as a result the European Common Market ECG Standards in Electrocardiography (CSE) project is heavily represented. The focus of the Conference was on the resting ECG and 47 papers were presented by acknowledged world experts on signal acquisition and measurement, diagnostic classifications of ECG's, body surface mapping, signal averaging and ambulatory ECG monitoring.

More than 25 years have elapsed since H.V. Piperberger and his coworkers began using computers for ECG processing. For readers unfamiliar with the field, this book provides an excellent introduction to state of the art techniques for recording, processing and analysis of ECG signals by computer. Unfortunately insufficient technical detail is provided to make this book a reference text and the serious reader would be obliged to follow up the references provided at the end of each paper for a detailed understanding of the contents. The emphasis on the computer diagnosis of ECG signals is also to some measure unfortunate as large-scale implementation of such systems have been notably few. The book makes this a very valuable text for those interested in extracting the maximum diagnostic information from the ECG using Inductive Inference methods or other techniques in Artificial Intelligence.

The papers range over a variety of aspects of the use of CAL. Some describe empirical research studies or curriculum and courseware development; others are analytical or reflective; a few deal with software and implementation strategies; yet others propose a research agenda. Not surprisingly, more research questions are posed than are answered. A wide spectrum of forms of CAL is discussed, including simulation and games, tutorial systems, data bases, videodata, expert systems and PROLOG.

With the benefit of several years of substantial funding of curriculum and courseware development through the Microelectronics Education Programme until 1986, the British contributions show that thinking has moved on from the earlier naive absorption with purely technical detail which characterised so much early work in computer education in many countries, including Australia. It is refreshing to see that half the papers may be described as concerned with educational questions about CAL, rather than with matters of software design or specific subject content. Indeed the dichotomy often found in computer education between the educationalists and the computer enthusiasts seems to be breaking down. (One of the authors expresses scepticism about the scope for CAL in the teaching of history.)

In particular, several papers deal with questions of process and the relation between CAL and classroom teaching-learning styles. Several papers discuss social processes, such as those of peer interaction, in the use of CAL, and approaches such as enquiry or discovery learning. Whereas too many early CAL enthusiasts have failed to consider questions of purpose and process, of tentatively assuming and promoting a practical and computational proposals might result from this consideration. The author clearly recognises this, but her attempt to formalise the encoding and decoding process in Part II (chapter seven) is forced and unsuccessful.

Part III brings us to case studies of the topological prepositions ('to', 'on' and 'in') in chapter nine and the projective prepositions ('in front of', 'behind', 'to the left of', etc.) in chapter ten. This is where the volume comes into its own. The presentation in chapter nine is excellent and comprehensive, and although the approach may not conform completely to expectations, it does come up with reasonable explanations of most of the distinctions. Chapter ten seems less solid, but contains a great deal of useful analysis which will take quite a while to assimilate.

For the newcomer to the subject, this monograph provides something of an introduction, but it is really necessary to refer to the classic papers which are referenced throughout. The research presented in chapters nine and ten is most certainly worth dissemination, however a journal paper and a conference paper would have seemed more appropriate. The attempt to relate it to questions of computation and representation is rather weak, and the reader would be better served referring to one of the standard Artificial Intelligence texts.

This is a collection of 23 papers presented at an international research seminar on Computer Assisted Learning (CAL) in the Humanities and Social Sciences, held in London in April 1986 with support from the British Economic and Social Research Council. The seminar dealt almost exclusively with the use of CAL in school classrooms.

The papers are clearly written, but short, which makes the book rather bitty. Despite the seminar's being styled 'international', only seven papers were from outside Britain, and a New Zealander provided the one contribution from outside Europe and North America. The organisers note that there are as yet few people undertaking research into the use of CAL in the social sciences. The participants almost without exception work in universities or colleges of education. Their disciplinary bases are mostly geography, history and economics, with one paper discussing the use of Prestel in political education.

The papers range over a variety of aspects of the use of CAL. Some describe empirical research studies or curriculum and courseware development through the Microelectronics Education Programme until 1986, the British contributions show that thinking has moved on from the earlier naive absorption with purely technical detail which characterised so much early work in computer education in many countries, including Australia.
particular set of ideological bases for CAL, Frances Blow states 'that pedagogic assumptions provide the basis for educational software . . .'.

Despite its price this could be a useful resource for the libraries of those institutions offering teacher training in computer education and for computer education support units. The relative handful of Australians engaged in research and evaluation related to CAL will find that it suggests some interesting questions deserving attention through research.

Barry W. Smith
The Australian National University


While perhaps not a world-shattering book, this one is certainly a timely arrival on the Australian scene. The idea of equipping students with individual computers is not new (it’s at least six years old) but it now seems to have caught the imagination of more than a few of our senile statesmen. For what other reason would political pressure from Canberra chemistry and physics. While there is evidence of steady progress, there is on computers used to examine students’ progress. There are papers on

don’t? Could it be the classroom computer? Perhaps it had better be the pedagogic assumptions provide the basis for educational software . . .'

Boutzev (page 103):

no suggestion that the new educational technology is going to spread like settings. There are two papers on computers used as simulators, and two date before they are made — but it does canvass the principal issues and answer, because it looks as if we shall soon be ‘in it’ up to our ears . . .!

As I said, this book is most probably timely and therefore important. It doesn’t dwell too much on the kind of facilities that might be available — these are changing so rapidly that any statements on these are out of date before they are made — but it does canvas the principal issues and report some encouraging successes in using computers in classroom settings. There are two papers on computers used as simulators, and two on computers used to examine students’ progress. There are papers on educating students in information systems, mathematics, engineering, chemistry and physics. While there is evidence of steady progress, there is no suggestion that the new educational technology is going to spread like wildfires without a great deal of nurturing and the infusion of substantial resources — this is not a manifesto for a revolution. To quote Boutzev and Bottzev (page 103):

The lack of hardware will not be the main obstacle to computerization of the educational process and individual work of students. The major problem is how to organise the rational use of micro-computers in engineering education, how to train students to use them in individual learning and in the problem-solving process. In the majority of cases one applies a pragmatic approach to these problems without having a good pedagogic basis.

The paper which struck me most forcibly was the last one, by M.E.A. Schmutzer of the Technische Universitat Wien on 'The Waning of the Post-Industrial University: No University of the Future?' He makes several telling points, as for example on pages 172-3:

Universities so far have been devoted to the principle of liberal access to knowledge. Now they are — to protect their own interests — well advised to look at the situation from a different point of view. Algorithmic knowledge cannot be liberally distributed any longer, only products of its application. . . Looking at the sum total of developments one comes to conclude that universities will have a hard time to resolve their conflicts and keep themselves distinct from ordinary business corporations.

Great oaks do indeed from little acorns grow!

This volume is the outcome of an IFIP WG 3.2 conference held at the University of Delft in April 1987. It is pleasing to note that this time the publisher arranged to have individual papers typeset uniformly, even though the final production was via an impact printer. Recommend this one for your library, and cogitate on Schmutzer’s paper!

John Lions
University of New South Wales


This book describes an information system planning method, called Strategic Value Analysis (SVA). The objective of SVA is to ensure that investments in information systems (IS) support the strategic objectives of the organisation. It is a top down approach, requiring an ongoing task force at middle management level and the understanding and support of top management. In essence the method involves decomposition of strategic objectives, development of a process model of the organisation using a data flow diagramming technique, and development of a data model of the organisation using an entity modelling technique. The constructive way in which these techniques are combined reflects the author’s many years of experience in this area as a consultant with Arthur D. Little, Inc. The SVA method would be most suited to a large organisation which has a formal strategic plan and believes that fundamental examination of its IS could have substantial payoffs.

Chapter 1 and 2 discuss the need for a systems planning method and the features of the SVA method in general terms. Chapter 3 provides an overview of the 10 steps in the SVA method. Step 1, 'Identify and Weight Strategic Business Objectives' is covered in Chapter 4 where a case study is introduced that is used for illustrative purposes throughout the rest of the book.

Chapter 5 covers Steps 2, 3 and 4 of the SVA process. These are, 'Define Top-down Business Processes and Data Flows', 'Review Existing IS', and 'Identify IS Capabilities that Help Processes to Meet Business Objectives'. The desired result from the exercises to this stage is the identification of a set of IS capabilities that are clearly connected, via level-by-level objectives, to the strategic objectives of the business.

Chapter 6 covers Step 5, 'Design Logical Data Base to Support Data Flows and Capabilities'. This is done using an entity modelling technique developed by the author (Curtice and Jones, 1982). The data model is prepared in parallel with the process model and is intended to provide the basis for long term IS integration given the relative stability of data compared to processes.

Chapter 7 covers Steps 6, 8 and 9. These are 'Synthesize Capabilities into Systems', 'Create Projects to Build and Install Systems and Architecture', and 'Develop Plan', the latter involving prioritizing and scheduling of projects. The SVA method provides an objective basis for attaching priorities to IS capabilities. Some useful guidelines are provided in respect of the other steps.

Step 7, 'Design Architecture to Support Systems and Data Base' is covered separately in Chapter 8. The distinction between logical and physical architectures is made and several common designs are reviewed. The relationships between the logical and physical views of the architecture and the need to see both as evolutionary is discussed.

The final step, Step 10, 'Maintain and Use The Plan' is covered in Chapter 9, where the claim that SVA provides better continuity from planning to implementation than other major IS planning/development methods is explained. Maintenance of the plan is seen as an ongoing process, with task force members remaining on the task force for two to three year stints, and the updated plan being formally disseminated annually.

The book contains one appendix, 'The SVA Metamodell', which provides a very useful summary of the SVA constructs and their relationships. This is provided both in narrative form and using the SVA entity modelling technique. At the end of each chapter are selected references and questions to help the reader relate SVA to their organisation. There is no index but this is not necessary in a book as short and as highly structured as this.

Overall, this book provides a clear and succinct description of a modern, yet proven, IS planning method. Despite its price it would be good value to the type of organisation likely to use the method.

Reference


Robert Reeve
University of New England


Book Reviews
The programming language C is derived from B which is derived from BCPL. ... B. ... B remains redundant, and many of the examples would be better presented via tables than verbally. There is a glossary of terms, a useful index, and many exercises at the end of each chapter. However I found his discussion of protocols while detailed, lacks the insight of Bertsekas and Gallager.

I have saved my discussion of Bertsekas and Gallager's book until last, because it is the book I value most. It is not replete with arcane technical engineering details and CCITT recommendation numbers. The authors, who teach at MIT, discuss the OSI layer model, but are not mesmerised by it. Their book is organised differently from the others: it surveys the common features among all the different types of network, extracts their common mechanisms and basic principles, and compares and contrasts them intelligently. It has the following chapters:

1. Introduction and Layered Network Architecture (30 pages).
2. Data Link Control and Communication Channels (80 pages).
3. Delay Models in Data Networks (94 pages).
5. Routing in Data Networks (126 pages).
6. Flow Control (40 pages).

The discussion is clear and thoughtful throughout. It casts light on how many protocols really work, and why they don’t always work as well as advertised (vide the discussion on the HDLC poll/final bit on page 88).

Beauchamp has been Director of Computer Services at the University of Lancaster in the UK, and he states that his book is suitable for senior undergraduate and postgraduate students, I would prefer to characterise it as ‘intended for other computer centre managers’. It will certainly suit managers who want to be informed properly about many aspects of current computer telecommunications, but not overwhelmed. It has a wide sweep, and it is the only one of the five to even mention the important new wonder protocols, MAP and TOP, from which we have been promised so much. It is generally well written, and has a good index. It hits the highlights of a great many topics, but does not plumb any depths. There are several straightforward exercises at the end of each chapter, with answers given in an appendix.

Halsall, who comes from the University of Sussex, offers his book as an undergraduate text. It is good on engineering background, and explains the standard aspects of protocols in a clear and workmanlike way without getting too enmeshed in detail. The OSI model figures prominently but the treatment of Local Area Networks (LANs) is a little meagre for my preference. The writing is sound, the author covers his ground at a steady, even pace, and there are few intellectual hurdles left for the reader to negotiate alone. There is a useful set of exercises at the end of each chapter, and I have been satisfied to use it as an introductory textbook for computer science undergraduates this year in place of Tanenbaum’s book.

Schwartz is a Professor of Electrical Engineering and Computer Science at Columbia University in New York. His book comes at the culmination of a long and productive career spanning forty years studying and teaching classical telecommunications. His book is by far the longest of the five, and possibly the most widely targeted for both computer scientists and electrical engineers, graduate and undergraduate. To quote from the preface, it ‘is organised around the layered architecture of the OSI model. [It] provides ... a brief introduction to IBM’s Systems Network Architecture (SNA) and to X.25’. The author has had a front row seat in the field for the past forty years, and so his views and opinions are more than usually relevant. His empathy with traditional telephone systems planning and traffic analysis shows through repeatedly in his general approach, the examples he chooses, and in the emphasis on particular points. His book has been carefully prepared with many diagrams, examples, exercises and references. I appreciate his treatment of the latter which are quoted both as footnotes and in a final consolidated list. However I find the typesetting and layout has a ‘brittle’ quality to it that I don’t enjoy. Moreover the author’s prose is frequently wordy and redundant, and many of the examples would be better presented via tables than verbally. There is a glossary of terms, a useful index, and many exercises at the end of each chapter. However I found his discussion of protocols, while detailed, lacks the insight of Bertsekas and Gallager.

McGraw-Hill (US), 586 pp., $90.00 (hardcover), $39.00 (softcover).

The Australian Computer Journal, Vol 20, No 1, February 1988
has the best discussion by far on collision avoidance methods for CSMA/CD networks.

By way of comparison, I decided to pick several well-known terms and see how they were treated by each author. All five books except Beauchamp's include a reasonable glossary of terms as an appendix (by way of atonement, Beauchamp has included a good final chapter on 'Recommended Reading' and a table of names and numbers for the major CCITT Standards Recommendations). All five refer to CCITT Recommendations X.21 and X.25, but the two American books do not dwell unduly on them. Purser lists no fewer than 100 CCITT recommendations in his index, and Beauchamp lists even more in a special table. While Beauchamp spends two pages on X.400 (Message Handling Service), Purser is the only one to describe it at reasonable length. All five mention ISDN, but only Schwartz discusses TCP/IP, an important legacy of the ARPANET project. Purser mentions Local Area Networks only in passing, the two English authors give them a fair to modest treatment, while only the American authors discuss them and the related IEEE 802 recommendations at length. Similar comments apply to discussions of DECNET and IBM's SNA. There must be some moral here about the state of network play on the two sides of the Atlantic!

In conclusion, among these five titles, I recommend:

Beauchamp, if you mainly want to 'Increase Your Word Power', i.e. derive a quick understanding of the wide-ranging terminology of this field and its origins and background.

Halsall, if you are seeking a general technical introduction to network hardware and software and their underlying principles;

Purser, if your problem is how to make sense of CCITT recommendations and how to choose the best combination of your local PTT's service offerings;

a combination of Schwartz, and Bertsekas and Gallager (possibly supplemented by Purser) if you are seeking an indepth, wide-ranging survey of current practice.

You could do worse than acquire all five!

References


John Lions
University of New South Wales

Letter to the Editor

COMMENT ON MILITARY GIS ARTICLE

In the paper by Nichol, Flebig, Whatmough and Whitbread ('Some Image Processing Aspects of a Military Geographic Information System', ACJ August 1987, page 154) a technique "PROC", for comparing the covariance structures of spectral bands in samples of remotely-sensed data, was described. A way to simplify the technique has now been found.

PROC, first proposed by Bogner (1981), found the covariance matrix \( C_X \) of a training area, and used eigenanalysis to find a diagonal matrix \( \Lambda \) and a unitary matrix \( E \) such that

\[
\Lambda^{-1/2}E^T C_X E \Lambda^{-1/2} = I.
\]

For each neighbourhood of an unknown image it formed the covariance matrix \( C_Y \), the transformed matrix

\[
C_W = \Lambda^{-1/2}E^T C_Y E \Lambda^{-1/2}
\]

and the distance

\[
D = (N/2) \Sigma \Sigma (C_W - \delta_{ij})^2
\]

of \( C_Y \) from \( C_X \), where \( M \) is the number of bands, \( N \) is the size of the neighbourhood and \( \delta_{ij} \) is the Kronecker delta function.

While PROC in this form is theoretically convenient, it can be simplified for computation, by noting that the double sum in \( D \) is the trace of a product of matrices which can be commuted cyclically. The definition is equivalent to

\[
A = C_X^{-1}C_Y
\]

\[
D = (N/2) \Sigma \Sigma (A_{ij} - \delta_{ij})(A_{ij} - \delta_{ij})
\]

When as much computation as possible is done during training, most of the work in finding \( C_W \), then \( D \), is in two matrix multiplications in the first step. The new path, via \( A \), requires only one matrix multiplication, so computation is roughly halved. The training step is also simplified, requiring only a matrix inversion instead of eigenanalysis.

REFERENCE


R.J. Whatmough
Electronics Research Laboratory
Defence Science and Technology Organisation
GPO Box 2151, Adelaide, SA, 5001

"20 DEADLY SINS . . ."

Thanks, Tony Montgomery, for your "20 Deadly Sins . . ." article, ACJ, 19, 4. For once I didn't immediately junk my ACJ. The ACJ is usually so esoteric that most members surely junk it. Can't we make the ACJ an optional extra, to preserve funds and forests?

Anyway, being a masochist, I read and bled! Indeed, have you been spying on me? By all means, let's get fair dinkum about professional standards and excommunicate offenders and dinosaurs. (Just give me time to clean up my act!) But don't let's follow the costs of the ASA which you reference, with membership costing double the ACS and prescribed PD seminars/courses costing a bomb!

Dick Kelly
Greenwich, NSW, 2065

RPS DISKETTES

In a strong bid to capture 20 per cent of the Australian diskette market by 1989, national distributor WPA Supply Company, St Leonards NSW is now marketing RPS High Focus diskettes, following the national launch in early February.

With two breakthroughs in coating technology the High Focus diskettes offer longer protection of stored data against wear, and also eliminate the chance of 'drop-outs' caused by inconsistent or weak signal strength.

RPS, short for Rhone Poulenc Systemes, have developed a unique 'enhanced isotropic' coating that gives the diskettes a consistently high signal strength. Whereas most diskettes have a safety margin ANSI clipping level between 35 and 40 per cent, High Focus diskettes offer 75 per cent, giving them greater tolerance of poor operating environments.

For further comment contact Grant Harrod, WPA Supply Company, (02) 438 1822.

CD-ROM CATALOGUE OF BIELEFELD UNIVERSITY LIBRARY

A CD-ROM catalogue of all the books in the University of Bielefeld library, West Germany, is being compiled by the university working in co-operation with the CCS system house, Hambury and Philips Germany.

A total of 1.4 million books are being catalogued on two CD-ROMs, covering 780,000 titles. One will contain a list of the books' titles while the other will have key words for locating them on the library's shelves.

Visitors to the library will be able to use personal computers to retrieve the data on the CD-ROMs which will be up-dated twice a year. Details of changes between updates will be noted on hard discs, accessible via the same PC's.

The first demonstration of these catalogues will be shown at the Hanover Fair, West Germany, from 16-23 March.

In Australia, Philips Telecommunications and Data Systems have recently established a fully operational CD-ROM bureau at North Ryde, offering CD-ROM preparation and development services, in addition to CD player supply and support.

RESEARCH PROJECT

Computer technology and its use by business organisations is the focus of a new research project by Griffith University lecturer Dr Rachid Zeffane.

His work is examining the impact of computer technology on administrative strategies within an organisational structure.

The work was funded last year by the University Research Grants Scheme Dr Zeffane received $7000 — and this year he will receive another $5000.

Dr Zeffane sent a comprehensive questionnaire to 500 business organisations throughout Australia and so far has received responses from 149. It is a 32 per cent response rate, which was in line with Dr Zeffane's expectations.

The survey will be followed up by in-depth interviews with about 16 of the organisations which responded to the questionnaire.

Dr Zeffane has begun a preliminary analysis of the data gained from the initial responses.

He has structured the questionnaire in such a way that it is possible to determine whether informational or operational uses of computer technology is the prime reason for its use in the organisations studied.

Already he has found that more than 29 per cent of respondents rank the most useful purpose served by computer technology as being for standardising management information systems.

The research is also gathering vital data on the uses of computer technology from the two separate angles of operational and informational use.

He sees the two uses as quite separate, but the weight given to each by the organisation's chief executive, the person to whom the questionnaire was addressed, was something revealed by answers to the questions.

"The operational side is the use of computers for the processing of a product, but computers may also be used for giving information for use in decision-making."

The information side of the usage of computers was shown to have a higher ranking than increasing productivity or reducing costs.

For further information contact Dr Rachid Zeffane on (07) 275-7758.

AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE — ASWEC 88

This year marks the twentieth anniversary of the recognition of software engineering as a discrete branch of computer science. The 1988 Australian Software Engineering Conference — ASWEC 88, held at the modern facilities of The Australian Defence Force Academy in Canberra over the period 11-13 May 1988, will mark this milestone in technological history.

ASWEC is conducted annually on rotation by the three peak professional bodies in the field of software engineering: the Australian Computer Society; The Institution of Engineers, Australia and the Institution of Radio and Electronics Engineers Australia. ASWEC 88 is the responsibility of the Australian Computer Society, with the Canberra Branch acting as hosts.

The extensive program will review software development over the past twenty years, project forward into the near and more distant future of software architecture, discuss marketing aspects, examine the future of the Australian software industry and consider the demands that software engineers will be making on computer hardware. A limited trade exhibition by invited software and hardware companies will complement the conference.

For further information contact the Conference Coordinator, Frank Poole, 7 Tobin Place, Holder, ACT, 2611. Telephone (062) 88 2884.
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 words 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 Bibliography and Subject Index of Current Computing Literature.

ACM Computing Reviews.

AMSI Mathematical Reviews.

CSA Computer and Information Systems Abstracts.

Data Processing Digest.

**ENGINEERING INDEX INC.**

IN SPEC Computer and Control Abstracts.

**INSPEC**

Electrical and Electronic Abstracts.

**ISI**

Current Contents/Comput Math.

**ISI**

Computer Math Citation Index.

**S P R I N G E R - V E R L A G**

Grenzegebiete.

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.