ISSN 004-8917

THE AUSTRALIAN COMPUTER JOURNAL

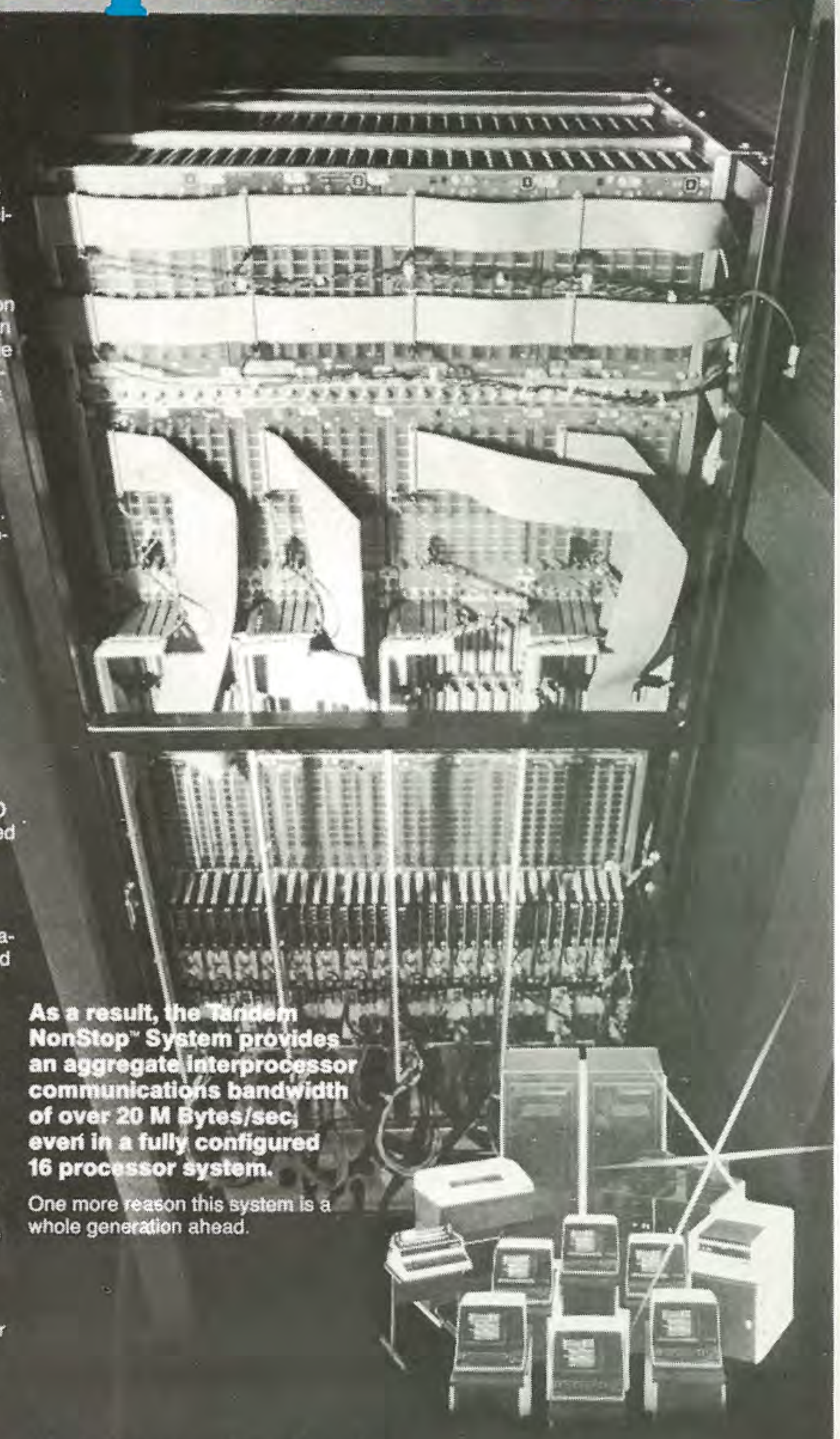
VOLUME 14, NUMBER 2, MAY 1982

CONTENTS

Published for Australian Computer Society Incorporated
Registered by Australia Post, Publication No. NBG 1124

NonStop™ Hardware

2 Unique Multiple Processor Link of a series.



The overall goal: a true high availability system with no single point of failure;

i.e., NonStop™ operation in an on-line transaction oriented system. The decision to provide continuous operation requires spreading the work across multiple processors. Immediately, efficient interprocessor communication becomes top priority. Any processor in the system must be able to absorb the workload of any other processor without disruption or loss of data integrity.

Shared memory has been a popular answer to the problems of interprocessor communication, but as a potential single point of failure, it is unacceptable for NonStop™ operation. Even if the single point of failure problems could be overcome, contention for the shared resource creates an unacceptable bottleneck, severely inhibiting performance and limiting system expansion.

The volume of interprocessor communication required to support fault tolerant operation in a transaction environment is significant. Far too much to use typical I/O bus connections between processors without clogging the system and making it I/O bound. Even with Tandem's high speed 4 M Byte/sec I/O channel.

DYNABUS was the solution:

We took all interprocessor communications off conventional I/O channels and put them on a unique and extremely fast interprocessor communications link, DYNABUS. This is a dedicated, two path, bidirectional line among all processors in the system, both paths running at 13 M Byte/sec. Either path is capable of handling all DYNABUS traffic if the other path fails.

Working under its own control, DYNABUS handles all interprocessor traffic for program assignments, job/data routing, priority decisions, health checks and housekeeping.

The DYNABUS contribution to performance is profound. Our multiple computer system has an effective four lane, no speed limit "data freeway" which doesn't become a bottleneck as the number of processors in the system is increased. Built-in buffers and packet multiplexing support optimal use of bandwidth.

As a result, the Tandem NonStop™ System provides an aggregate interprocessor communications bandwidth of over 20 M Bytes/sec, even in a fully configured 16 processor system.

One more reason this system is a whole generation ahead.

A whole generation ahead

TANDEM NonStop™ Systems

TANDEM COMPUTERS DISTRIBUTED IN AUSTRALIA BY:

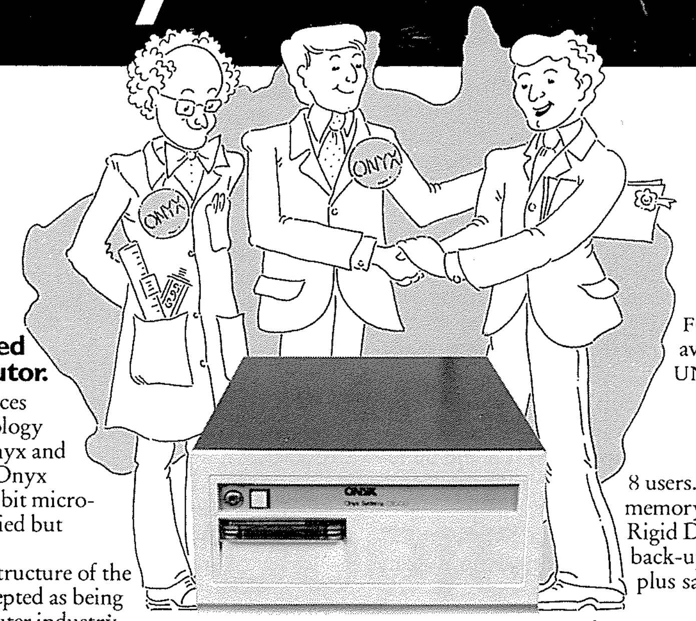
MIS MANAGEMENT INFORMATION SYSTEMS PTY. LTD.

3 Bowen Crescent, Melbourne, Vic. 3004 Telephone: (03) 267 4133	22 Atchison Street, St. Leonards, N.S.W. 2065 Telephone: (02) 438 4566	S.G.I.O. Building, Cnr. Turbot & Albert Sts, Brisbane, Queensland 4000 Telephone: (07) 229 3766	3rd Floor, 151 South Terrace, Adelaide, S.A. 5000 Telephone: (08) 212 1110

NCAMIS/60

Australia has a new computer company that's been around for years.

Onyx Australia.



Australia's only authorised Onyx distributor.

The unique advances in microcomputer technology initiated originally by Onyx and incorporated within the Onyx C5000/8000 series 8 and 16 bit microcomputers have been copied but never equalled.

The lean pricing structure of the Onyx systems is also accepted as being unparalleled in the computer industry.

Now, Impact Computers and Onyx Computers have combined and, with the formation of Onyx Australia, the recognised benefits of Onyx microcomputers are reinforced Australia wide by a network of Onyx specialist hardware support and applications advice.

The Onyx service policy.

There are a number of advantages to be gained by dealing with the only specialist Onyx distributor. Onyx Australia is the only company offering Australia wide hardware maintenance and service through STC. Spare parts are quickly available and down-time can be restricted to within hours.

More importantly however, Onyx Australia has a genuine policy of co-operation and dedicated service in its association with dealers and customers.

Expert advice, specialised attention and of course, generous dealer discounts are just a small part of the Onyx Australia service policy.

Key advantages to dealers also lie in the uniquely tangible benefits the Onyx C5000/8000 series provide to users in industry, business, government and the professions.

And Onyx Australia has a comprehensive range of applications programs to suit virtually every kind of user function.

Application programs.

General ledger, creditors, debtors, stock control, sales analysis. (With source available.) Applications suited to Insurance Brokers, Underwriters, Legal, Medical, Educational areas. Fashion, Pharmacy, Electrical retailers, Builders, Developers, and many more.

Software support.

Onyx Australia support groups for OASIS, UNIX and PASCAL will enable dealers to realise the full potential of Onyx systems. Plus screen editors, COBOL program generators and a range of DBMs.

Industry compatible versions of COBOL, PASCAL, BASIC, C BASIC 2, FORTRAN and C are available on OASIS, CP/M and UNIX operating systems.

Hard disk computer features.

Multi-tasking with up to 8 users. 64K to 1MB parity checking memory. 6MB to 144MB Winchester Rigid Disk. 12MB Cartridge tape back-up. Prices are from $7,000 plus sales tax.

Onyx. The only choice in microcomputers.

For a full range of information about Onyx microcomputers and the Onyx Australia dealer service policy mail this coupon to:

NSW.
Onyx Australia Pty Ltd.
55 Phillip Street, Parramatta, NSW 2150.
Telephone (02) 633 5222.

Other States.
Onyx Australia Pty Ltd.
14 Inman Road, Dee Why, NSW 2099.
Telephone (02) 982 7200.

Name

Address

Postcode Phone

ONYX

Onyx Australia Pty Ltd.
Incorporating Impact Computers and Onyx Systems (Aust) Pty Ltd.

CITY WEST OND23

News Briefs from the Computer World

"News Briefs from the Computer World" is a regular feature which covers local and overseas developments in the computer industry including new products, interesting techniques, newsworthy projects and other topical events of interest.

INSURANCE BROKERS BENEFIT FROM SANYO MBC 3000 SYSTEM AND LOCAL SOFTWARE

Over thirty users of a system that has been specially designed for both small and larger insurance brokers throughout Australia are benefitting through reduced costs, faster customer service, and are producing invoices, reminder notices and reports faster than any manual system.

Hunter Computers Pty Ltd has developed the 'Insurance Broker's System', designed for use with equipment from Sydney-based Sanyo Office Machines Pty Ltd.

The Sanyo MBC 3000 with 8" hard disk drives was chosen for its large memory capacity, Australia-wide service facilities, simple operation and excellent compatibility with the software package.

For over two years, a users 'club' has provided feedback to suggest and highlight any modifications to enhance the effectiveness of the system. This approach has the benefit of keeping all users completely up to date with the latest system programme.

A questionnaire has been designed to assist potential users of the system in evaluating whether or not the system would be effective in their particular application.

According to Mr. David O'Neill, Sales Director for Hunter Computers, "I believe that the Insurance Broker's System is the most popular and advanced of its type in Australia today. Now, by using Sanyo computers, we are able to install the system throughout Australia, thus leading to further enhancement of the software and benefitting the brokers."

Mr. Roger Daniels, Office Manager for Insurance Brokers, T.A. Markey & Co. Pty Ltd, a user of the system said "Our previous manual system required us to employ one full time and one part time typist in the preparation of accounts for new business, renewals and alterations. A 'day book' system was used to record payments to us and to the insurance companies.

Much work was required to check on items such as overdue accounts. Renewal and customer information was kept on a card system. With the business we are doing today, it would have meant an additional full time clerk/typist resulting in higher wages, office space and equipment, superannuation, sick leave, holiday pay and workers compensation premium.

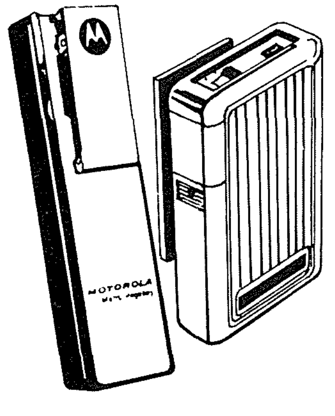
Our computer is capable of producing as many invoices as we demand, and also sorts and stores relevant segments of information for later presentation in summaries and reports. All the salesmen have to do is complete an 'input sheet' and the operator is able to key in the basic information. Often a whole morning's work can be printed an hour later and we automatically have the various accounting checks and balances simultaneously."

RSC-6 COMPUTER CATALOGUE

Tandy Electronics have just released the most comprehensive microcomputer catalogue yet produced in Australia.

The RSC-6 Computer Catalogue, with 48 colour pages of information on business, personal and educational computer hardware and software, is free at any Tandy Store or Computer Centre.

WHAT DOES RADIO PAGING DO FOR YOU



GUARANTEES:-
Greater **EFFICIENCY** Increased **PRODUCTIVITY** Higher **PROFIT**

SAVES:-
TIME; MONEY; LABOUR; PETROL; & MECHANICAL MAINTANCE

RENT BUY LEASE

FROM THE SPECIALISTS IN **BEEP & VOICE** PAGING

CAPITAL COMMUNICATIONS

958 1266
680 Willoughby Road
Willoughby

635 6466
6 Union Street
Parramatta

cemac

Interiors...Doors...Floors

Access Flooring

For computer rooms, sub-stations, control rooms or wherever continual access for maintenance of services is a requirement.

Partitions, Wall Panelling, Entrance Screens, Ceilings

Cemac offers a complete range of products and services to create functional and beautiful office interiors.

Fire Doors

High technology, 1, 2 and 3 hour rated doors are produced to Australian standards.

Project Management

Single source responsibility co-ordinating the entire complex interaction of all trades involved in tenancy design and installation.

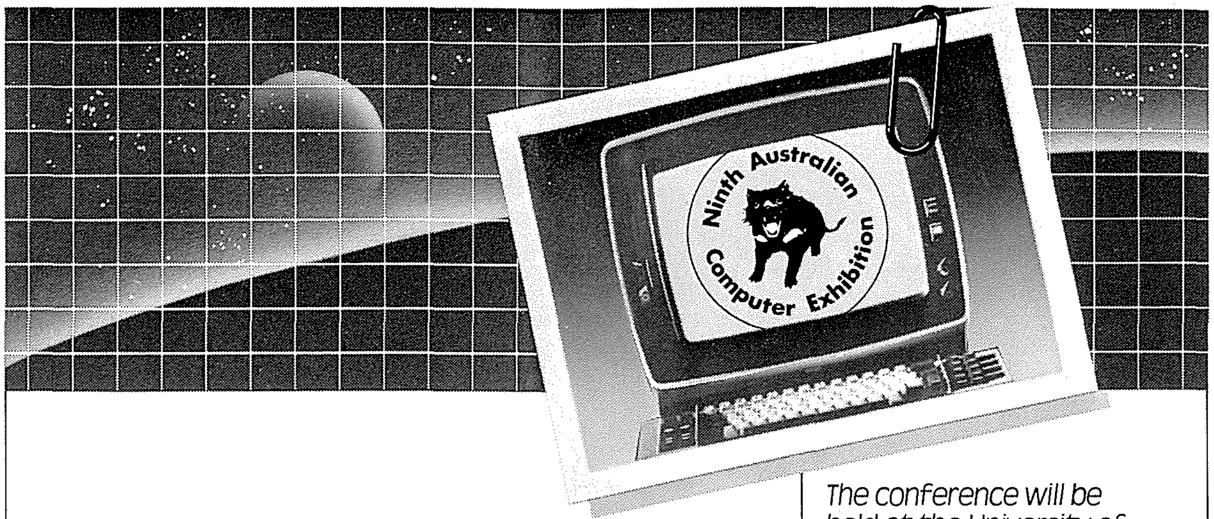
Renovations

The refurbishing of commercial buildings is a further highly specialised service offered by Cemac.

cemac

Sydney 290 3788
Melbourne 419 8233
Brisbane 221 5099
Adelaide 45 3656
Hobart 29 5444
Perth 444 7888

CEIN 0065



PROGRAM YOURSELF TO SEE THE LATEST IN COMPUTERS.

The Ninth Australian Computer Exhibition and Conference. Hobart, 1982.

This major computer event offers a rare opportunity to keep up with the latest advances in computer technology in the eighties. Leading Australian and international companies will be exhibiting the very latest in computer hardware and software, as well as related products. See millions of dollars worth of sophisticated equipment you may otherwise only read about.

Computer experts will be there in force.

Computer experts and members of the trade will be flying in from all over Australia, and the world, to see and contribute to this outstanding Exhibition.

The cost of airfare and accommodation is a small price to pay for access to such invaluable expertise and information.

Don't miss out. Program the following information into your memory banks.

When:
Tuesday August 24,
to Friday August 27, 1982.

Where:
Because of its size, the Exhibition will be held at two separate venues within two minutes walking distance of each other - the Hobart City Hall, and the Elizabeth Street Pier Building.

The conference will be held at the University of Tasmania and Wrest Point Hotel.

Exhibition hours:
Tuesday 10.00 am - 6.00 pm.
Wednesday 10.00 am - 6.00 pm.
Thursday 10.00 am - 6.00 pm.
Friday 10.00 am - 9.00 pm.

Hosted by:
The Australian Computer Society and the ACS Tasmanian Branch.

Arrange your accommodation and travel early to avoid disappointment.

For full details, write to:
Ninth Australian Computer Conference,
P.O. Box 216,
Sandy Bay. TASMANIA. 7005.
Phone: (002) 23 1824

Free entrance with business card or, for free trade tickets, write to or telephone:

Riddell Exhibition Promotions,
166 Albert Road,
South Melbourne. VIC. 3205.
(03) 699 1066

ANOTHER EXCITING RIDDELL EXHIBITION

RP520

Guest Editorial

This issue of the *Australian Computer Journal* is a special one devoted to the topic of Software Engineering. Although the term has been in use now for more than a decade, there is still no widely accepted definition. Broadly speaking it can be viewed as a discipline concerned with the development and utilisation of tools and techniques for the production of high quality software within budgeting and scheduling constraints. When the term was first invented in the late 60's, it was used in a somewhat provocative manner to indicate that all was not well with the way software systems were being designed and constructed. The software industry seemed to be plagued with late deliveries, cost overruns and an extremely low level of reliability of the final products. It was felt that the methodologies developed in other areas of human endeavour could also be applied to the production of software systems so that they were "well engineered" rather than just "programmed". Programming is an art that can be taught to most people. We now know that engineering software is an extremely difficult task requiring a great deal of discipline and skill.

During the last decade, there has been considerable activity in the software engineering field. Many research projects were initiated with the results being published in one of a number of new journals which came into existence to service the field, for example, IEEE Transactions on Software Engineering. The first International Conference on Software Engineering was held in Washington in 1975 and was primarily concerned with self-justification and demonstrations of the existence of Software Engineering as a discipline. Subsequent conferences however have dealt with every aspect of the software development life cycle. The sixth such conference will be held in Japan, in September this year and will focus on the general environment for the production of quality and user friendly software. Topics to be discussed include:

- Tools and Techniques of Software Engineering
- Requirements and Specifications
- Theoretical Foundation of Software Engineering
- Computer Aided Design and Production of Software
- Software Project Management and Human Factors
- Software Quality Control and Assurance
- Software Maintenance
- Software Metrics
- Software Engineering for New Computer Architecture, Distributed Systems, and Networks
- Impact of VLSI on Software Engineering
- Practices and Experiences of Software Production
- Education on Software Engineering

Most universities and colleges now offer courses in Software Engineering in an attempt to bridge the gap between their programming courses and the outside world where software lives on to be used, modified and maintained rather than just being assessed and then discarded. Project work involving programming teams is an important aspect of such courses. It is interesting to note that many employers are now advertising for software engineers rather than programmers since they have become aware of the dangers of hiring someone who may not be familiar with the appropriate tools and techniques to be used for the production of reliable software.

When I was first asked to be the guest editor for the special issue on Software Engineering, I was very pleased to be able to accept the invitation. Although I realized that it would be a very time consuming and arduous task to select the papers to be published from the many that would be submitted, nevertheless I felt that it would provide me with an excellent opportunity to obtain an overview of the work going on in Software Engineering in this country, particularly, in industry. There has been much criticism of the ACJ over the years from practitioners working in the field that the journal is dominated by the academics who publish incomprehensible papers of little interest to most of the members of the Austalian Computer Society. There have even been calls from time to time that the Journal be discontinued. I believed this special issue was a long awaited opportunity for industry to dominate with contributions describing the tools and techniques it employs in constructing software. This was not to be the case. Although the overall response to the call for papers was poor, the submissions from industry were almost non-existent. Either the techniques currently in use are so secret that no-one wishes to reveal them in the open literature or nothing worthwhile writing about is currently being utilized. The fact that there is only one paper in this issue from an author with a commercial background and then one who is working overseas reflects the state of the submissions and not my own particular bias. In fact, I had decided to give preference to articles submitted by authors working in industry but was unable to do so since they simply did not appear. In spite of all the complaints in the past, people did not avail themselves of the opportunity, when it was presented to them, of describing what they are doing.

In Dwyer's tutorial paper, an attempt is made to draw together the concept of "disciplined programming" and the COBOL language which although widely used in industry does not readily lend itself to the writing of correct programs. The paper discusses how a program might be proved correct in terms which can be readily comprehended by the practising COBOL programmer.

In the only paper included in this issue from an author outside the walls of academia, Clarke surveys the development and use of program-generators for commercial applications. He suggests that the use of such generators is on the increase and foreshadows a future where programming languages will be superseded by specification languages which will enable applications to be described in a non-procedural way and the operational system generated automatically from such a description.

The paper by Keedy and Richards argues that there are many benefits to be gained by treating files as information hiding modules rather than free-standing data structures. This extends a very important principle of software engineering expounded in the early 70's that information about a module should not be generally accessible to other modules irrespective of whether they need to use it or not. The data structures and procedures which comprises a module can be hidden behind a procedural interface, thereby making the software more understandable and hence more maintainable.

Parkin's paper represents a review of the work by Cho on the input space model of software testing. This work is extended to allow test programs to be generated for languages defined by BNF.

Although I have never been a supporter of Halstead's software science, I have included Lister's paper in this special issue as a counter-balance to the last article on this topic published in the ACJ in 1978. This and other publications suggested that software science might be making significant breakthroughs towards deriving some metrics for software quality. It appears however that the early promises have not been fulfilled and that the foundations of software science are in fact extremely weak.

Peter C. Poole,
Guest Editor,
University of Melbourne

Cobol, Comments and Correctness

B. Dwyer*

A tutorial introduction to 'disciplined programming', with suggestions for its use in a Cobol programming environment.

Keywords and phrases: disciplined programming, verification, proof of programs, Cobol, coinage analysis.
CR Categories: 1.3, 3.50, 5.24.

1. INTRODUCTION

The following shows how 'disciplined programming' can be applied to the day-to-day writing of Cobol programs. The main ideas of disciplined programming are these.

(1) A program can be shown to be correct by proof, but debugging can never guarantee correctness.
(2) The most easily understood programs are those with simple proofs.
(3) A proof should not be made retrospectively, but as part of the act of creating the program.
(4) Making a proof helps the programmer discover a better program.
(5) Some methods of program construction and some computer languages lead to easier proofs than others.

These points are demonstrated convincingly in E.W. Dijkstra's book, "A Discipline of Programming" (Dijkstra, 1976).

Cobol does not lend itself to formal proof as well as some other languages, for example Pascal (Wirth and Hoare, 1973). As a result our proofs will be verbal, rather than algebraic.

2. PRE-CONDITIONS AND POST-CONDITIONS

To 'prove a program' is to show that it has certain desired properties. As a rule, the most important property is for the program to generate the required output. We may also wish to know that the program will terminate, that it can never refer to non-existent array elements, or cause arithmetic overflow, and so on.

The way to define such requirements is as logical assertions. These are non-procedural statements about the final state of key program variables, typically the output files. They specify the desired relationship between the output variables and the input data. For some kinds of proof, they may also refer to hidden variables, such as the execution time, or the number of iterations of a loop.

The method of proof is to make logical assertions at various places in the program, and show that they are properly related. The most important assertions concern the initial and final values of variables. Assertions about their intermediate values are included to simplify the proof. These serve a similar function to theorems or lemmas in a mathematical argument, enabling a complex proof to be built from simple steps.

In a large program, the intermediate assertions serve a second useful purpose. If the program is best considered at several conceptual levels, the assertions can be made at corresponding levels of detail. For example, we may show that the input-output logic of a large program will correctly update its files, where 'updating' is as yet an undefined concept. We later make more specific assertions to define and prove the updating procedures. This avoids a clutter of unnecessary detail at the higher conceptual level.

It is normal to place assertions at the entry and exit point of each sub-routine, and at the head of each loop (Alagic and Arbib, 1978). Given this basic set of assertions, it is possible to derive assertions about any other point in the program, or to complete a detailed proof, by purely mechanical reasoning. The basic assertions themselves cannot be derived mechanically, but rely on a proper understanding of how the program works. A basic set of assertions is therefore a proof in principle: anyone with sufficient motive could put them to the test. A program without assertions cannot be checked, because it lacks a definition of what it should do.

We will adopt a structured subset of Cobol that forbids GO TO's and implements all loops by PERFORM . . UNTIL statements. It will then be enough to make assertions about the entry and exit of each procedure (whether sub-program, section, or paragraph). Because loop bodies must be written as separate procedures, their entry and exit conditions will replace those placed at the head of loops. This slight departure from normal practice is made for the sake of uniformity.

(If we had adopted a less structured form of Cobol, the same rule for choosing a basic set of assertions would apply: one should be placed at each entry and exit point of a procedure. Since every procedure-name creates a procedure entry and every GO TO creates an exit, far more assertions would be needed than in an equivalent well-structured program. This is precisely the reason why structured programs are easier to understand. However, using GO TO's with moderation is acceptable, provided that each is documented by an assertion.)

An assertion made about conditions at the exit from a procedure is called a 'goal', 'result', or more commonly a 'post-condition'. An assertion about the entry to a procedure is called an 'assumption', 'initial condition', or 'pre-condition'.

3. GOOD AND BAD ASSERTIONS

To demonstrate the use of assertions, let us consider the definition of a procedure, CONVERT-TO-JULIAN. This will be as follows.

*Department of Computing Science, The University of Adelaide, Box 498, GPO, Adelaide, South Australia 5001. Manuscript received January 1982.

Pre-condition – DATE-GIVEN should contain a valid date of the form YYMMDD in the range 00/01/01 to 99/12/31.

Post-condition – JULIAN-DATE contains an integer in the range 0 to 36524, equal to the number of calendar days that DATE-GIVEN follows 00/01/01 (1st January 1900).

These assertions may either be written as comments in the program itself, or kept as supporting documentation. An advantage of comments is that they are more likely to be read, and therefore more likely to be kept up-to-date. A defensive programmer might even write assertions in the form of debug statements.

By adopting certain conventions, writing assertions can be less of a chore. We will assume the following conventions.

(1) Because DATE-GIVEN is an input, but not an output, we assume that it is not modified by the procedure.

(2) Because JULIAN-DATE is an output, but not an input, we assume that its initial value is immaterial.

There are further conventions that we may adopt. CONVERT-TO-JULIAN might refer to a table giving the length of each month. This table would then be an input, and the pre-condition would assume that it had suitable contents. However, these details are of little immediate interest to the user of CONVERT-TO-JULIAN, and are better documented at a more global level. Likewise, the procedure might corrupt temporary variables. Again we have the choice of documenting this at the current level or more globally. Our decisions should be dictated by the level of anticipated program maintenance. By reading the text of CONVERT-TO-JULIAN we can easily spot which variables are referred to or modified. If they are not documented locally, we should expect to find them documented at a more global level. Every variable used in the procedure should be documented somewhere, if only to state that it is a scratch-pad.

The pre-condition for CONVERT-TO-JULIAN requires that DATE-GIVEN should be a valid date. The result when it is not valid is undocumented, so by convention is undefined. There is no promise that an invalid date will not cause CONVERT-TO-JULIAN to fail. The onus is on the user of any procedure to ensure that its assumptions are met. In a similar vein, the post-condition of a procedure should not document the final values of temporary variables. Once it has been stated that such side-effects take place, users of the procedure have a right to expect that they are features that will be permanently supported.

We may contrast our assertions with the following documentation.

"CONVERT-TO-JULIAN converts a date to Julian form, by reference to MONTH-TABLE. The value of LEAP-YEAR is set to 1 if it is a leap year."

This is bad for at least three reasons. First, it does not specify how a date is to be represented, or what is meant by a "Julian form". (Perhaps it is YYDDD.) Second, it would be trivial to discover that the procedure refers to MONTH-TABLE. Third, the setting of LEAP-YEAR ought to be of no interest to anyone.

Pre-conditions and post-conditions should document every output variable that contains a result, and every input parameter. They should not concern themselves with the internal workings of the procedure, or its side-effects. It is probably better for them to use problem-oriented language (Shneiderman, 1980), but they should avoid meaningless jargon.

4. SEQUENCES OF STATEMENTS

Structured programming allows complex procedures to be constructed in three ways; sequence, selection, and iteration. We need to reason about such structures to prove the properties of complete programs. A sequence of statements is particularly easy to check. We need only to show that the post-condition of each statement matches the pre-condition of the one that follows it. Consider this procedure.

```
ADD-ONE-WEEK.
    PERFORM CONVERT-TO-JULIAN.
    ADD 7 TO JULIAN-DATE.
    PERFORM CONVERT-TO-YYMMDD.
```

To prove its correctness, we must know its pre-condition and post-condition, and also those of its component statements. Let us assume the following assertions for ADD-ONE-WEEK.

Pre-condition – DATE-GIVEN should be a valid date of the form YYMMDD, in the range 00/01/01 to 99/12/31.

Post-condition – RESULT-DATE contains a date in the format YYMMDD, that follows DATE-GIVEN by 7 calendar days.

We will assume that CONVERT-TO-JULIAN is documented as before. The ADD statement is documented by the Cobol compiler manual (we hope!), and we will assume that CONVERT-TO-YYMMDD is documented by the following pair of assertions.

Pre-condition – JULIAN-DATE is an integer in the range 0 to 36524.

Post-condition – RESULT-DATE is a date in the format YYMMDD, that follows 1st January 1900 (00/01/01) by JULIAN-DATE calendar days.

To prove the correctness of ADD-ONE-WEEK we need to show that;

(1) The pre-condition of ADD-ONE-WEEK satisfies the pre-condition of CONVERT-TO-JULIAN.

(2) The post-condition of CONVERT-TO-JULIAN satisfies the pre-condition of the ADD statement.

(3) The post-condition of the ADD statement satisfies the pre-condition of CONVERT-TO-YYMMDD.

(4) The post-condition of CONVERT-TO-YYMMDD satisfies the post-condition of ADD-ONE-WEEK.

The first is true by inspection. The second is true provided that JULIAN-DATE has been defined with at least 5 digits, guaranteeing that overflow will not occur. The third test is not met, because after the ADD, JULIAN-DATE will have the range 7 to 36531. However, the fourth condition is true by inspection. The attempt at proof reveals that the stated pre-condition of ADD-ONE-WEEK is incorrect. The range of DATE-GIVEN should have been stated as from 1st January 1900 to only the 24th December 1999.

Although many find it less intuitive, it is also possible to construct proofs working back from the post-condition of a sequence towards its pre-condition. One advantage of this method is that we usually have more precise requirements for goals than for initial conditions. We can then derive the minimum necessary pre-condition from the goal.

5. CONDITIONAL STATEMENTS

When we have a selection between statements, we can reason back from the post-condition to find the initial conditions that apply to each alternative. We can then decide what conditions need to be evaluated in order to make the proper choice.

Consider determining the hourly rate of pay for an employee. The rate depends on two factors, the employee's own basic rate, and a penalty rate that may apply to the job. The hourly rate is given by the following rules.

(1) It is either the basic rate or the penalty rate.
(2) Jobs without a penalty rate are shown by a penalty rate of zero, in which case the hourly rate is always the basic rate.
(3) The hourly rate is not less than the penalty rate.
(4) The hourly rate is not less than the basic rate.

From the first rule alone, we can see that the procedure will have the following form.

```
    FIND-HOURLY-RATE.
        IF condition-1;
              MOVE PENALTY TO HOURLY-RATE
        ELSE
              MOVE BASIC TO HOURLY-RATE.
```

It is not hard to guess condition-1, but let us attempt to derive it by the proof process.

The goal or post-condition is given by the above four rules. For this problem, it is easy to express the rules as Cobol conditions, giving our argument more precision. (It is not always so easy.) The equivalent Cobol conditions are these.

```
(1) HOURLY-RATE = BASIC OR HOURLY-RATE =
          PENALTY
(2) HOURLY-RATE = BASIC OR PENALTY NOT =
          ZERO
(3) HOURLY-RATE NOT < PENALTY
(4) HOURLY-RATE NOT < BASIC.
```

(Notice how we handle the implication in rule (2). We can always translate "A implies B" into "B OR NOT A". If the hourly rate is not the basic rate, then the penalty rate must be non-zero.)

To find the conditions under which the statement

```
    MOVE PENALTY TO HOURLY-RATE
```

is the correct choice, we reason backwards from the goal. Anything that is true of HOURLY-RATE after this move, should be true of PENALTY before the move, otherwise the goal will not be achieved. So we can derive the pre-condition of the move by re-writing the post-condition, substituting 'PENALTY' wherever we see 'HOURLY-RATE'. We obtain this pre-condition

```
(1) PENALTY = BASIC OR PENALTY = PENALTY
(2) PENALTY = BASIC OR PENALTY NOT = ZERO
(3) PENALTY NOT < PENALTY
(4) PENALTY NOT < BASIC.
```

The first and third conditions will always evaluate as true, so they can be ignored. The fourth is what we may have expected to find for condition-1; but what is the meaning of the second condition? Our derivation has shown us that condition-1 should be the conjunction of the second and fourth conditions.

```
    (PENALTY = BASIC OR PENALTY NOT = ZERO)
    AND (PENALTY NOT < BASIC)
```

By separating the cases of equality and inequality, we can write the same thing in a more enlightening way.

```
    (PENALTY = BASIC)
    OR (PENALTY > BASIC AND PENALTY NOT =
          ZERO)
```

This form reveals that the move is correct when both rates are equal, or when the penalty rate is the greater, provided that it is not zero. The move would be incorrect in the case that the penalty rate exceeds the basic rate, and is also zero. This could only arise when the basic rate was negative. Rule (2) tells us to choose the basic rate, whereas rule (3) tells us to choose the penalty rate. Either the rules are inconsistent or they imply that the basic rate can never be negative. It seems unlikely that we would want to handle negative rates, so we can make BASIC NOT < ZERO a pre-condition of the procedure as a whole. Condition-1 then reduces to the expected

```
    PENALTY NOT < BASIC.
```

Although proof techniques have not produced any surprises in condition-1, they have alerted us to a hidden assumption. With a negative basic rate and a zero penalty rate, it would be impossible to satisfy all the conditions of the goal.

For completeness, we should now find the correct pre-condition for the alternative move statement,

```
    MOVE BASIC TO HOURLY-RATE.
```

This is done in the same way as before, this time re-writing 'BASIC' for 'HOURLY-RATE' in the post-condition. There are no surprises, and this is left as an exercise for the reader. The condition we obtain is,

```
    BASIC NOT < PENALTY.
```

This is not the exact inverse of condition-1, reflecting the fact that either move will serve when both rates are equal.

6. ITERATION

Disciplined programming helps the programmer most in the proper construction of loops. We shall consider two examples. The first illustrates how assertions are used to reason about iterative programs. The second shows how formulating assertions helps us find a solution to a problem.

6.1 Copying a File

Consider the following procedure to copy the records of IN-FILE to OUT-FILE. (All records are of the same type.)

```
    COPY-THE-FILE.
        OPEN INPUT IN-FILE,
             OUTPUT OUT-FILE.
        MOVE "N" TO END-OF-FILE.
        PERFORM READ-A-RECORD.
        PERFORM COPY-A-RECORD
             UNTIL END-OF-FILE = "Y".
        CLOSE IN-FILE, OUT-FILE.
        STOP RUN.
    COPY-A-RECORD.
        MOVE IN-RECORD TO OUT-RECORD.
        WRITE OUT-RECORD.
        PERFORM READ-A-RECORD.
    READ-A-RECORD.
        READ IN-FILE, AT END
             MOVE "Y" TO END-OF-FILE.
```

We focus our attention on the loop body, COPY-A-RECORD. Since this procedure uses the record in IN-RECORD, it clearly expects that end-of-file has not been detected. But at the exit from the procedure, a further READ has been issued, so that end-of-file may have been detected there. However, the UNTIL condition of the PERFORM statement ensures that, if there is a further iteration, end-of-file was not detected. The pre-condition and post-condition of a loop body are always related in this way. The post-condition alone does not guarantee the pre-condition, but the post-condition and the complement of the UNTIL condition together must guarantee it.

A loop has a number of other properties. Once the UNTIL condition has been satisfied, the loop should have achieved its goal. Therefore the post-condition of the loop body, together with the UNTIL condition itself, must equal the goal of the whole loop. The initialisation for any loop can be deduced logically: we have to ensure that either the pre-condition of the loop body is satisfied, or the goal is satisfied already.

We will postulate the following pre-condition for the loop body, COPY-A-RECORD.

(1) Both files are open.
(2) End-of-file has not been detected, and END-OF-FILE NOT = "Y".
(3) On the nth iteration, the nth record of IN-FILE is in IN-RECORD.
(4) On the nth iteration, the first (n–1) records have been copied to OUT-FILE.

The post-condition is similar.

(1) Both files are open.
(2) END-OF-FILE = "Y" only if the end of file has been detected.
(3) If the end-of-file was not reached, the (n+1)th record of IN-FILE is IN-RECORD.
(4) On the nth iteration, the first n records have been copied on OUT-file.

We see that, provided that the UNTIL condition is not satisfied, the post-condition for the nth interation satisfies the pre-condition for the (n+1)th. If the UNTIL condition is satisfied, all the records must have been copied. The loop terminates in just as many iterations as there are records to copy.

The initialisation for the loop must be chosen to satisfy the pre-condition of the loop body. Therefore the outer procedure must open the files and issue the first READ. Also if the UNTIL condition is initially satisfied, the goal must have already been reached trivially. This is the case of copying an empty file.

The argument used to prove a loop is always inductive. We show that if the pre-condition of the loop body is satisfied on the nth iteration, it will be satisfied on the (n+1)th. Hence, provided that it is satisfied on the first, it will be satisfied on every iteration.

(The treatment of PERFORM with the VARYING option is similar, provided that we make allowance for the hidden operations on loop variables.)

6.2 Designing Loops

The disciplined method of constructing a loop is as follows.

(1) Formulate the goal of the PERFORM . . UNTIL statement.
(2) Choose a pre-condition and a post-condition for the loop procedure so that,
    (a) The post-condition guarantees the goal, if the UNTIL condition is satisfied.
    (b) The post-condition guarantees the pre-condition otherwise.
    (c) The pre-condition can be satisfied trivially by suitable initialisation.
(3) Choose a loop body that correctly relates the pre-condition and post-condition.
(4) Ensure that this procedure always makes progress towards the goal.
(5) Choose appropriate initialisation statements.

6.3 Coinage Analysis

To illustrate how disciplined programming can be applied to a real problem, we consider 'coinage analysis'. The problem is to find how many coins of different denominations are needed to make a payment in cash. In practice, the requirements for many such payments would be added together.

We shall assume that N-COINS different coins (or notes) are involved. The value of each coin is given by the array VAL. The objective is to find the value of the array QTY, which specifies the quantities needed of each coin. We can formalise the objective as follows.

"The sum of QTY(i) * VAL(i), as i ranges from 1 to N-COINS, should equal the amount to be paid."

We must take care: this objective may be difficult or even impossible to reach. (We have set the task of solving a Diophantine equation.) One way to be certain that there is a solution, is to know that the amount to be paid is a whole multiple of the value of the smallest coin. We shall assume it as an initial condition.

Having stated the goal, we can suggest a possible post-condition for the loop body.

"The sum of QTY(i) * VAL(i), as i ranges from 1 to N-COINS, should not exceed the amount to be paid."

This is a good candidate, because it includes the goal as a special case. It is also easy to satisfy initially, by setting all the quantities to zero. The loop body can make progress by reducing the amount by which the amount payable exceeds the sum of the products. Therefore, if the loop body increases the quantity of any coin by at least one, the loop is bound to terminate. Let us explore the consequences of this choice.

It will be necessary for the loop to keep track of the sum of the products of QTY and VAL, or it will be impossible to test for the goal with a Cobol UNTIL clause. We will use the variable AMOUNT-PAID for this purpose. We therefore propose the following post-condition for the loop body.

(1) AMOUNT-PAID equals the sum of VAL(i) * QTY(i), as i ranges from 1 to N-COINS. (We need this to be able to test for the goal.)
(2) AMOUNT-PAID never exceeds AMOUNT-PAYABLE. (This is how we measure progress towards the goal.)
(3) (AMOUNT-PAYABLE – AMOUNT-PAID) is always a whole multiple of the value of the smallest coin. (We include this to ensure that an initially feasible problem does not turn into an insoluble sub-problem.)

The pre-condition will be similar, except that we may assume that AMOUNT-PAID is strictly less than AMOUNT-PAYABLE.

The form of our solution will therefore be as follows.

```
COINAGE-ANALYSIS.
    Set all quantities to zero.
    MOVE ZERO TO AMOUNT-PAID.
    PERFORM INCREASE-AMOUNT-PAID
        UNTIL AMOUNT-PAID = AMOUNT-
        PAYABLE.
INCREASE-AMOUNT-PAID.
    Choose a coin whose value does not exceed
    (AMOUNT-PAYABLE – AMOUNT-PAID).
    Increase the quantity of that coin by at least one.
    Update the value of AMOUNT-PAID.
```

(The reason that the loop body must choose a small enough coin is to preserve the second assertion above. We do not want to overshoot the goal!)

The proposed loop body is not yet satisfactory. It does not maintain the third condition above. It must choose coins of a value that is a multiple of the smallest, or there is a danger that it will not be able to finish. One way to make sure that the amount left to pay is a multiple of the smallest coin, is to insist that all entries in the VAL array are multiples of the smallest coin. (Of course, if the smallest coin is 1 cent, this is no problem. However, if the smallest coin were a 10 cent piece, and the set also included 25 cent coins, the procedure could become blocked.) We shall add the requirement to our list of initial conditions.

There is nothing in our procedure that prevents it always choosing the smallest coin. That is always a safe choice, but also the least efficient! Clearly part of the goal has not been stated. Is it perhaps to use the fewest coins? Suppose that we add that to the goal. The corresponding post-condition for the loop body would be that AMOUNT-PAID uses the fewest coins. We must then hope to prove that if this were true on one iteration, it would be true on the next. Unfortunately we cannot do so. Imagine that the coins available are 25 cent, 10 cent, and 1 cent pieces. The best way to pay 30 cents is with three 10 cent pieces; but suppose that the procedure has already chosen a 25 cent piece. This is certainly the simplest way to pay an amount of 25 cents. It is also the optimum first choice in paying 26, 27, 28, or 29 cents, but the procedure would be blocked from finding the best way of paying 30 cents. To find the least number of coins from this set requires a backtracking algorithm, hardly justified by the problem. Let us say that the goal is to make the payment in a "reasonably efficient" way.

With these new considerations in mind, we can make the loop body more specific. At each iteration, it should choose the largest-valued coin possible. Progress towards the goal will be most rapid if it then uses as many coins of that value as it can.

```
INCREASE-AMOUNT-PAID.
    Choose the largest coin whose value does not
    exceed (AMOUNT-PAYABLE – AMOUNT
    PAID).
    Increase the quantity of that coin as much as
    possible.
    Update the value of AMOUNT-PAID.
```

It simplifies the choice of the largest coin to have the VAL array ordered, with the coins decreasing in value from first to last. We add this further assumption to our list of initial conditions. We have now derived the following loop body.

```
INCREASE-AMOUNT-PAID.
    MOVE 1 TO COIN.
    PERFORM CHOOSE-COIN
        UNTIL VAL (COIN) NOT >
        (AMOUNT-PAYABLE – AMOUNT-
        PAID).
    COMPUTE QTY (COIN) =
        (AMOUNT-PAYABLE – AMOUNT-
        PAID) / VAL (COIN).
    COMPUTE AMOUNT-PAID = AMOUNT-PAID
        + QTY (COIN) * VAL (COIN).
CHOOSE-COIN.
    ADD 1 TO COIN.
```

7. DISCUSSION

There are some improvements that can be made to this procedure. It is slightly more efficient to keep track of the difference between AMOUNT-PAYABLE and AMOUNT-PAID, than of AMOUNT-PAID itself. Also, it is futile for the loop using CHOOSE-COIN to start with COIN = 1 each time. It is better to start where the preceding iteration left off. It is not hard to modify the assertions to take these changes into account. (This can be an exercise for the reader.) But it remains true that our algorithm is certainly not the same as the standard solution to this problem, which is as follows.

```
COINAGE-ANALYSIS.
    MOVE 1 TO COIN.
    MOVE AMOUNT-PAYABLE TO AMOUNT-
        UNPAID.
    PERFORM ALLOCATE-A-QUANTITY
        UNTIL COIN > N-COINS.
ALLOCATE-A-QUANTITY.
    COMPUTE QTY (COIN) =
        AMOUNT-UNPAID / VAL (COIN).
    COMPUTE AMOUNT-UNPAID = AMOUNT-
        UNPAID – QTY (COIN) * VAL (COIN).
    ADD 1 TO COIN.
```

This solution is undoubtedly more attractive in certain respects: it is shorter and more efficient. However, the test to ensure that AMOUNT-UNPAID reaches zero is mysteriously absent! How can we prove that the standard solution is correct? The key step must be to discover the post-condition of the loop body, ALLOCATE-A-QUANTITY. Its assertions seem to be the following.

(1) AMOUNT-UNPAID is less than any VAL (i), as i ranges from 1 to (COIN – 1).
(2) AMOUNT-UNPAID is not negative.
(3) AMOUNT-UNPAID is a whole multiple of the value of the smallest coin.
(4) The quantities of all coins preceding COIN have been chosen in a "reasonably efficient" way.
(5) AMOUNT-UNPAID equals the value of AMOUNT-PAYABLE, less the sum of QTY (i) * VAL (i), as i ranges from 1 to (COIN – 1).

The first of these assertions guarantees that the procedure achieves a useful goal. Once the loop is complete, AMOUNT-UNPAID must be less than the smallest coin, the best that can be achieved.

Given valid data, the standard solution and our disciplined solution will produce the same result. They rely on the same set of initial assumptions. But given invalid data, (e.g. to pay 4 cents in 5 cent pieces), the standard solution will terminate with an incorrect result, whereas the disciplined solution will fail. (The value of COIN will exceed N-COINS.) This could be considered a point in favour of our disciplined solution: it does not ignore its mistakes. Some might argue that the standard solution puts the cart of "reasonable efficiency" before the horse of getting a correct result.

Of course, nothing in the disciplined approach forced us to obtain the solution that we did. It is possible, that guided by experience or foresight, we could have reached the standard solution instead. Or again, we might have discovered a new and even better solution. Disciplined programming is not a substitute for experience or commonsense. Indeed, its very thoroughness can be a fault. By starting with a badly chosen set of assertions, it becomes possible to derive a correct, but very messy program. Certainly the converse is true. Messy programs invariably rely on needlessly complicated assertions. As an illustration, consider the following solution.

```
     COINAGE-ANALYSIS.
          MOVE AMOUNT-PAYABLE TO AMOUNT-
               UNPAID.
          MOVE 1 TO COIN.
          MOVE ZERO TO QUANTITY.
          PERFORM CHOOSE-A-COIN
               UNTIL COIN > N-COINS.
     CHOOSE-A-COIN.
          IF VAL (COIN) > AMOUNT-UNPAID
               MOVE QUANTITY TO QTY (COIN)
               ADD 1 TO COIN
               MOVE ZERO TO QUANTITY
          ELSE
               ADD 1 TO QUANTITY
               SUBTRACT VAL (COIN) FROM
                    AMOUNT-UNPAID.
```

The number of coins of each value is determined by repeated subtraction, rather than by division as in the previous solutions. Despite this, the procedure avoids nested loops. It is not easy to understand the loop that remains, however. The assertions for the loop body are more complicated than for the other solutions. Finding them is left as an exercise (or challenge) for the reader.

It would be wrong to assume that disciplined programming in practice is carried out as formally as in these examples. A programmer would not always formalise the post-condition before writing a procedure. It is more likely, especially for trivial problems, that the procedure will come first and the assertions second. If that is so, an essential third step is to check that they agree. This reveals errors when they are cheapest to correct. It was not immediately obvious that the employee's hourly rate problem contained a hidden assumption. Nor was it evident that the standard coinage analysis procedure does not guarantee to choose the minimum number of coins. (In practice, the sets of coins available in Australian, British, or American currencies are such that it will. However, it is not easy to find the general conditions for the procedure to give the optimum choice.) It is unlikely that these potential bugs would be revealed by testing. A programmer is likely to bring the same preconceptions to devising test data as to writing the procedures.

Disciplined programming provides several advantages. It helps the programmer to define the problem and find a solution. Assertions cross-check the program, exposing latent bugs. They provide superb documentation for future program maintenance. Any tool can be used badly, but a good tool teaches its user new skills. Disciplined programming is an excellent tool.

8. ACKNOWLEDGEMENT

The author wishes to thank Barbara Kidman of the University of Adelaide, and Ian Watson of the South Australian Institute of Technology for reviewing this paper, and offering helpful suggestions for its improvement.

9. REFERENCES

ALAGIC, S. and ARBIB, M.A. (1978), *The Design of Well-structured and Correct Programs,* Springer Verlag, New York.

DIJKSTRA, E.W. (1976), *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, New Jersey.

SHNEIDERMAN, B. (1980), *Software Psychology,* Winthrop, Cambridge, Mass., pp. 67-69.

WIRTH, N. and HOARE, C.A.R. (1973), "An axiomatic definition of the programming language Pascal", *Acta Informatica,* 2, 4, 335-355.

A Background to Program Generators for Commercial Applications

Roger Clarke*

The emergence and key features of program generators are explained. Examples are given of the appearance and use of one particularly advanced product.

Keywords and Phrases: application generator, DELTA, macro language, macro processor, pre-processor, program generator, software portability, specification language, very high level language.
CR Categories: 41.2, 4.22.

INTRODUCTION

Since the dawn of programming better methods have been sought. One major focus has been upon efficiency in the use of processor and main memory resources, and in some circumstances these factors remain paramount.

Another focus has been on the manner in which programs are prepared. Varying degrees of maturity have been reached in the many aspects of languages. Their power has developed to such an extent in fact, that they offer far more than that needed by the vast majority of applications. As a result of this the problems arise of finding sufficient staff who are sufficiently highly trained to handle such languages, then constraining them to a narrow (and partly arbitrary) discipline in its use.

In order to combat such problems new methods of program preparation are emerging. These depend on parameter driven utility programs which generate a high level language program. Sub-problems may still require the power of the host language; for such cases it is necessary to be able to insert code into the appropriate location in the generated program. It is reasonable to view such program generators as preliminary attempts at future higher level languages. They are however identifiable products, and have some characteristics different from existing languages. This article will deal with them independently from questions of language design.

After a brief discussion of the reasons which stimulated the production of program generators, their emergence is traced and the concepts central to the theory are presented in stepped form. Examples are given based on one such product, and brief comments provided on the impact of the new tools.

THE STIMULUS

Modern theories of system and program development are poorly served by old languages and programming environments. Yet the enormous investment in software and in trained software development staff precludes a simple-minded revolution. One approach to provide a 'bridging' technology between old and new is to install a pre-processor before the compiler, to enable and/or require programmers to write in structured style, despite the weaknesses of the host language. In addition other deficiencies in the language can be catered for. An important product in this field was MetaCOBOL (see ADR, 1974a, 1974b), a commercial application of the 'Stage II' generator (Waite, 1974). It offered the ability to create additional verbs (case-construct, in-line PERFORM, initialise-table), to improve syntax (explicit ENDIF, a quasi-local variable feature), to recognise multiple alternative short forms, and to 'massage' the layout of the code for consistent presentation and indentation — critical factors in making programs readable and maintainable by persons other than the author.

The problem with conventional high-level languages, even when front-ended in this way, is that their power and complexity demand considerable expertise on the part of the programmer. Few problems arise in commercial programming that aren't capable of appropriate solution; but there are far too few suitably trained people to do the solving. Given that the vast majority of development groups work within a fairly small set of (partly consciously chosen) techniques, the full power of the host language could be foregone.

An additional problem is the matching of programming technology to the system analysis and design technologies that precede it in the application-software production-line. It is now fairly clearly established that multiple languages at different levels of abstraction are necessary (Hawryszkiewycz, 1981) and that therefore language translation problems will occur. In addition these languages can be expected to require some time yet before they stabilise, and the likelihood of multiple alternative languages at any given level of abstraction seems to be quite high. It is therefore desirable that the interface between the design and the programming syntaxes be supported by a powerful macro-language. Only in this way can the programmer/coder in all cases be provided with the means to perform simple, quick and efficient translation from the design documents/text files into compilable code.

THE DEVELOPMENT PATH OF PROGRAM GENERATORS

Progress has been achieved incrementally, and this

*Ueberlandstrasse 465, 8051 Zurich, Switzerland. Manuscript received January 1982.

article proceeds in a similar manner. The first necessary step was the realisation that commercial application development involved considerable repetition of effort, and on the other side of the coin, considerable code redundancy. Many functions were coded once per program rather than once per application, or even once for the entire installation. Several facilities have been used to overcome this wastage; for example Copy Libraries and Subprogram Calls remove localised and small-scale redundancies.

In addition to redundancy in processing code there is structural repetition. By this I mean that the majority of program structures are, or could be, formal variants of a set of models. To combat the wastage resulting from structural repetition requires a fundamental reorganisation of applications development, and investment in more effective supporting software.

The term in common use for such software seems to be 'program generator' and that term will be used in this article. Some more precise phrase such as 'parameter driven assembly of high level language programs' would be advantageous, but wordy.

PHASE O – REDEPLOYMENT OF STAFF

The prevailing nonsensical EDP convention of commencing to count at zero is conformed with by harking back to the most primitive, and sometimes the most effective manner of knowledge transfer. Experience in the development of commercial software is exchanged between projects in a planned manner through the assignment of staff with relevant 'know-how'. An even greater amount of experience sharing is achieved in less planned fashion thanks to the velocity of staff within the job market.

This method of knowledge transfer is entirely informal, too heavily reliant on individuals, and unmeasurable. Given the considerable variation between user applications across the various sectors of large and small primary, secondary and services industries, government enterprises and utilities and the public service it is difficult for tertiary courses to provide entrants to the information industry with directly useful applications experience.

Since formal education in such matters is difficult to come by, the interchange of staff between projects and employers will remain an important factor in knowledge transfer in all areas of computer applications. The possibility of formalising the process is greater in the more precise field of programming than in system analysis and design, yet even in this field the first steps were small and tottering.

PHASE 1 – COPY-A-PROGRAM AND AMEND

Plagiarism began with the selection of a program that bore some resemblance to the new one and the copying of the parts that seemed relevant and helpful. The method comprises Figure 1:

- selection of a model program;
- copying to a new file;
- leaving lines unchanged which are common to both programs;
- deleting lines particular to the old program;
- amending lines which are common but which contain terms particular to the program (such as the name of the program and the name of the driving file);
- inserting lines particular to the new one.

This approach can achieve significant gains:

- experience is explicitly transferred;
- it can take less time to prepare the source file;
- it can take less time to achieve a clean program;
- the resulting program is similar in style to its 'father'.

It would be wrong to overlook the inherent problems.

- how is the program selected as suitable for 'fatherhood';
- how correct is 'father' as regards its original task;
- how relevant is 'father' to the new problem. Many mismatches between the two will be subtle, emerging only when testing reveals strange anomalies;
- no relationship is maintained between 'father' and 'son'. Subsequent changes in one are not easily associated with the other.

Nonetheless many organisations have profited from this technique.

PHASE 2 – COPY-A-SKELETON AND AMEND

A step which overcomes many of the deficiencies of Phase 1 is the formalisation of the 'father'. That task can require considerable investment depending on the suitability of the models available, the degree of difficulty of the program type involved, the ambitiousness of the project and the experience and competence of the staff assigned.

The preparation of the skeleton involves the following:

- define the program type to be supported;
- identify those parts of the sample program(s) common to the program type;
- define the variants of the program type which are to be catered for, and which are beyond the scope of that skeleton;
- assemble a 'first-cut' version of the skeleton from the sample(s);
- identify the variables as such. For example the driving file may have been CUST; it might be replaced with DFN (for 'Driving File Name'). In practice it is beneficial to use a string which is not legal in the source language;
- since few programs are direct analogues of one another, build in options which the programmer can select as appropriate. This might for example be

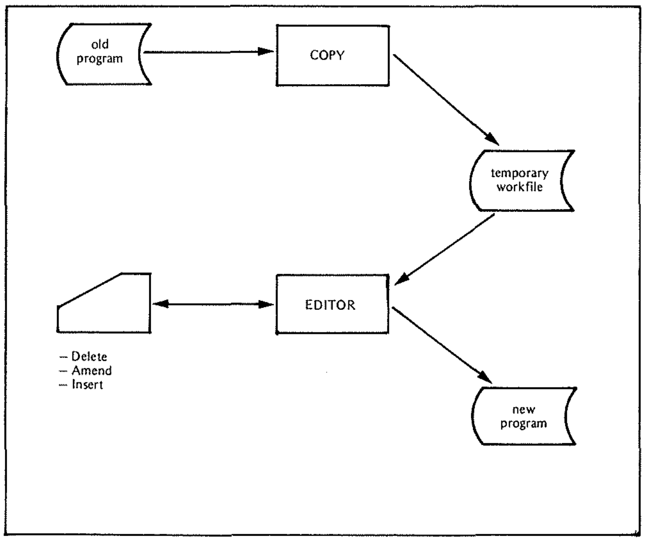


Figure 1. Copy-a-Program-and-Amend.

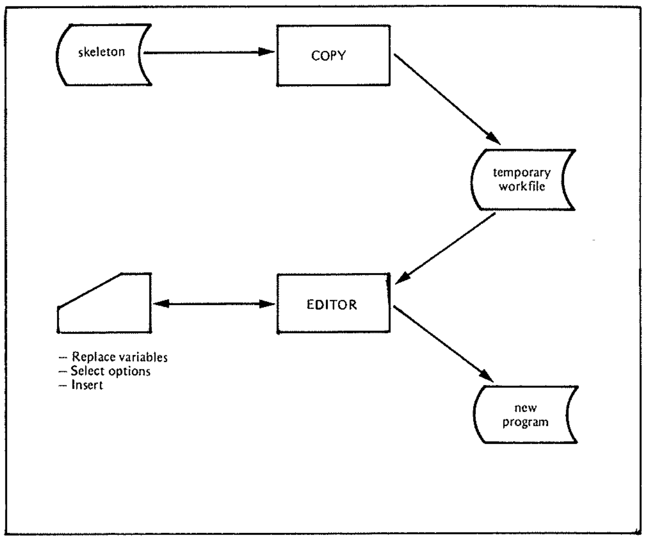


Figure 2. Copy-a-Skeleton-and-Amend.

achieved by the marking of optional lines as active or commented out;

— define the points within the skeleton at which programmers will under particular circumstances need to insert additional code.

The development of a program using such a skeleton comprises Figure 2:

— selection of the appropriate skeleton;
— copying to a new file;
— the replacement of the variables;
— choosing the appropriate options;
— inserting additional lines particular to that program.

The scale of the effort involved varies widely. In the author's experience a file handling sub-program requires about six variables, no additional code, and about five minutes' work. For a reasonably flexible on-line master file maintenance program about 35 variables and 15 options were needed. The number of insertion lines varied directly with the amount and complexity of validation — between 50 and 2000 lines — giving a total development time between two hours and four days. If the average line rate seems high (600 lines/hour for simple programs, 100 for the more difficult ones), it should be recalled that this code is composed almost entirely of editing instructions directly translated from the specifications and containing virtually no control structures.

Advantages of this approach as compared with conventional programming are:

— experience has been invested in the skeleton, and is directly transferred to each program;
— less time is required to prepare the program;
— the new program requires testing only of the program-specific code (assuming that the particular combination of options was tested as part of the skeleton's development), hence less time is required to achieve a clean program;
— the resulting program's style is dictated by the skeleton.

There remain deficiencies:

— the selection of an appropriate skeleton for the task depends on criteria that are rarely fully understood;
— investment in some amount of abstract theorising and experimentation is a precondition of success. Installations which oppose abstraction *per se* and limit their techniques to those taught by their equipment and software suppliers are therefore ill-served by this method. It requires confidence on the part of the installation management that they can manage the risks involved;
— a sufficiently large volume of programs of each type is necessary to justify the investment. In the author's experience a breakpoint was already reached with three or four programs, but that is sensitive to the skeleton builder's experience in and flair for both skeleton building and the program types;
— an on-line development environment is essential, with suitable supporting software, in particular a full-screen editor with string replacement and line insertion capabilities (Clarke, 1982a);
— no continuing relationship exists between the skeleton and the programs produced from it. Subsequent corrections and improvements to the skeleton can only be included in each of its progeny by painstaking effort.

Efforts to overcome this last deficiency lead to the third phase.

PHASE 3 – SIMPLE PROGRAM GENERATORS

Once skeletons have been established it becomes attractive to have the benefit of the maintenance of those skeletons flowing more-or-less automatically to its progeny. The classes of maintenance include error correction (a skeleton is, like any program, 'clean' only until the next bug is found), efficiency improvement, the adaptation of existing facilities to new standards and to new run time environments, and the provision of additional features.

The step required to link programs to their skeleton is to store the instructions used in their preparation, and regard these rather than the generated high level language code as the source program. As Figure 3 depicts, the instructions to be stored comprise the assignment of values to variables, the selection of options and the insertion of additional source lines. A utility program is required to

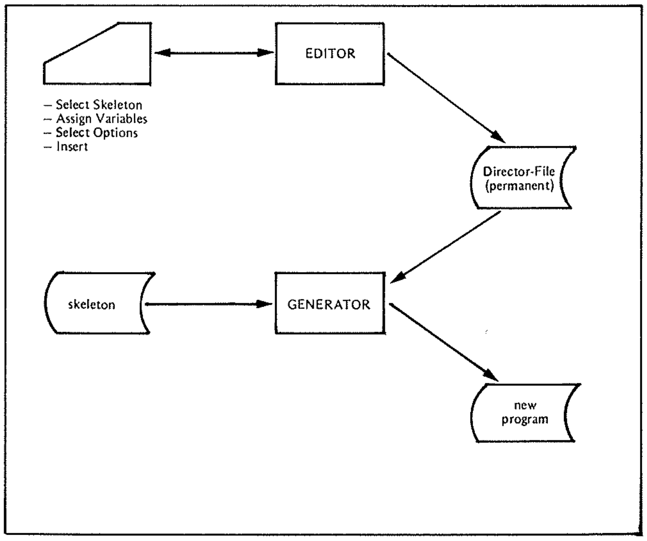


Figure 3. Simple Program-Generator.

merge the skeleton with the additional source lines, carrying out the (global) variable replacement and option setting as it goes. Such a utility is popularly termed a program generator; for the input I will use the term 'director file'.

The development of such a generator requires string handling capabilities. Nonetheless implementation even in COBOL requires under 10 days, and installations with expertise in more suitable languages should require yet less effort. Assuming that a small collection of 3-4 skeletons is created, then the breakeven point will be of the order of only two or three uses per skeleton – a point reached or reachable in almost any single new application.

An additional investment involved is the formalisation of the skeletons. A Phase 2 skeleton can contain loose comments of the form 'if both options A and B are selected, then datafields X and Y must be OCCURed twice, with consequent changes in Procedures P and Q'. This may have been the most cost-effective solution in Phase 2, but cannot be tolerated once a generator is implemented. Such problems are quite soluble, but require disproportionately high investment. (The pragmatic solution is to add this condition to the list of variants unsupported by the generator: 'For Priority Release' as the sales brochures say.)

When a skeleton is revised, all that is necessary to pass the revisions to its progeny is to re-run the generator against the director file. Late amendments in the file handling technique or the user interface no longer justify fears of excessive rework costs and delays.

Some limitations must be recognised of course:

— considerable unanimity is required as to what constitutes good programming style and appropriate program structure;
— the preparation of suitable skeletons requires familiarity with a wide range of program types as well as the ability to abstract;
— machine overhead is incurred by the generation run. The programs require far less testing, but the net effect is hard to measure and correspondingly easy to argue about. On a small software development installation (TI 990 with 256kb memory and five screens) the generator required about three minutes (elapsed) for a 500 line director file and a 1500 line skeleton. This compared favourably with the compile time of the generated program, despite the inefficiencies of COBOL string-handling;
— subsequent amendments to a skeleton must be made with rather more care than with a Phase 2 skeleton. It is necessary to generate and test first that program for which the change is required, then a range of sample programs appropriate to the population of the progeny, then all of the progeny;
— in addition to the normal 'where used' capabilities needed for copyfiles, datafiles and subprograms, the use of the skeletons themselves must be monitored. This is most easily achieved if the invocation of the skeleton is controlled from the director file itself;
— the use of an existing skeleton for a new project often involves additional investment. (Typically the original version assumed only one record type per file, while the new project must handle two or more.) Generally it seems better to allow skeletons to proliferate rather than invest too much too soon chasing the chimera of 'truly general' master programs;
— the method decreases the creativity involved in applications programming. Other sources of programmer job satisfaction must be substituted for that lost if low morale and high turnover aren't to rob the installation of the potential productivity gains;
— a stratification, or at least segmentation, of programming staff results, with differentiated education, experience and even psychological profiles. Given that the existing distinctions between systems and applications programming groups can result in friction, the addition of a 'methods programming' group could be an unwelcome additional ingredient in the political cauldron; and yet most systems programming staff are ill-suited to the work involved because of its strong applications orientation;
— the method invites the naive application of an inadequate tool to a different or more subtle problem. It is essential that in seeking productivity improvement we do not force development staff into under-investment in the problem comprehension and design phases and thereby trivialising their appreciation of the application.

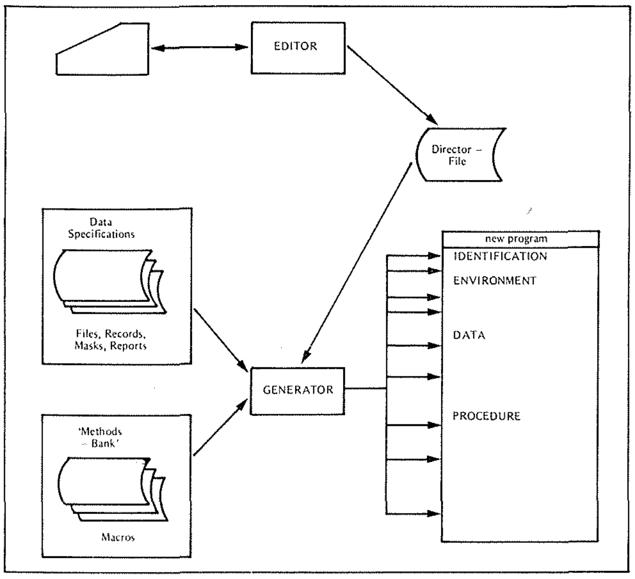


Figure 4. Sophisticated Program-Generator.

PHASE 4 – SOPHISTICATED PROGRAM GENERATORS

The 'merge-and-replace' type of generator remains trapped within the conceptual boundaries of its host language. There are two very important and related limitations that can be overcome only if the framework of the generation run is changed.

The sequential processing of a single skeleton has the result that a skeleton must resemble the program that is to be generated, with the exception that some symbols appear which would not be valid input to a compiler, some denoting locations for insertion, others awaiting replacement: the skeleton and the generated program are synchronous.

The other limitation is that in a family of skeletons there will be a considerable amount of redundancy. In particular, file definition and file access routines will appear not merely in each skeleton, but even several times in each. It is desirable that code which is common to multiple skeletons be stored once only, in an independent sub-skeleton.

The instance of file handling is particularly impor-

```
.PROG-DEMO1, AUTHOR=ROGER
```

generates:

```
000100  IDENTIFICATION DIVISION.                     01000064
000200*************************                      01000065
000300  PROGRAM-ID.                  DEMO1           01000066
000400  AUTHOR.                      ROGER.          01000068
000500/                                              03000003
000600  ENVIRONMENT DIVISION.                        01000081
000700*************************                      01000082
000800  CONFIGURATION SECTION.                       01000083
000900  SOURCE-COMPUTER.      PRIME 550.             01000085
001000  OBJECT-COMPUTER.      SYCOR 585.             01000086
001100  INPUT-OUTPUT SECTION.                        23 DELTA
001200  FILE-CONTROL.                                23 DELTA
001300  I-O-CONTROL.                                 23 DELTA
001400/                                              03000005
001500  DATA DIVISION.                               23 DELTA
001600  FILE SECTION.                                23 DELTA
001700/                                              03000007
001800* ------------------------------------------   23 DELTA
001900  WORKING-STORAGE SECTION.                     23 DELTA
002000/                                              03000028
002100* ------------------------------------------   23 DELTA
002200  PROCEDURE DIVISION.                          23 DELTA
002300  DX-MAIN SECTION.                             23 DELTA
002400  O-PROG.                                      23 DELTA
002500  P-PROG.                                      23 DELTA
002600  C-PROG.                                      02 DELTA
002700  STOP-RUN.                                    02 DELTA
002800       STOP RUN.                               01000131
```

Figure 5. The Minimum-Complexity Program.

```
. PROG-DEMO2, AUTHOR=ROGER
. SL=P-PROG
      DISPLAY "HELLO, USER! WHAT'S YOUR NAME?".
      ACCEPT WS-NAME.
      DISPLAY "CONGRATULATIONS " WS-NAME "!!!".
      DISPLAY "YOUR PROGRAM WORKS ALREADY!".
. SL=WORK01
 01   WS-NAME            PIC X(OB).
```

Figure 6. A Slightly More Complicated Program.

tant, because yet a further level of abstraction exists. In order to facilitiate the portability of applications software between differing machines, compilers, and file handling environments, it is necessary to store those parts of the program which are environment-dependent in separate 'sub-sub-skeletons' which can be exchanged in order to generate a new version of the application to run on, say, an interstate branch's much smaller and perhaps separately sourced installation. (The same problem occurs in relation to the handling of on-line terminals, although defining the interface between the logical and the physical sub-skeletons is made much more difficult by the absence of de facto standards.) See Clarke (1982b, 1982c) for further discussion of such matters.

It is not difficult to restructure the simple generator described in the previous section to include invocations of sub-skeletons, depending on some condition in the director file or the main skeleton. The problem is that the synchronisation between skeleton(s) and generated program is destroyed. In the case of a file handling sub-skeleton, the sub-skeleton will endeavour to insert code in a location (say the file access routines), while the skeleton still contains code that must be inserted at an earlier location.

The requirement is, then, that the director file be read sequentially, resulting in invocations of sub-skeletons, and the 'assembling' of an output file. The output file must

```
ADD TESTMAC, 5, 2, N
```

invokes this Macro:

```
**PDL*8112311159/TESTMAC/02/  TEST-MACRO
. *
. *     Converts an alphanumeric field with contents in the form
. *     '9999.99' into a numeric field of the form 9999V99
. *
. *     The number of digits is freely-choosable.
. *
. *     Parameters: 01 - number of whole-digits
. *                 02 - number of decimal-digits
. *                 03 - whether a subroutine
. *                      is to be created (Y/N)
. *
. * -------------------------------------------------------
. SL=WORK01
 01   WS-ALPHANUM- #01 #02.
        05   WS-AN- #01 #02-WHOLE       PIC 9( #01).
        05   FILLER                     PIC X.
        05   WS-AN- #01 #02-DECIMAL     PIC 9( #02).
 01   WS-NUM- #01 #02.
        05   WS-N- #01 #02-WHOLE        PIC 9( #01).
        05   WS-N- #01 #02-DECIMAL      PIC 9( #02).
 01   WS-NUM- #01 #02REDEF              REDEFINES WS-NUM-
                                        #01 #02 PIC 9( #01)
                                        V9( #02).
. IF-03. EQ. Y
. SL=SUBROUTINES
 CONV-AN- #01 #02.
      MOVE WS-AN- #01 #02-WHOLE     TO WS-N- #01 #02-
                                       WHOLE.
      MOVE WS-AN- #01 #02-DECIMAL TO WS-N- #01 #02-
                                       DECIMAL.
 CONV-AN- #01 #02-EXIT.  EXIT.
**********************************************
. IFEND
. * -------------------------------------------------------
```

to generate this code:

```
WORKING-STORAGE SECTION.
   .
 01   WS-ALPHANUM-52.
        05   WS-AN-52-WHOLE        PIC 9(5).
        05   FILLER                    PIC X.
        05   WS-AN-52-DECIMAL      PIC 9(2).
 01   WS-NUM-52.
        05   WS-N-52-WHOLE         PIC 9(5).
        05   WS-N-52-DECIMAL       PIC 9(2).
 01   WS-NUM-52REDEF               REDEFINES WS-NUM-52
                                       PIC 9(5)V9(2).
```

Figure 7. Macro-Calls.

be addressable at multiple points, not merely at the (current) next record (a partitioned or segmented sequential file as distinct from purely sequential). For flexibility sub-skeletons should be able to be invoked conditionally, and iteration, nesting and even recursion should be possible. In addition parameter passing between different elements must be facilitated. Such a generator is complex, requiring modular construction for reliability, maintainability and extendability, and involving the investment of man-years of effort. Figure 4 depicts such a generator.

Examples of products which offer at least some of the requirements are: CPG, an American product of the late 1970's; CL/1, an Australian product released in 1979, MANTIS from CINCOM (IBM-specific, 1979); NoCode, an American product (1980); and the cutely-named 'The Last One', a UK product (1981). ADR's IDEAL is overdue for release. Philips' PET/X1150 development-machine incorporates generator-elements.

TABLE 1. Properties of Program-Generators.

The Product Provided
- capable of immediate use without initial investment by the user
- based on an interpretative language so as to be portable between software environments and machines
- includes standard macros for common functions which can serve as a starting point for the integration of the product into the user's particular environment
- is consistent with and capable of operation in parallel with other development environments, and in particular with the maintenance of existing applications by conventional methods
- is suitably documented and the documentation is well indexed
- education and introductory documentation are provided
- maintenance and support are provided
- on-going development of the product is guaranteed
- version control and upwards compatibility are assured

The Macro-Language
- standard macros are under user control
- additional macros can be written by the user
- offers DO-verb, and complex conditionals or decision table
- DO-verb and conditionals are nestable to an adequate depth
- offers computational and string handling capabilities
- capable of passing parameters
- parameters may be local or global, and 'typed'
- simple file reading capabilities
- additional locations can be defined
- all locations are accessible by any macro

Program-structure skeletons
- ability to generate the vast majority of program structures with simple parameterised invocations
- all control code for level breaks should be generated
- appropriate locations for insertion of program specific code
- the resulting code should be suitably modular and structured (within the constraints of the generated language)
- ability to specify exotic program structures in a convenient, auditable, powerful but compilable language

Outputs
- generates an industry standard language(s)
- is sufficiently flexible that variants within the standard, not-quite-standard and add-on compiler features can be handled
- generates code that is consistent in style with the prevailing installation standards no matter from which skeletons/macros it may be generated. This is important during the first years following its installation, since maintenance may be performed on the generated programs rather than the original source
- the code generated by all methods is consistent in apppearance
- generates documentation as an integral part of the code
- generates a where-used listing for macros/skeletons
- can generate skeleton JCL for testing and production purposes

Use
- simple to use for simple programs, in particular a simple report generator syntax such that trainees can quickly become productive and experience early positive feedback
- powerful for larger and more complex programs such that the productivity of experienced staff is significantly enhanced
- consistency of use for each type of standard program (i.e. the preparation of simple print programs, simple batch, complex batch, on-line enquiry, data capture and update programs should not differ more than is necessary)
- 'naturalness' of the language and its syntax, rather than obscure mathematical script
- completeness of syntax validation
- clarity of error messages
- accessibility of the documentation for reference purposes
- the capability to reflect user modifications and extensions

Mode of Processing
- can access multiple macros, including many level nesting and perhaps also recursion
- adequately efficient in its usage of machine resources (run time, file access, main storage)
- written in reentrant code and is actually shareable by many users
- capable of concurrent execution by an effectively unlimited number of users, e.g. suitably qualified workfile names, macros accessed in read-only mode
- allows definition of reference libraries and documentation options at run time

Interface to its Environment
- ability to mesh with techniques used within the organisation (structured analysis, structured design, Relational Analysis, HIPO, structograms *a la* Nassi and Schneiderman, decision tables Jackson or Warnier Program Design Methodology, structured programming, etc.)
- interface with Data Dictionary software
- interface with formalised system requirements and system design utilities
- interface with screen definition facilities
- interface with report layout facilities
- interface with project planning and control
- interface with the testing and debugging facilities
- independence from its host machine, i.e. runs on many machines (and in principle on any machine)
- independence from its target machine(s)
- independence from supplier specific environmental software (operating system, file handler/database, languages, on-line monitor, data communications monitor, etc.)

The author has experience of a Swiss product, DELTA (see Clarke, 1982b, 1982c), which fulfils the requirements. It has enjoyed considerable success in German-speaking areas, and is available in both Britain and Australia. It had the market to itself following its market release in 1976, but a small flood of competitors is lining up to do battle. The generator package comprises an interpreter, a set of 'processors' (providing efficient performance of the most common facilities such as the basic program shell, and decision table and pseudo-code interpretation), a range of standard macros (providing file handling, a report generator, etc) and a macro language to enable the writing of further macros.

The distinction between a skeleton and a macro is important. A skeleton contains no control structures; the director file drives the run, but the generator itself performs all the decision-making. In the case of a macro the stored code is not just passive, but contains selection and iteration decisions, based on parameters supplied in the director file, and additional variables computed during the generation run.

This language is available to the software developer, so that he can go further than merely amending existing macros: he can also develop his own to match the requirements of the installation. The following examples of the use of a Phase 4 Generator are based on DELTA, because of the author's familiarity with that product, but also because it embraces all of the important concepts and mechanisms.

EXAMPLES

Figure 5 depicts the preparation of the minimum complexity program. The basic Processor is invoked using the command .PROG; a variety of optional parameters may be set. The result is a program shell containing the minimum set of commands consistent with the particular target compiler. The precise content of the generated shell is determined by macros supplied by the vendor but fully under the using organisation's control.

In addition so-called 'locations' are created into which lines of high level language code can be inserted. Each location is accessible in 'open-extend' mode, i.e. lines

```
. PROG-CUST, AUTHOR ROGER, WRITTEN JUL 81
. SL=REMARKS
*
*        ON-LINE DEMONSTRATION-PROGRAM (CUSTOMER
         FILE-MAINTENANCE)
*        -----------------------------------
*
. *          Create program-structure:
. *
. ADD OLSTRUC, 1, (DSP, CRE, AMD, DEL), –
.                 (MSKCUST1,MSKCUST2)
. *
. * ---------------------------------------
. *
. *           Validation-code:
. *
. SL=VAL-01-DEL
. *
. *          Delete prohibited if current or previous year's
. *          Sales are other then zero:
. *
      IF    T01-SLSYTC = ZERO
      AND T01-SLSYTP = ZERO
          NEXT SENTENCE
      ELSE
. ADD VALERROR, 905, , SLSYTC
. * ---------------------------------------
. *
. *        Define Customer Logical-Record:
. *
. ADD LR-CU, UPDATE-ONPLACE, 1
. *
. * ---------------------------------------
```

The above depends on data definition files (which are the responsibility of the applications team), about 10 standard macros, 5 additional macros written and maintained by the installation standards team, and about 10 macros which generate the program structure and screen handling.

Figure 8. An On-Line Program Using DELTA.

are loaded successively into that slot. The process is directly analogous with a box of 80 column cards in which the permissible insertion points are marked with thick cardboard. Each new card (including new markers) can be inserted immediately before any marker. Figure 5 in itself cleanly compilable, although its execution would cause little excitement. Very slightly more interest would be aroused by the program generated by Figure 6, in which two locations have been used, that for basic processing, and the basic working storage location.

Figure 7 illustrates the next conceptual step, the invocation of macros. Great power can be achieved in the use of pre-written code through the nesting of macros. For example the author uses a single line invocation (together with separately prepared mask definitions) to generate an on-line update program with inquiry, creation, amendment and deletion capabilities, any number of masks and some 30 locations into which the more complex validation and file handling code can be inserted (Figure 8). The additional coding is also strongly supported by additional macros.

A hierarchy of self-supplied macros is one of a range of ways in which the program structure can be generated. A processor is supplied for normal batch processing programs, another generates structures in a manner consistent with Jackson's Program Design Methodology, and an interpreter is available to generate more exotic forms from a structured 'pseudo-code'.

A number of processors are also supplied as part of the basic product to achieve run time efficiency in the handling of certain standard functions. Chief among these is the File Processor which, with the aid of one or more macros generates all code necessary for definition of and access to each file. It also includes facilities for integrating the file processing into the program structure. In COBOL this involves entries into at least the following locations: SELECT, FD, RECORD-DESCRIPTION, OPEN, CLOSE, File handling Subroutines and the calling of the file access routine(s). Macros for the various file types are supplied, and can be used in that form or extended to suit the user's particular requirements.

A further point of importance about Figure 8 is the machine-independence of the DELTA source file. It was written and tested on a PRIME 550, then re-generated on that machine in the form appropriate for a SYCOR (Data 100) Model 585. Differences between the file definition, file handling and (very differently conceived) screen handling methods were catered for with little difficulty. Implementation of precisely that program on further machines involves the preparation of file and screen macros appropriate to the new target machine and/or target environment. Clarke (1982b) discusses this example at greater length.

PROPERTIES OF PROGRAM GENERATORS

Table 1 contains a list of factors to be considered when assessing alternative products or designing one's own. Since this article is tutorial rather than analytical this point is not discussed further.

IMPACT OF PROGRAM GENERATORS

The benefits brought by a sophisticated program generator include the faster development of cleaner products, quicker and more reliable maintenance and enhancement, the opportunity for genuinely portable applications, and shorter lead times for trainees.

The development process, the organisation of development teams, and the organisation and operation of the supporting 'methods programming' team are significantly affected.

TOWARDS APPLICATION GENERATORS

The focus of this article, and indeed of the products which it discusses, is the generation of independent programs. The design of a collection of programs to fulfil a complex of purposes is viewed as a separate exercise. In order to generate an entire application from an application specification, a logically complete and precise statement of the requirements would be needed in a set of consistent and compilable syntaxes. Implementation parameters (e.g. the physical allocation of records and the gathering of functions into programs) would also be required.

CONCLUSION

In the near future only specialist 'methods programmers' will deal at the level of detail of present high level languages. The vast majority of commercial development will be done by programmer coders using utilities to capture the parameters for input to program generators.

In the period 1982-1987 many of these generators will be machine specific, generating a special language code, and be subject to myriad intended and unintended restrictions. Later more of them will achieve substantial machine independence and generate industry standard languages. A very few such second generation products are already on the market.

Somewhat further in the future it seems reasonable to anticipate effective application generators which will operate on one or more system design languages to produce executable code directly.

ACKNOWLEDGEMENT

The assistance of colleagues at Brodmann Software Systeme AG, and of Herrn Peter Buchmann, Claude Reibel and Dr. Reinhold Thurner of SODECON AG, the suppliers of DELTA, in the preparation of this article is gratefully acknowledged.

REFERENCES

ADR (1974a): *Macro-Writing for the MetaCOBOL User*, Doc Nr P502M, Applied Data Research, Princeton, NJ.

ADR (1974b): *Macro-Writing for the MetaCOBOL Specialist*, Doc Nr P551M, Applied Data Research, Princeton, NJ.

CLARKE, R. (1982a): Editors for Software Development, *Aust. Comput. Bull.*, 6, Feb 1982, pp. 21-25.

CLARKE, R. (1982b): Generating Self-Contained On-Line Programs Using DELTA; submitted to the Ninth Australian Computer Conference, Hobart, August 1982.

CLARKE, R. (1982c): Generating Transaction-Oriented On-Line Programs Using DELTA, submitted to the *Aust. Comput. J.*

HAWRYSZKIEWYCZ, I.T. (1981): 'Some Trends in System Design Methodologies', *Aust. Comput. J.*, 13, Feb. 1981, pp. 13-23.

WAITE, W.M. (1974): *Implementing Software for Non-Numeric Applications*, Prentice-Hall, NJ.

BIBLIOGRAPHICAL NOTE

The author has been active in commercial data processing since 1971 in functions ranging from systems analysis and project-leadership through research into the privacy implications of the information industry, to the technology of software development. He has worked for and with a variety of industrial, commercial and consulting organisations, including more recently 1½ years with The Stock Exchange, London, and three years with a software house in Zurich. He completed an M.Comm. (Accounting and Information Systems) in 1975 following nine years' part-time study at the University of New South Wales.

A Software Engineering View of Files

J. L. Keedy* and I. Richards**

The paper takes a fresh look at files, and argues that many benefits can be derived by treating them as information-hiding modules rather than free-standing data structures. A case is made for a hierarchical structure which includes both access routines and semantic routines. A powerful protection mechanism based on semantic routines is discussed.

Keywords and Phrases: Information-hiding, module, modular structure, hierarchical structure, file system, access methods, software engineering, protection, privacy.
CR Categories: 4.34, 4.35.

1. INTRODUCTION

Files (e.g. in commercial data processing systems) are normally regarded as free-standing data structures, in so far as program access and the protection of information are concerned. In this paper we try to show that such a view of files is unsatisfactory from the standpoint of the software engineer, and we propose an alternative view which overcomes the problems inherent in the conventional approach.

The qualities which software engineers aim to achieve in major software systems include reliability, efficiency, adaptability, maintainability, and, especially in the case of files, protection of information. None of these qualities has been achieved with a high degree of success in systems which regard files as free-standing data structures, although the level of success depends on the file system or data base system available. In current systems we find three basic levels of support for files.

Minimal support is provided in primitive operating systems which recognise the existence of files and offers some assistance in organising their placement on disc (or other backing-store devices), but little else. In such systems user programs may have to supply their own routines for organising files internally. No checks exist to ensure that the right set of routines is used when the file is accessed, and no attempt is made to ensure that only authorised users can access the information in a file.

More advanced operating systems provide a file system which includes some standard access methods, such as indexed sequential or hashed random. They also provide checks to ensure that files are only accessed by authorised users and in authorised ways, e.g. by 'read-only', 'read/write', 'append', and similar protection attributes. Usually there is no check that the right access routines are used.

Some data base systems provide a more sophisticated view of files. A distinction may be drawn between 'physical' file structures and a set of 'logical' views of the file or files. This is usually achieved by means of a data dictionary which contains extensive information about the placement, representation and protection attributes of information in the data base, down to the level of fields in records. Such systems show an awareness of the main software engineering objectives mentioned above, but the methods which are required to support them are generally inefficient and costly. Maintenance of the data dictionary itself can be a difficult task and the frequent accesses which are made to it can be costly in terms of processor time and disc channel usage. But the main problem with data base systems is that they are usually monolithic in nature and depend on a central administration of some sort, both within the computer and off-line in the form of a 'data base administrator'. Furthermore, they are usually unsuitable for controlling *all* the data in a system. For example, they rarely handle operating system tables, system or job journal files, source files, spooling files, etc.

In the view of the authors, it seems that a much simpler approach to files is needed than that found in data base systems, an approach which can be used uniformly for all files and which meets the requirements of a software engineering environment. In terms of *reliability*, for example, it must be guaranteed that a file is accessed only via routines which understand its internal organisation. The *efficiency* requirement implies that it should be easy to change the internal structure of a file from an existing access method to a more optimal one, without having to modify the programs which use the file. There is a need also to *adapt* the file for uses other than those originally envisaged, for example, because it might hold (or need to hold) information useful to different parts of an organisation. Files, like programs, are frquently subject to change in the costly *maintenance* phase of a software system, and this implies that the maintenance programmer should easily be able to understand the implications of proposed changes. And finally, protection of the information in a file should be both flexible (i.e. expressable and controllable in terms of the user's *real* protection needs) and at the same time complete (i.e. secure against fools and against malicious users). In the rest of this paper we develop an alternative approach to file management which attempts to meet these objectives, and we outline how this might be implemented in an efficient way.

2. THE INFORMATION-HIDING PRINCIPLE

The basic proposal is that files should not be treated as free-standing data structures. Instead they should be regarded as modules organised according to the information-hiding principle. This structuring principle, which aims to encapsulate information about the design and implementation of major data structures and algorithms in individual modules rather than allow it to

†Department of Computer Science, Monash University, Clayton, Victoria 3168. ††Department of Computer Science, Melbourne University, Parkville, Victoria 3052. *Manuscript received January 1982.*

appear at module interfaces, was developed largely in response to the many problems experienced by designers of major operating systems and other complex software systems in the 1960s. It was common practice at that time to decompose systems into modules in such a way that major data structures of the system (e.g. peripheral tables, job queues) were free-standing, in the sense that they were directly accessed by many different modules of the system. In consequence, detailed information about their contents and representation had to be known to many programmers and programmed into many modules, just as in current day application systems 'information' about the contents and representation of files is reflected in the application programs. This led to at least the following problems:

(i) Programmers had to communicate with each other extensively about the data interfaces to their programs (Parnas, 1971).
(ii) Duplication of effort and wastage of space occurred because each program using a structure had to have its own 'access' routines, e.g. tree traversal routines if the structure was organised as a tree.
(iii) Even a minor change to a data structure involved finding and modifying all the modules using the structure.
(iv) There was considerable risk that a module might be overlooked when a change was made — which largely created the syndrome of frequent operating system releases with new bugs.
(v) There was a tendency not to make changes because of the extensive effort and risks involved.
(vi) Access to data structures had to be synchronised correctly, which was not easy with the many modules involved.

The basic solution to these problems, which are not dissimilar to those encountered in application systems which treat files as free-standing structures, was to hide the detailed information about data structures behind entirely procedural interfaces, i.e. to regard a structure and its access routines as a single module. Other modules requiring access to information in the structure would do so by calling these access procedures rather than by directly reading from or writing into the structure. Thus less communication between programmers was necessary (because a procedural interface contains less information than the detail required to describe a complex data structure), there was less duplication of effort and waste of space (because the access routines were written once only), changes to data representations to achieve greater efficiency were localised within a single module, and synchronisation was greatly simplified.

The information-hiding principle, if used properly, hides not only data structures but also algorithms (e.g. the user of a module which sorts a data structure can be unaware of the sorting algorithm used and his programs are unaffected if the algorithm is changed) and it can be used to hide other details such as lower abstract machines, including real hardware systems (Rosenberg and Keedy, 1978). It applies equally to the decomposition of user modules (Parnas, 1972b) and operating systems (Keedy, 1978). It is, of course, still important to specify interfaces (Parnas, 1972a), but since these are expressed entirely in procedural terms it is easier to formalise the specifications using techniques which can be understood by programmers (Keedy, 1979).

The technique can also be used hierarchically. For example, typical operating systems maintain many queues. A queue access module might be defined along the following lines:

```
    type queue;
    procedure enqueue (x : item);
        [This procedure inserts item x at the tail of the
        queue]
    Procedure dequeue (var y : item);
        [This procedure removes the item at the head
        of the queue, and returns its value to the caller
        as item y]
    procedure qlength (var z : integer);
        [This procedure returns the integer value z,
        which is a count of items on the queue]
```

This specification does not define how the queue is organised. It could be an array, a linked list, etc. (Error conditions, the maximum queue length, etc., have been omitted for simplicity.) Such a module, represented pictorially in Figure 1, would be useful in many parts of the system. For example, it could be used by a process scheduler to maintain a queue of processes. The process scheduler itself would be defined as an information-hiding module with a suitable set of operations for controlling processes, as the following simplified definition illustrates:

```
    module process scheduler;
    procedure create-process (var p : integer);
        [This procedure creates a new process and re-
        turns its identifier in the parameter p]
    procedure start-process (p : integer; a : address);
        [This procedure causes process p to start execu-
        ting at address a]
    procedure delay-process (p : integer);
        [This procedure temporarily halts process p]
    procedure resume-process (p : integer);
        [This procedure resumes execution of the
        delayed process p]
    procedure delete-process (p : integer);
        [This procedure destroys process p].
```

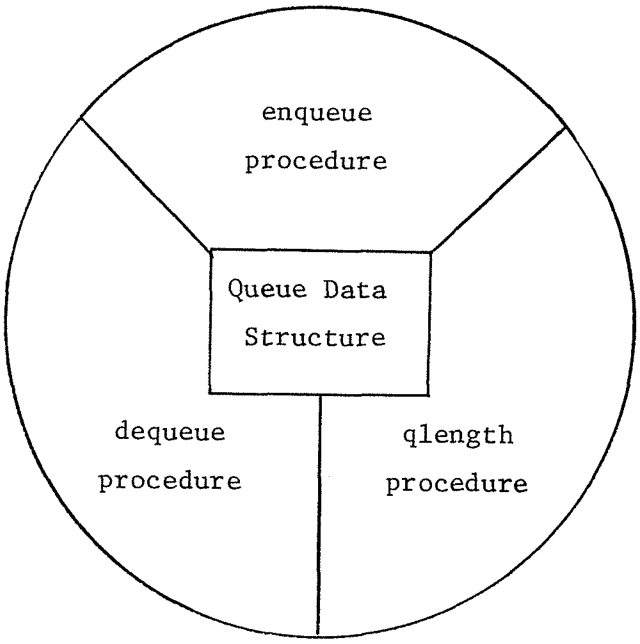


Figure 1. An Information-hiding Module to Organise a Queue.

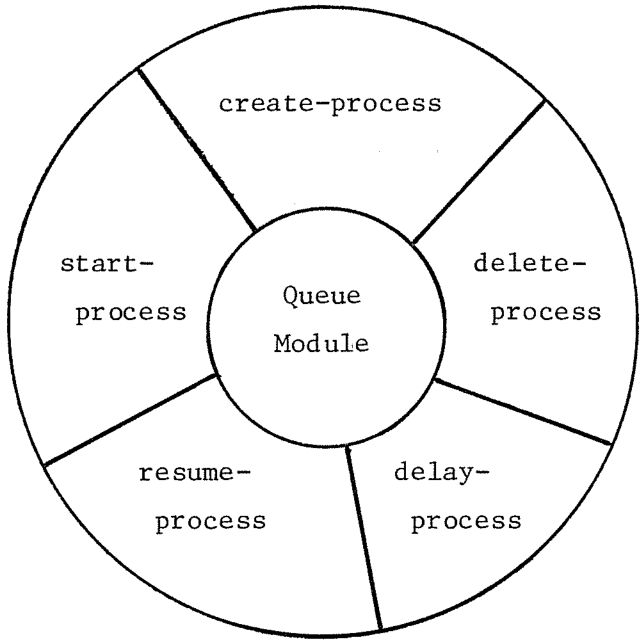


Figure 2. A Process Scheduler Module using a Queue Module.

Other parts of the operating system could use this module to control their processes without needing any information about how the module works — in principle without even knowing that the process scheduler needs queues. In practice the process scheduler would use the queue module to organise its information (Figure 2). At the same time other system modules needing queues could make use of the queue module for their own purposes. Ideally this would be arranged by using the same reentrant code in each case, but with separate instances of queue data structures. Notice that this hierarchical organisation of modules naturally leads to a separation of *access* modules which understand how data structure are organised but not the use to which they are put, and *semantic* modules which provide meaningful operations on the data but are not concerned about its detailed organisation.

The information-hiding principle is closely related to two other concepts. The first of these is the data abstraction technique found in several research programming languages (Wulf, London and Shaw, 1976; Liskov, Snyder, Atkinson and Schaffert, 1977), and more recently in Ada (Ichbiah, Barnes, Heliard, Krieg-Brueckner, Roubine and Wichmann, 1979); it derives from the class construct in Simula (Dahl, Nyhrhaug and Nygaard, 1968). The second is the 'object model', which relates objects in computer systems to the operations associated with them (Jones, 1978).

3. FILES WITH ACCESS ROUTINES

There is some commonality between the application of the information-hiding principle and the use of file access methods in many existing systems, but the latter often do not keep rigorously to the principle, nor were they developed with most of the previously mentioned aims in mind, except to avoid duplication of effort. On the other hand, the information-hiding principle (like the idea of data abstraction) has usually been applied only to temporary data structures in the computational memory. Nevertheless, it is clear that both ideas can be unified without great difficulty. The interface definition for a conventional random access module might be summarised as follows:

```
type random;
        procedure create  (..........);
        procedure open  (...........);
        procedure close  (...........);
        procedure insert  (..........);
        procedure delete  (..........);
        procedure modify (..........);
        procedure retrieve  (........);
```

The parameters to these procedures would identify the appropriate file and records within the file, etc.

As it stands the definition conforms to the information-hiding principle. It would not do so if the user program could also access information within the module, e.g. a file control block or file definition table. Also, problems would arise (a) if the module could be used to access files not created by it; or (b) if other programs or modules could access files created by it. Ideally, a mechanism should exist to ensure that these routines only and always access files created by the create procedure. Such a mechanism is usually absent in existing systems.

The same reentrant routines could, of course, be used to access all random files. In conventional systems we tend to visualise this as shown in Figure 3. However, this is really a reflection on the poor code-sharing facilities provided in most computers. A more useful visualisation is shown in Figure 4. The reasons for this will become evident as we proceed. At this point suffice it to say that each *file module* can more easily be considered to have a single logically separate identity, which embraces both the data and the access method (even though the code of the access routines might physically be shared).

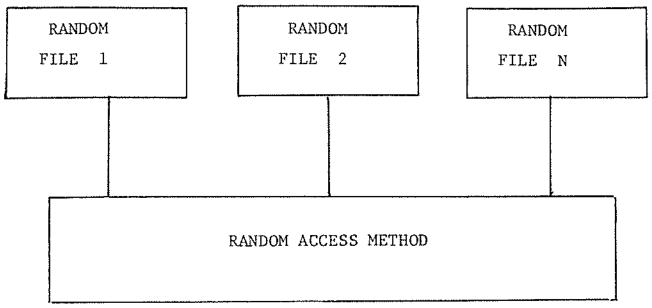


Figure 3. The Conventional View of a File Access Module.

4. FILES WITH COMPLEX INTERNAL STRUCTURES

Some file access methods have a more complex structure than that described in the previous section. The paradigm for this is an indexed sequential file which has an

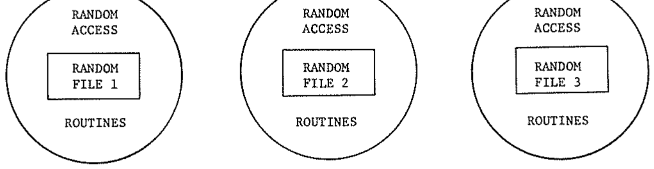


Figure 4. An Information-hiding View of File Access Routines.

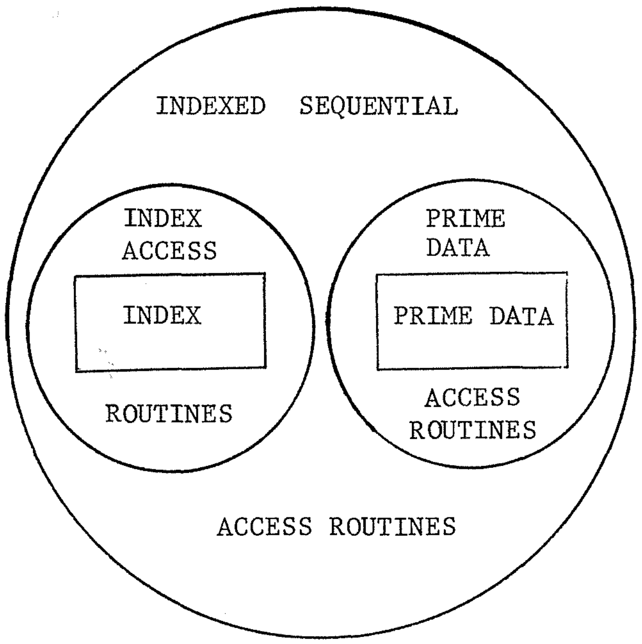


Figure 5. Information-hiding for the Internal Structures of a Complex File.

index, a prime data area, and possibly an overflow area (which we ignore for the sake of simplicity). A program which uses such a file should not need to know about this complex internal structure. Instead it will expect to have an interface similar to, perhaps a superset of, that described for random files in the previous section. On the other hand the existence of different major data structures internally suggests that each such structure should have its own access routines. This leads to the hierarchical structure shown in Figure 5. From the software engineering viewpoint this has the advantages of good structure, in particular that a change to one structure, say the index, is localised within a small module. Given a suitable computer architecture it might also have the further advantage that one group of routines — say the index access routines — could also be shared independently, perhaps by some other access method that needs an index but organises its prime data differently. Again, it is essential that only the right routines can be used with the right data.

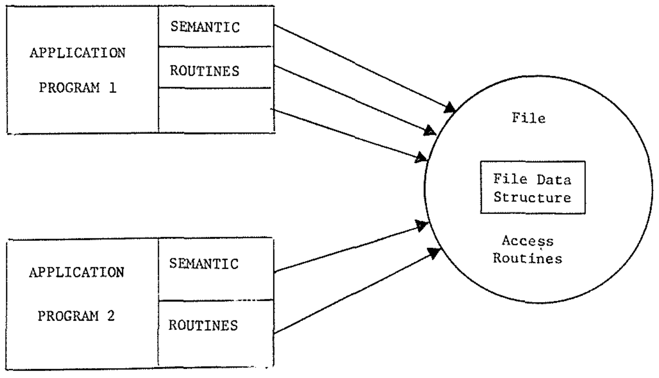


Figure 6. Semantic Routines in the Application Programs.

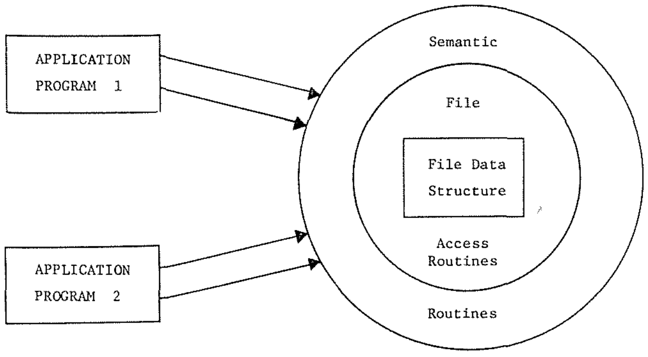


Figure 7. Semantic Routines associated with the File.

5. SEMANTIC ROUTINES

In conventional systems a knowledge of the meaning of the data held in a file is usually programmed into the application programs. For example, if the file holds information about bank accounts the application programs, if well-structured, will contain subroutines to perform operations such as 'open an account', 'make deposit', 'make withdrawal', 'add interest', 'authorise overdraft', 'read account history', 'read current balance', 'close account'. We refer to such subroutines as 'semantic routines', and it is these routines which call the file access routines (Figure 6).

In some cases we may find that the same semantic operation is needed in several programs. For example, 'read current balance' and 'close account' might appear both in a program used by bank tellers and in a program used by head office auditors. This situation introduces similar problems to those discussed in relation to access routines, e.g. duplication of programming effort, wastage of memory space, and more difficult maintenance. The latter arises, for example, if the fields in records of the file are changed. These problems can be solved in the same way, i.e. by detaching the semantic operations from application programs and associating them in a hierarchical fashion with the file itself (Figure 7). In this way the application programs become simpler and more intelligible, while the file module becomes a more meaningful abstract object. But the main advantage of this approach becomes clear when we consider the question of information protection.

6. PROTECTION OF INFORMATION

Current methods of file protection are based on cumbersome software mechanisms which usually treat files as free-standing structures and therefore can only control access in terms of operations such as 'read-only', 'read/write', 'append', etc. In other words, access rights bear no relationship to the semantics of the file usage. Some data base systems can improve on this, but only at the expense of even more cumbersome software.

If, however, we could develop a mechanism which sees protection in terms of permission to use (i.e. call) particular semantic operations (as outlined in the previous section), a much finer grain of protection would be achieved without the intervention of cumbersome monolithic software.

The basis of such a protection scheme would be the presentation of a 'capability' to the call mechanism of the computer, when the application program calls a semantic

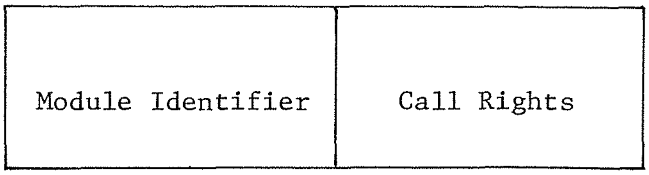


Figure 8. A Module Capability.

routine. The capability would consist of two parts, a unique module identifier and a set of access rights to indicate which procedures of the module could be called (Figure 8). These call rights could be represented by a bit string, with each bit corresponding to one of the semantic routines of the module, or as an integer representing a defined set of procedures (Bishop, 1977). The capability itself would need to be protected in one of the usual ways, i.e. by tagging or by partitioning (Fabry, 1974), and would be held in a directory equivalent to a file directory in existing systems. An example showing how semantic operations on files can be protected using capabilities implemented by a different but similar technique, used in the Hydra system, has been described by Wulf et al. (1974).

The power of our mechanism can be illustrated using the example of the bank account file described in the previous section. We can imagine that various bank employees would need the right to access bank accounts, but in different ways. Figure 9 illustrates how they might be constrained to use only the appropriate semantic routines. The ticks in each column represent the procedures which can be called by a particular employee.

It now becomes clear why it is more convenient to view the file and its associated routines as a single module with a single identifier. In this example we could envisage that many such bank account files might exist, typically one per bank branch of a major banking organisation. The employees at a particular branch (the first three columns) would each be given an appropriate capability only for their own branch's file, not for all files, so that a capability simply to call the procedures without identifying the file instance would be unsatisfactory.

In the proposed scheme the module identifier must contain enough information to enable the system to identify both the file data segments and the associated procedures. This can be achieved efficiently by the identifier defining the data only, and by maintaining with the data a pointer to the code segments of the module.

Semantic Routines \ Bank Agents	Bank Teller	Branch Accountant	Branch Manager	H.O. Auditor	H.O. Accounts
open account		✓	✓		
deposit	✓	✓	✓		
withdraw	✓	✓	✓		
add interest					✓
authorise overdraft			✓		
read account history		✓	✓	✓	✓
read current balance	✓	✓	✓	✓	✓
close account	✓	✓	✓	✓	

Figure 9. Access Rights based on Semantic Routines.

One question which arises relates to the size of a file. In the bank account example, it would be possible to treat each customer's account as a separate file, to treat all the accounts at a particular branch as a file, or to treat all accounts from all branches as a single file. Various trade-offs must be considered. The proliferation of many small (e.g. single account) files would lead to space management problems and to a proliferation of capabilities held in each user's directory. On the other hand, a single file for all accounts at all branches would mean that all bank employees would have some access to all accounts, e.g. at other branches, which would also be unsatisfactory from the protection viewpoint. Thus, in this case a reasonable compromise would be to maintain one account file per branch, which minimises space management problems and the proliferation of capabilities, but which provides what appears to be a reasonable level of security.

A potential problem with identifying a file and its code in a single identifier occurs if we wish to call a procedure which simultaneously handles two or more files, e.g. to merge or compare them. However, this is easily overcome by providing a mechanism which allows capabilities to be used not only as call destinations but also as parameters to such calls. In such a case information-hiding can be guaranteed if the mechanism checks that such parameters are passed only to modules of the same code type.

7. BENEFITS OF UNIFORMITY

The approach to files which we have described has a further advantage. Since files are now regarded as modules with code entrypoints they take on an external appearance which is identical to all other major modules of software in an information-hiding system. Application programs can be regarded as modules with (usually) a single interface procedure. Program modules appear as modules with multiple entrypoints, as also do operating system modules, subroutine libraries and data abstractions at the programming level. This means that uniform mechanisms can be used to catalogue them, to call them, to protect them, and to synchronise them.

Such uniformity of mechanism is greatly enhanced if the conventional distinction between filestore and computational memory is abandoned in favour of a single homogeneous virtual memory which contains all modules and their data, whether temporary or permanent. The feasibility of such a memory organisation has already been demonstrated by Multics (Organick, 1972) and other research systems, but more notably in the IBM System/38 (Houdek and Mitchell, 1978). None of these systems, however, enforces information-hiding as described above.

8. FINAL REMARKS

An objection often raised against the scheme described above, and against other schemes which rely heavily on procedure calls to create good structure, is that the call overheads are too high. Consequently there is a temptation to relax the information-hiding principle, for example by allowing variables as well as procedures to be 'exported' from modules. In our view this temptation should be vigorously resisted, and other methods should be used to avoid such overheads (Keedy, 1980a). It should be remembered also that most modern computers now provide extensive hardware support for procedure calls.

The first author and his colleagues are currently developing two computer systems which will efficiently implement the ideas described in this paper. The first, MONADS II, is based on a HP2100 16 bit minicomputer which has been extensively modified to provide a homogeneous paged virtual memory with 2^{16} address spaces each 2^{16} bytes maximum in length (Abramson, 1981; Rosenberg and Keedy, 1981), and capability-based addressing for on-stack and off-stack data as well as capability-based module calling. Because of the limited addressing range, and for other reasons also, this is regarded as a pilot system only and a further processor, MONADS III, is now in the design stage and will be built from standard components, including bit-slice chips. The homogeneous virtual memory, based on the model described in (Keedy, 1980b), will have an addressing range of the order 2^{28} x 2^{28} nibbles (4-bit units) with a 32-bit word size. The microcode of both systems and the operating system (portable between the two) will together provide full support for the ideas described in this paper.

It should be noted, however, that while new hardware and operating systems will simplify the application of the principles and techniques described in this paper, many of the benefits can be gained using existing tools. For example, if the system designer is constrained to use COBOL and a conventional operating system, it is possible to design a system along the lines described above, using existing file access methods and defining semantic routines as part of the file design phase. Individual application programs would be designed to use the semantic routines already defined, but then the COPY facility of COBOL would be used to insert copies of the semantic routines into these programs as necessary. Although full protection could not be achieved in this way (unless some control could be exercised over the use of the COPY facility), the remaining software engineering aims could at least partially be fulfilled.

9. ACKNOWLEDGEMENTS

We would like to thank Tony Montgomery for suggesting improvements on an earlier draft of this paper. Thanks are also due to members of the MONADS project for many conversations on the subject of the paper, especially to David Abramson, John Rosenberg, and David Rowe, who are heavily involved in the design and implementation of the MONADS II and MONADS III systems.

We also thank the Australian Research Grants Committee for software and hardware grants (F77/15337 and F80/15191), and the Monash Special Research Grants Committee for grant SC18/79.

10. REFERENCES

ABRAMSON, D. (1981), Hardware Management of a Large Virtual Memory. *Proc. 4th Australian Computer Science Conference,* Brisbane, pp. 1-13.

BISHOP, P.B. (1977), Computer Systems with a Very Large Address Space and Garbage Collection. Ph.D. Thesis, MIT, May 1977.

DAHL, O.-J., MYHRHAUG, B. and NYGAAD, K. (1968), The Simula 67 Common Base Language. Norwegian Computer Centre, Oslo.

FABRY, R.S. (1974), Capability-based Addressing. *Comm. ACM,* Vol. 17, No. 7, pp. 408-412.

HOUDEK, M.E. and MITCHELL, G.R. (1978), Translating a Large Virtual Address. IBM System/38 Technical Developments, IBM Corporation, pp. 19-21.

ICHBIAH, J.D., BARNES, J.G.P., HELIARD, J.C., KRIEG-BRUECKNER, B., ROUBINE, O. and WICHMANN, B.A. (1979), Preliminary ADA Reference Manual, and Rationale for the Design of the ADA Programming Language. *ACM Sigplan Notices,* Vol. 14, No. 6, Parts A and B.

JONES, A.K. (1978), The Object Model, a Conceptual Tool for Structuring Software. In: *Operating Systems, An Advanced Course,* Ed. R. Bayer, R.M. Graham and G. Seegmuller, Lecture Notes in Computer Science, Vol. 60, Springer Verlag, Berlin, Heidelberg, New York, pp. 7-16.

KEEDY, J.L. (1978), The MONADS Operating System. *Proc. 8th Australian Computer Conference,* Canberra, pp. 903-910.

KEEDY, J.L. (1979), On the Specification of Software Subsystems, *Aust. Comput. J.,* Vol. 11, No. 4, pp. 127-132.

KEEDY, J.L. (1980a), On the Exportation of Variables, *Aust. Comput. J.,* Vol. 12, No. 1, pp. 23-27.

KEEDY, J.L. (1980b), Paging and Small Segments: A Memory Management Model. *Proc. 8th World Computer Conference,* North-Holland, pp. 337-342.

LISKOV, B., SNYDER, A., ATKINSON, R. and SCHAFFERT, C. (1977), Abstraction Mechanisms in CLU. *Comm. ACM,* Vol. 20, No. 8, pp. 564-576.

ORGANICK, E.I. (1972), *The Multics System: An Examination of its Structure,* MIT Press, Cambridge, Mass.

PARNAS, D.L. (1971), Information Distribution Aspects of Design Methodology, *Proc. 5th World Computer Conference,* North-Holland, pp. 339-344.

PARNAS, D.L. (1972a), A Technique for Software Module Specification with Examples. *Comm. ACM,* Vol. 15, No. 5, pp. 330-336.

PARNAS, D.L. (1972b), On the Criteria to be Used in Decomposing Systems into Modules, *Comm. ACM,* Vol. 15, No. 12, pp. 1053-1058.

ROSENBERG, J. and KEEDY, J.L. (1978), The MONADS Hardware Kernel, *Proc. 8th Australian Computer Conference,* Canberra, pp. 1542-1552.

ROSENBERG, J. and KEEDY, J.L. (1981), Software Management of a Large Virtual Memory, *Proc. 4th Australian Computer Science Conference,* Brisbane, pp. 173-181.

WULF, W., LONDON, R. and SHAW, M. (1976), An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 4, pp. 253-264.

WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., and POLLACK, F. (1974), HYDRA: The Kernel of a Multiprogramming Operating System, *Comm. ACM,* Vol. 17, No. 6, pp. 337-345.

The Input Space Model for Software Testing

N. Parkin*

The input space model of software testing as propounded by Cho is reviewed. The limitations and potential of the model are investigated. The work of Cho is extended to testing of languages defined using Backus-Naurform (BNF). An example of a symbolic input attribute decomposition (SIAD) tree for a PASCAL subset is included to illustrate the method of extension.

Keywords and Phrases: software testing, input space models, statistical quality control, software failure, reliability.

CR Categories: 4.12, 4.2, 4.6.

1. INTRODUCTION

As Dijkstra (1972) has remarked "Testing shows the presence of bugs not their absence". Clearly there is something profoundly worrying about this statement, particularly as software is taking on more and more critical roles such as patient monitoring, nuclear early warning, space craft guidance, etc. Yet still the only practical way to find out whether software is performing to specifications is to test it. (Of course, very worthwhile and interesting work is going on in the area of program proving, both manual and automatic, but we still cannot apply these techniques to large practical projects.)

A major problem with testing arises when we apply the technique to large practical systems. The input domain for such a system, i.e. the set of all valid inputs, is usually very large and very often it is infinite. This means that we cannot exhaustively test the system. Furthermore, even if we could process all of the inputs there would be severe problems in examining every one of the corresponding outputs produced. We need some kind of sampling technique which will enable us to look at a finite subset of inputs and infer the reliability properties of the software. Cho (1980) has proposed a technique for the generation of random samples from the input space and has applied it to the testing of FORTRAN and COBOL compilers and to certain application programs.

In this paper we extend the technique to languages which are defined in BNF.

2. THE INPUT SPACE MODEL AND QUALITY CONTROL

Kopetz (1980) has proposed the following model of software. It is influenced by the work of a number of people including Denning (1971), Dijkstra (1972), Goos (1973), and Horning and Randell (1973).

A program is an ordered set of statements

$$\{S_1, S_2, \ldots, S_n\}$$

The execution of a single statement is called an *action.* Kopetz considers a number of aspects of the model. We are interested here in the transformational aspect. In the transformational model each action is considered as a function. The function operates on variables $x_1, x_2, \ldots, x_n$ which acquire input values from domains $D_1, D_2, \ldots, D_n$ and results from ranges $R_1, R_2, \ldots, R_n$. The *input space* is the Cartesian product of the D_i, $i = 1, \ldots, n$. The *output space* is the Cartesian product of the R_i, $i = 1, \ldots, n$. The variables are called the *input variables* and *output variables* of the action.

In order to apply the model to a whole process rather than an action the following definitions are introduced. (It is assumed that the process is embedded in an environment.) A *changed variable* of the process is any variable which is an output variable to any one action. A *significant variable* of the process is any variable which is an input variable to any one action. An *input variable* of the process is every variable that is a significant variable of the process and a changed variable of the environment. An *internal variable* of the process consists of all variables that are used in any one action of the process and are neither input nor output variables of the process.

The transformational input space model is obtained by ignoring the internal variables of the process and dealing only with the input variables, the output variables and the data transformation of the process.

Cho (1980) takes the input space model and applies traditional techniques of statistical quality control. In this approach a software system is considered as analogous to an industrial process which produces an infinite number of outputs. Each output is considered to be a product unit. The major advantage of the method is that *numerical* confidence levels can be claimed for software quality after testing. The testing procedure described by Cho involves six steps:

1. Define the product unit.
2. Define the product unit defectiveness.
3. Determine a sampling plan.
4. Construct test data by a random procedure using a SIAD tree.
5. Analyse the test results.
6. Perform statistical inference on the test results.

*Department of Computer Science, Massey University, Palmerston North, New Zealand. Manuscript received December 1981.

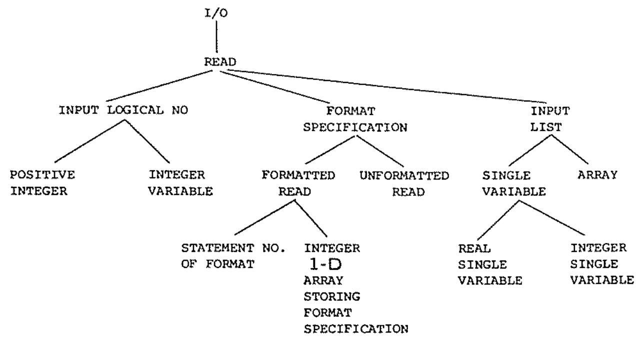
A sampling plan is organised using traditional quality control techniques. The inputs to the sampling plan if single sampling techniques are used are $\theta_1, \alpha_1, \theta_2, \alpha_2$, where α_1 is the producer's risk at defective rate of θ_1 and α_2 is the user's risk at defective rating of θ_2. Note that α_1, the producer's risk is the probability of having a good software unit rejected whereas α_2, the user's risk is the probability of accepting a bad software unit.

The above four parameters must be agreed between the producer and the user before the testing starts. At present there is little case history so agreement may be difficult. However as the technique becomes established this difficulty should go away or at least become less of a problem.

The SIAD (or symbolic input attribute decomposition) tree is a representation of the input data of the program. The relationships between the data components is shown by the tree. In Cho's treatment numerical data elements have tree elements indicating the lower and upper bounds. The provision of lower and upper bounds is of course applicable to any sequenced data structure so that this concept can be extended to other data types in a heavily typed language such as PASCAL. The technique is illustrated in the next section by a FORTRAN example.

3. THE SIAD TREE FOR FORTRAN

The following example, which is based on a larger example in Cho (1980), shows how the SIAD tree for a FORTRAN compiler is obtained and used to select random language elements.



A linear ordering of the tree elements is obtained and this is used to give each tree element an index number as shown in Table 1. (Cho uses the pre-order traversal to give the ordering shown.)

TABLE 1.

Index	Ref	Element
1	X1	I/O
2	X11	READ
3	X111	INPUT LOGICAL NO.
4	X1111	POSITIVE INTEGER
5	X1112	INTEGER VARIABLE
6	X112	FORMAT SPECIFICATION
7	X1121	FORMATTED READ
8	X11211	STATEMENT NO. OF FORMAT
9	X11212	INTEGER 1-D ARRAY STORING FORMAT SPECIFICATION
10	X1122	UNFORMATTED READ
11	X113	INPUT LIST
12	X1131	SINGLE VARIABLE
13	X11311	REAL SINGLE VARIABLE
14	X11312	INTEGER SINGLE VARIABLE
15	X1132	ARRAY

Tree elements are then selected using a (well-tested!) pseudo-random number generator. A given element once selected then has a statement built around it. (This is currently done manually.) If in the example, index 14 were generated, the corresponding statement(s) might be

```
      READ (5,1) N
1     FORMAT (1X, I/O)
```

If index 13 were also generated the corresponding statement(s) might be

```
      READ (5,2) Y
2     FORMAT (1X, F10.2)
```

The test program might then be

```
      READ (5,1) N
1     FORMAT (1X, I10)
      READ (5,2) Y
2     FORMAT (1X, F10.2)
      WRITE (6,1) N
      WRITE (6,2) Y
      STOP
      END
```

Note that additional statements had to be added to make the program acceptable. The additional statements for FORTRAN are usually DIMENSION, WRITE, STOP, END, etc. Note also that some elements occur several times in the tree so that the predecessors of the element are needed when producing the appropriate statement.

After the test programs have been run, the results are examined and a simple accept/reject decision is taken on each program. This allows the use of standard industrial quality control tests and experimental designs.

4. EXTENSION TO A BNF DEFINED LANGUAGE

One advantage of languages such as Algol 60, PASCAL, etc. is that their syntax is defined in terms of BNF. This section shows how we can use a BNF description of a small PASCAL subset to generate a SIAD tree for use in quality control measurements.

The small subset shown in Table 2 is chosen to keep the size of the problem within bounds. It illustrates most of the problems involved in using the BNF definition to produce a SIAD tree. The subset is taken from Calingaert (1979) with slight amendments. Instead of recursive productions of the form

stmtlist = stmt | stmt ";" stmtlist

we use the alternative

stmtlist = stmt {; stmt}

where components enclosed in { } are repeated zero or more times. The aim is to generate random components of the language and then to compose them into a plausible program. (We notice here a possible flaw in the logic of the process. The components are randomly generated but the program must combine the statements into a reasonable sequence.)

Besides recursion, another problem in the language subset considered is that of type and declarations. The approach adopted is to omit type and declaration from the

TABLE 2. Definition of a PASCAL Subset

```
program       = "VAR" decllist ";" cmpdstmt "."
decllist      = declaration | declaration ";" decllist
declaration   = IDENTIFIER ":" type
type          = "BOOLEAN" | "CHAR" | "INTEGER" | "REAL"
cmpdstmt      = "BEGIN" stmtlist "END"
stmtlist      = stmt | stmt";"stmtlist
stmt          = simplestmt | structstmt
simplestmt    = assignstmt | iostmt
assignstmt    = IDENTIFIER ":=" expression
expression    = expression "+" term | term
term          = term "*" factor | factor
factor        = "(" expression ")" | IDENTIFIER
iostmt        = ("READ" | "WRITE") "(" IDENTIFIER ")"
structstmt    = cmpdstmt | ifstmt | whilestmt
ifstmt        = "IF" condition "THEN" stmt ["ELSE" stmt]
whilestmt     = "WHILE" condition "DO" stmt
condition     = expression RELATION expression

                    (a)  Syntactic rules

IDENTIFIER    = letter{ letter|digit }
letter        = "A" | "B" | . . | "Z"
digit         = "O" | "1" | . . | "9"
RELATION      = "<" | "≤" | "=" | "≠" | "≥" | ">"

                    (b)  Lexical rules
```

SIAD tree proper. Each time an identifier is introduced its type is generated randomly from the types indicated. Declarations are then produced afterwards so as to match the types which have been randomly generated.

The SIAD tree for the PASCAL subset is shown in Table 3.

The SIAD tree for a given set of BNF productions is not unique. Arbitrary decisions may have to be taken in order to keep the tree within reasonable bounds. The SIAD tree produced for the Pascal subset was obtained by following closely the approach used by Cho (1980). A refinement of the approach would take account of the expected frequencies of occurrence of language elements.

5. DISCUSSION

Musa (1980) is not entirely convinced of the utility of the input set approach. He cites two deficiencies of the approach:

1. The large number of possible input sets for any useful program makes it impractical.
2. The proportion of input sets that execute successfully is not particularly meaningful to software engineers; MTTF (mean time to failure) is more useful since it is related to costs and other impacts of failure and since it is compatible with standard reliability theory.

Musa does consider the input space approach as a valuable concept and feels that it may be useful in understanding the so-called 'testing compression factor'. (Musa introduced the term 'testing compression factor' to account for the removal of redundancy inherent in testing of the operational environment. For example, if one hour of test represents 10 hours of operation then the testing compression factor is 10.)

The approach of Cho goes some way towards dealing with the first of Musa's objections cited above. The second deficiency suggests a possible extension of the input space approach to determining mean-time-to-failure of software. In fact there are two ways of reconciling Musa's objections with the technique proposed by Cho. We may call these the complementary approach and the extensional approach. In the complementary approach we see Cho's SIAD tree and quality control technique as allowing a completely different approach to software testing. The ability of the method to draw on well-tried and tested (sic) results from statistical quality control is a strong advantage of Cho's approach.

TABLE 3. SIAD Tree for the PASCAL Subset

index	element	symbol
1	cmpdstmt	X1
2	*begin* stmtlist *end*	X11
3	stmt {; stmt}	X111
4	simplestmt	X1111
5	assignstmt	X11111
6	identifier := expression	X111111
7	term {+ term }	X1111111
8	factor {* factor}	X11111111
9	(expression)	X111111111
10	identifier	X111111112
11	iostmt	X11112
12	READ (identifier)	X111121
13	WRITE (identifier)	X111122
14	structstmt	X1112
15	cmpdstmt	X11121
16	ifstmt	X11122
17	*if* condition *then* stmt	X111221
18	*else* stmt	X1112211
19	expression	X1112212
20	term {+ term }	X11122121
21	factor{* factor}	X111221211
22	(expression)	X1112212111
23	identifier	X1112212112
24	relation	X1112213
25	<	X11122131
26	≤	X11122132
27	=	X11122133
28	≠	X11122134
29	≥	X11122135
30	>	X11122136
31	expression	X1112214
32	term{ + term}	X11122141
33	factor {* factor}	X111221411
34	(expression)	X1112214111
35	identifier	X1112214112
36	whilestmt	X11123
37	*while* condition *do* stmt	X111231
38	expression	X1112311
39	term {+ term}	X11123111
40	factor {* factor}	X111231111
41	(expression)	X1112311111
42	identifier	X1112311112
43	relation	X1112312
44	<	X11123121
45	≤	X11123122
46	=	X11123123
47	≠	X11123124
48	≥	X11123125
49	>	X11123126
50	expression	X1112313
51	term {+ term }	X11123131
52	factor {* factor}	X111231311
53	(expression)	X1112313111
54	identifier	X1112313112

Musa's model of software reliability is a well-known and important one. In the extensional approach the software quality control technique of Cho may be refined and used to give information on mean time to failure and on the 'testing compression factor'.

6. CONCLUSION

The author believes that the SIAD tree approach of Cho is a valuable first step towards software quality control particularly as it has been demonstrated that languages defined in terms of BNF are amenable to the approach. The method will also be applicable in connection with the data-oriented software development techniques of Jackson (1975) and Warnier (1976). However it is felt that the method needs refinement and that more data on expected frequencies of occurrence of data elements needs to be available.

Software quality control is an appealing technique. It is still in its infancy but as more sophisticated methods of sampling based on knowledge of properties are developed the technique will produce more valid results.

It does not provide a measure of MTTF but it does involve the user in a formal acceptance methodology and this is a valuable aspect of the technique.

7. REFERENCES

CALINGAERT, P. (1979): *Assemblers, compilers and program translation,* Computer Science Press, Potomac.

CHO, C.K. (1980): *An Introduction to Software Quality Control,* Wiley, New York.

DENNING, P.J. (1971): Third Generation Computer Systems, *Comput. Surv.,* 3, No. 4, pp. 175-216.

DIJKSTRA, E.W. (1972): Notes on Structured Programming, in *Structured Programming,* ed. E.W. Dijkstra, O.J. Dahl and C.A.R. Hoare, Academic Press, London, pp. 1-82.

GOOS, G. (1973): Hierarchies, in *Advanced Course on Software Engineering,* ed. F.L. Bauer, Springer-Verlag, Heidelberg, pp. 29-46.

HORNING, J.J. and RANDELL, B., (1973): Process Structuring, *Comput. Surv.,* 5, No. 1, pp. 177-193.

JACKSON, M.A. (1975): *Principles of program design,* Academic Press, London.

KOPETZ, H. (1978): *Software reliability,* MacMillan, London.

MUSA, J.D. (1980): The Measurement and Management of Software Reliability, *Proc. IEEE,* 68, No. 9, September 1980, pp. 1131-1143.

WARNIER, J.D. (1976): *Logical Construction of Programs,* Van Nostrand Reinhold, New York.

Software Science — The Emperor's New Clothes?

A. M. Lister*

The emergent field of software science has recently received so much publicity that it seems appropriate to pose the question above. This paper attempts to provide an answer by examining the methodology of software science, and by pointing out apparent anomalies in three major areas: the length equation, the notion of potential volume, and the notion of language level. The paper concludes that the emperor is in urgent need of a good tailor.

Keywords and Phrases: Software science, programming language, length equation, potential volume, language level.
CR Categories: 4.6, 5.2.

1. INTRODUCTION

In the last three years software science (Halstead, 1977) has received a great deal of publicity (e.g. Fitzsimmons and Love, 1978; van der Knijff, 1978; IEEE, 1979). The flood of literature has been so voluminous that the passive spectator may have been persuaded that here indeed is a significant breakthrough, or wonder, if of more cynical disposition, whether enthusiasm has outrun discretion. The situation is reminiscent of that in the fairytale, where courtiers outbid each other to exclaim over the magnificence of the emperor's new clothes; it was left to a small child to point out that the emperor was in fact naked. This paper is written from the child's point of view, in the belief that critical appraisal is currently appropriate.

Section 2 of the paper outlines the notation and terminology of software science. Its purpose is not to summarise the theory (see, for example, Fitzsimmons and Love, 1978, or van der Knijff, 1978) but to introduce the notation used later. Section 3 discusses some aspects of the methodology of software science, while the following sections detail apparent deficiencies in three specific major areas; the length equation, the potential volume of an algorithm, and language level. Section 7 contains a summary and some concluding remarks.

2. SOFTWARE SCIENCE MEASURES

The fundamental measures of software science, from which all others are derived, are (for any program)

$n1$ — number of distinct operators used
$n2$ — number of distinct operands used
$N1$ — number of operator occurrences
$N2$ — number of operand occurrences.

The *vocabulary* of the program is

$$n = n1 + n2$$

and the program *length* is

$$N = N1 + N2$$

The *volume* of a program, which is the minimum number of bits required to hold it, is

$$V = N \log_2 n$$

The *potential volume* V^* of an algorithm is the volume of the minimal program required to express it. V^* is a property of an algorithm, and is independent of the programming language used.

The ratio

$$L = V^*/V$$

is the *level* of a program, and measures the degree of compaction which would be achieved if the language used allowed the algorithm to be expressed in its minimal form. The *effort*

$$E = V/L$$

is held to be a measure of the effort (and hence time) required to write or understand a program.

The final measure of software science is the *language level* of a programming language, given by the product

$$\lambda = L V^*$$

which is asserted to be the quantitative measure which corresponds to intuitive ideas of the level of a programming language.

Calculation of the quantities above for a particular program requires the measurement of $n1$, $n2$, $N1$, and $N2$ for the program, together with a knowledge of V^* for the corresponding algorithm. If $N1$ or $N2$ are not available the *length equation*

$$N \cong n1 \log_2 n1 + n2 \log_2 n2 \ (=\hat{N})$$

is claimed to yield a good approximation to the length N. If V^* is unknown then

$$\hat{L} = (2 \times n2)/(n1 \times N2)$$

can be used as an approximation to the program level L (which can in turn be used to compute approximations $\hat{E}$ and $\hat{\lambda}$ for E and λ).

†Department of Computer Science, University of Queensland, St. Lucia, Qld. 4067. Manuscript received 20 February 1982.

3. THE METHODOLOGY OF SOFTWARE SCIENCE

In principle there are two ways in which to validate the hypotheses of software science (which would then become "laws"). The first is by deduction from known properties of algorithms and programs; the second is by inference from repeated observation. At present our knowledge about algorithms and programs is insufficient for the deductive approach to be viable, though some attempts (e.g. Halstead, 1977; Gordon, 1979) have been made. These attempts have provided plausibility arguments about why the hypotheses *may* be true, but they fall far short of deductive proofs that they *are* true. For example, Halstead (1977, pp. 9-11) derived the length equation by an argument about the information content of different strings of symbols, but the argument rests on questionable assumptions, and at one point erroneously equates two different sets. As Fitzsimmons and Love (1978, p. 6) admit, "no rigorous mathematical derivation for this equation is currently known".

In the absence of deductive proofs software science must rely for its validity on empirical evidence that it works. On the face of it that evidence is impressive: Fitzsimmons and Love, for example, summarise twenty-two experiments which appear to have achieved good results. However, closer examination reveals that the evidence is not so conclusive as it seems, for reasons discussed below.

Since all software science measures are derived from counts of operators and operands in programs it is important that the counting scheme be clearly defined and consistent across experiments. For example, it should be clear which symbols are to be classed as operators and which as operands, whether I/O statements and declarative text are to be included, and whether a symbol which is used (overloaded) with different meanings is to be counted separately for each meaning. Moreover, since software science hypothesises about the influence of language on programming, the counting scheme should be applicable to programs written in any of a wide class of languages. Halstead's original counting scheme (Balut, Halstead, and Bayer, 1974) was devised for Fortran programs; it forms the basis of an analysis program (Ottenstein, 1976) used extensively at Purdue University, the home of most software science research. Unfortunately, the scheme is difficult to apply to more modern languages, particularly those with structured data types (e.g. Pascal's records) and less primitive control constructs. These difficulties can of course be resolved, since they are largely a matter of definition: the problem is that they have been resolved in different ways by different researchers, so that no consistent counting scheme has been used for experiments on non-Fortran programs. The consequences are serious, since it has been shown (Elshoff, 1978) that small variations in a counting scheme for PL/1 programs can affect certain measures by as much as 50 per cent. One can conjecture that the effects of inconsistency over different languages might be even greater. (It is interesting, too, that none of Elshoff's variants was obviously the "right" one to use: he remarks [p. 44] that "the merits of any of the methods can be supported depending on one's point of view".) Until consistent counting schemes are devised and adopted it would be rash to pin too much faith on some experimental results.

Another area which merits caution is the interpretation of the results themselves. Many workers have reported their results in the form of a correlation coefficient between two measures, the first as predicted by the theory and the second as observed in the experiment. A high correlation is usually held up as evidence that the hypothesis in question is true. Correlation coefficients are, of course, notoriously subject to abuse: some reasons for cautious interpretation in this context are —

(1) A correlation coefficient is a measure of linear dependence between two random variables. None of the reported experiments attempts to show that the variables in question are indeed random.

(2) Given that the two variables *are* random, a high correlation coefficient indicates that there is a strong linear relationship between them, but it does not indicate what that relationship is. For example, a high correlation between $\hat{N}$ and N does not show, as some workers have suggested, that $\hat{N} = N$, but only that $\hat{N}$ and N are related. Linear regression analysis, which would establish what the relationship is, has not been performed in most reported experiments, and is not even mentioned in the comprehensive survey of results given by Fitzsimmons and Love (1978).

(3) In many reported experiments the sample size is too small for great significance to be attached to the results. For example, of the twenty-two experiments summarised by Fitzsimmons and Love fourteen have sample size less than 15 and only two have a sample greater than 50 (the sample size for one experiment is not given).

(4) The effect of small sample size is sometimes compounded by the extreme dependence of the results on one or two particular observations. A notable example arises in an experiment on students' programs (Shen, 1979), where the results are so heavily dependent on the worst student that his elimination from the sample (of 31) reduces a reported correlation coefficient from 0.46 to 0.20. The extent to which this effect pervades software science results is impossible to gauge since not all reports provide the necessary raw data. It is however evident in some of Halstead's work on the length equation (Halstead, 1977), where it is inadvertently disguised by the use of a logarithmic scale and the "averaging" of sets of data points.

Of course these reservations do not apply to all reported results. The purpose of listing them is not to discredit all the experimental evidence in one fell swoop, but to point out that it is not so substantial as might first appear. Since, as mentioned earlier, the quality of evidence is vital, it is unfortunate that the rather cavalier methods of some researchers (and the undocumented methods of others) make it difficult to know which results can be treated with confidence. It was lack of confidence in some results, increased by failure to reproduce them in independent experiments, which led to the analysis reported in the next three sections.

4. THE LENGTH EQUATION

In this section we suggest that the software science length equation is not as well established as is often claimed, and that it may not hold for programs written in "structured" languages (such as Pascal) unless a counter-intuitive counting scheme is adopted.

The length equation (see section 2) states that the length N of a program may be closely approximated by the estimator $\hat{N}$ defined by

$$\hat{N} = n1 \log_2 n1 + n2 \log_2 n2$$

TABLE 1. Performance of the length estimator for large Pascal programs (1) repeated control structures as single operators (2) repeated control structures as distinct operators.

Program	$N/\hat{N}$	
	(1)	(2)
1	2.28	1.37
2	2.41	1.36
3	1.48	0.822
4	1.12	0.655
5	1.72	1.06
6	1.20	0.895
7	1.64	0.779
8	1.61	1.08
9	1.57	0.863
Mean	1.67	0.987
Std. Dev.	0.408	0.208

provided that the program is well-written in the sense of containing few impurities (Halstead, 1977). The significance of the length equation is that the length of a program, and hence the volume and effort measures, can be estimated before the program is written (provided the distinct operators and operands required can be enumerated).

Empirical evidence supporting the length equation has been provided by several researchers, and has been conveniently summarised by Fitzsimmons and Love (1978). It is noteworthy that all the evidence reported by Fitzsimmons and Love stems from Fortran or PL/1 programs. Further measurements (Johnston and Lister, 1979) on small Pascal programs have also provided some evidence to support the length equation, despite the fact that the programs, being written by students, probably contain an unusually high proportion of impurities. However, similar measurements on nine large professionally written Pascal programs (including the Pascal compiler itself) show some large discrepancies between the values of N and $\hat{N}$. The radio of $N/\hat{N}$ for each program is shown in the centre column of Table 1. Although the sample is small, the discrepancies are large enough to provoke further analysis.

The centre column of Table 1 shows that $\hat{N}$ consistently underestimates N for the programs considered. Examination of the measurements on these programs shows that *n1* is considerably less than *n2*, whereas for small programs *n1* and *n2* are of comparable magnitude. This observation suggests that for large Pascal programs $\hat{N}$ falls short of N because the contribution from *n1* is too small.

In the counting scheme used to produce Table 1 (and in all schemes published elsewhere) *n1* is derived from —

(1) The built-in operators, procedures and functions of the language.
(2) User-defined procedures and functions.
(3) Labels which are the target of a control transfer.

Halstead (1977, p. 8) remarks that "the ability to define labeled points, like the ability to define new functions, removes any limitation on the growth of *n1* that might otherwise be imposed by . . . the design of a language." However, programs written in Pascal, or any other language with "structured" control constructs, contain very few operators of class 3. This means that once such programs are large enough to contain most of the built-in operators the growth of *n1* with program size is constrained by the number of user-defined procedures and functions. Analysis of the sample of large Pascal programs indicates that there are simply not enough of these user-defined operators to give a value of *n1* large enough to satisfy the length equation.

The situation with Fortran programs, which have provided most of the empirical support for the length equation, is quite different. Since nearly all transfers of control in Fortran are effected by jumping to a label, operators of class 3 make a significant contribution to *n1*. Moreover, the ubiquity of such operators removes any *a priori* constraint on the growth of *n1* with program size. Thus the fact that large Fortran programs reportedly obey the length equation while large Pascal ones do not is apparently due to the differences in control constructs between the two languages.

Support for this hypothesis comes from the right most column of Table 1, which shows the result of altering the Pascal counting scheme so that each occurrence of a control construct (e.g. *while . . . do*) is counted as a distinct operator, irrespective of how many times it is used. This counting scheme, which reflects a crude mapping from Pascal control constructs to Fortran control constructs, can be seen to give much closer adherence to the length equation than shown previously. Such a counting scheme, however, is most unappealing. One of the beauties of Pascal is its economy of control structure: it seems counter-intuitive to wilfully disregard this economy in the counting scheme used.

Results which might be thought to cast doubt on the analysis above are those reported by Elshoff (1978) for "structured" PL/1 programs. Since PL/1 possesses control constructs similar to those of Pascal (if not quite as elegant), it might be expected that large PL/1 programs would diverge from the length equation in the same way as large Pascal programs. This expectation is apparently confounded by Elshoff's results, which are claimed to support the length equation over a large range of program sizes. However, closer examination of Elshoff's experiments reveals that —

(a) despite their "structured" nature the sample programs contain a significant number of GOTO statements, as is evidenced by the reported effect on *n1* when the method of counting GOTO's was changed.
(b) adherence to the length equation is due in part to a counting scheme for constants which was tuned specificially to achieve such adherence.

It is interesting to note that despite optimisation of the counting scheme Elshoff's measurements give no better support to $n1 \log_2 n1 + n2 \log_2 n2$ as an estimator for N than they do to a number of other arbitrary functions. For example, Table 2 shows the correlation between N and $\hat{N}$, and the root mean square distance of measured points from the line $N = \hat{N}$, for various definitions of $\hat{N}$. The correlations are almost uniformly high, and $\hat{N} = 10n2$ in fact gives a better fit than $\hat{N} = n1 \log_2 n1 + n2 \log_2 n2$. (Incidentally, $10n2$ also gives a better fit for the sample of 31 Fortran programs used by Shen (1979), though we do not propose it as a new length estimator!) Table 2 illustrates the danger, referred to in Section 3, of putting too much faith in correlation coefficients.

In the absence of a deductive proof the length equation must stand or fall on the empirical evidence provided, which to date has come solely from Fortran and PL/1 programs. The above analysis suggests that the evidence from PL/1 programs is inconclusive, and that evidence from Fortran programs depends heavily on that language's prim-

TABLE 2. Performance of various length estimators on Elshoff's PL/1 programs.

Definition of $\hat{N}$	Correlation between N and $\hat{N}$	RMS Distance from $N = \hat{N}$
$n1 \log_2 n1 + n2 \log_2 n2$	0.985	497
$n2$	0.987	3427
$10n2$	0.987	432
$n1 + n2$	0.985	3388
$n1^2 + n2^2$	0.942	222939

itive control constructs. It also suggests that evidence from languages with structured control constructs will support the length equation only if a counter-intuitive and unjustifiable counting scheme is adopted.

5. POTENTIAL VOLUME

We recall from Section 2 that the potential volume V^* of an algorithm is the volume of the minimal program required to express it. This minimal program is in fact a call to a procedure which embodies the algorithm: certainly no algorithm can be expressed in less than a procedure call. A procedure call is conventionally regarded in software science as having two operators (the procedure name and a grouping symbol) and as many operands as there are conceptually distinct input or output parameters. Since each symbol is used only once, the volume of the procedure call, and hence V^*, is given by

$$V^* = (2 + n2^*) \log_2 (2 + n2^*)$$

where $n2^*$ is the number of parameters (Halstead, 1977). Thus the calculation of V^* is straightforward, provided the parameters can be readily enumerated.

Unfortunately, this is the case in only the simplest examples, such as algorithms which compute mathematical functions of a specified number of variables. In general it is extremely difficult to determine how many conceptually distinct parameters an algorithm has. For example, an algorithm which operates on English text (perhaps to generate an index) may be regarded as having a single input parameter which is the entire text, or a number of parameters equal to the number of characters in the text. The effect of these different views on the value of V^* is significant. A perhaps more convincing example is a compiler, where it is difficult to formulate a plausible argument in favour of any particular number of parameters. It is interesting to note that in the author's measurement of a Pascal compiler the value of V^* (calculated indirectly from the relation $V^* = LV$ with the estimator $\hat{L}$ substituted for L) was 369, implying that the compiler has approximately 61 parameters. It is difficult to imagine what these 61 parameters are.

There is little guidance in the literature to the evaluation of $n2^*$ in non-trivial cases: all the accessible examples are of algorithms which transform readily identifiable inputs into equally identifiable results. Moreover, the input and output parameters in these examples are atomic data items such as integers, whereas in many real-life cases of interest the input and output are structured in some way. It seems important that this structure be taken into account when determining the minimum number of symbols in which an algorithm can be expressed: an ad hoc method of doing this has been used in one particular experiment (Johnston and Lister, 1979), but other methods, which give different results, are equally plausible.

Of course, the number of parameters of an algorithm is a matter of one's point of view or, more technically, one's level of abstraction. In the limit, any number of parameters can be reduced to one by suitable encoding: examples are the incorporation of the parameters into a single record, or (more esoterically) their encoding into a single integer by some Godel numbering scheme. Thus at a high enough level of abstraction all algorithms can be regarded as having a single parameter, a point of view which would render the definition of V^* vacuous. Such a view is technically compatible with Halstead's definition of $n2^*$ as the number of "conceptually distinct" parameters though it is clearly not in the spirit of that definition. "Conceptually distinct" is a term which acquires meaning only if the level of abstraction is defined; until this is done the evaluation of V^* will remain an arbitrary and ill-defined exercise.

The implications are serious, since the measures L, E, and λ are all derived from V^*. In practice many researchers avoid V^* by using the estimator $\hat{L}$ in place of L, hence deriving estimates of E and λ. The validity of this practice depends on the accuracy of $\hat{L}$ as an estimator for L. As with other software science relations, no deductive proof that $\hat{L} = L$ has been offered, and the relation therefore relies on empirical evidence. Since such evidence can come only from experiments in which L is computed from V^* it seems reasonable to treat it with caution.

6. LANGUAGE LEVEL

The language level λ is an attempt to formulate a qualitative measure which corresponds to intuitive notions about the level ("high" or "low") of a language. Halstead (1977) has tabulated mean values of λ for six languages, and these values do indeed reflect general consensus about the levels of the languages concerned. However, the current author's attempts to establish a similar value for Pascal led to two results which deserved further analysis. First, there was a wide variation in the values of λ obtained for different programs, and second, the mean value of λ for Pascal was lower than that reported by Halstead for assembly language. (The programs used in these experiments were those described in section 4 and the sample described by Johnston and Lister [1979].)

The variations in λ seem disturbing in view of Halstead's assertion that "the product L times V^* [$=\lambda$] remains constant for any one language" (Halstead, 1977, p. 62). In fact, despite Halstead's assertion, the variations are not surprising, since the following argument shows that λ cannot be constant over all algorithms expressed in a given language. Since $\lambda = LV^*$ and $L = V^*/V$ we have $\lambda = (V^*)^2/V$; thus for λ to be constant it is necessary for V to vary with the square of V^*. Now V^* depends on the number of parameters of the algorithm (however that is defined), while V is a measure of the bulk of the algorithm when expressed in the language concerned (and the bulk depends on the algorithm's complexity). It is most unlikely that the complexity of an algorithm is in any mathematical sense related to the number of parameters, least of all by a square law. Indeed, to take a single example, there is an infinite number of algorithms of widely varying complexity (and hence widely varying V when implemented in the same language) which all compute a function of a single variable and which all therefore have $V^* = 4 \log_2 4$.

Another anomaly which casts doubt on the constancy of λ arises from consideration of the effort measure E. Since $E = V/L$ and $\lambda = LV^*$ we have $E = (V^*)^3/\lambda^2$. This relation implies that in a particular language all algorithms with the same potential volume (e.g. all functions of a single variable) can be programmed with equal effort. If that were true, life would be easy indeed!

These arguments indicate that contrary to Halstead's assertion λ is not a constant for a particular language. However, it is conceivable that the *mean* value of λ, taken over a number of different programs may (despite a large variance) be a measure of the expressive power of a language. If this is so then any published value of λ should indicate the set of programs from which it is derived, which in turn should be justified as being in some sense representative. Furthermore, the levels of different languages should be compared only if they are obtained by programming the same set of algorithms. Regrettably this has not been the case.

If one accepts the arguments above, the role of λ is at best reduced from an absolute to a comparative measure of language level. Its importance even in this reduced role is debatable. For a particular algorithm expressed in languages A and B, the ratio λ_A/λ_B is the same as the ratio V_B/V_A. The value of λ as a measure distinct from V is therefore questionable.

7. CONCLUSIONS

This paper has argued that without deductive proofs the validity of software science must rest on the quality of evidence provided. This evidence is superficially impressive, but close inspection reveals methodological flaws which generate considerable unease. Furthermore, arguments have been advanced to suggest that

(1) The length equation may hold only for languages with primitive control constructs.
(2) The notion of potential volume is ill-defined.
(3) The validity of the language level measure is suspect.

These arguments threaten three major areas of software science; although they are not conclusive they are forceful enough to suggest that the foundations of software science are perilously weak. If software science is to be convincing it needs a clearer definition of its assumptions, goals, and domain of application. It also needs a methodology which is seen to be rigorous. It is possible that software science will indeed develop in these ways, providing useful and worthwhile results. At present, however, an interim judgement is that the emperor may not be quite naked, but he is so raggedly dressed as to be almost indecent.

8. ACKNOWLEDGEMENTS

Dan Johnston supplied many of the measurements which motivated this paper, and the University of Southern California provided hospitality while it was being written. Both are gratefully acknowledged.

9. REFERENCES

BALUT, N., HALSTEAD, M.H., and BAYER, R. (1974), The experimental verification of a structural property of FORTRAN programs. *Proc. ACM Annual Conf.*, ACM, New York, pp. 207-211.

ELSHOFF, J.L. (1978), An investigation to the effects of counting methods used on software science measurements, *ACM SIGPLAN Notices*, Vol. 13, No. 2, Feb. 1978, pp. 30-45.

FITZSIMMONS, A., and LOVE, T. (1978), A review and evaluation of software science, *Computing Surveys*, Vol. 10, No. 1, March 1978, pp. 3-18.

GORDON, R.D. (1979), A qualitative justification for a measure of program clarity, *IEEE Trans. on Software Engineering*, Vol. 5, No. 2, March 1979.

HALSTEAD, M.H. (1977), *Elements of Software Science*, Elsevier North-Holland, New York.

IEEE (1979) IEEE *Transactions on Software Engineering* (special issue on software science), Vol. 5, No. 2, March 1979.

JOHNSTON, D.B., and LISTER, A.M. (1979), An experiment in software science, *Proc. Symposium on Language Design and Programming Methodology*, Sydney, September 1979. Reprinted in *Lecture Notes in Computer Science*, No. 79, Springer-Verlag, pp. 195-215.

OTTENSTEIN, K.J. (1976), A program to count operators and operands for ANSI Fortran modules, *Report CSD-TR 196*, Department of Computer Science, Purdue University.

SHEN, V.Y. (1979), The relationship between student grades and software science parameters. *Proc. 3rd Int. Conf. on Computer Software and its Applications*, Chicago, Nov. 1979.

VAN DER KNIJFF, D.J.J. (1978), Software physics and program analysis. *Aust. Comput. J.*, Vol. 10, No. 3, August, 1978, pp. 82-86.

Data Base Interfaces

D. H. Scuse*

This paper describes how the dependence of systems of application programs on data base management systems can be lessened by placing a software interface between the application system and the data base management system. The interface makes the design and implementation of the application system easier by supporting more sophisticated data models and more complex data manipulation than are normally supported by the data base management system, and can also be used to increase the control that the data base administrator has on the data base environment.

Keywords and phrases: data base, DBA, data model, interface.
CR category: 4.33.

1. INTRODUCTION

If a software program product (such as a data base management system) does not provide all of the facilities that are required by the users of the product, the group supporting the product at an installation normally must either modify the program product or modify the application programs that use the program product so that the application programs perform the extra processing that is required. Modifying a program product is difficult because the support group does not necessarily have personnel sufficiently skilled to be able to locate and modify the appropriate modules in the program product. If the modifications can be made, the changes must be reapplied each time a new version of the product becomes available, and a change in the logic of the program product may force the support group to start over again in determining where and how to apply the changes. Even if the program product was developed at the same installation, the support group will not necessarily want to modify a stable product. Thus, the users are normally forced to modify the application programs that use the program product. If the requirements or the program product itself change, then the application programs must be modified again so that the new circumstances are taken into consideration. In a large system involving several hundred application programs, making even a small modification to the programs is a very time-consuming process, especially when each application program must be tested before the modified system can be put into production.

Fortunately, the capabilities of a program product can be extended without modifying either the product itself or the application programs that use the product. The solution to the problem involves placing a software interface between the application programs and the program product. Instead of invoking the program product directly, the application programs invoke the interface which then invokes the program product. The use of such an interface with a data base management system makes the design, implementation, and maintenance of the application system much easier by concentrating the low-level details of the data base manipulation in the interface instead of in the application programs.

The remainder of this paper examines some of the benefits of using such an interface in a data base environment. The effect that a data base interface has had on a major research project is also described.

2. ABSTRACT DATA MODEL

There are three levels of data models that can be used in the design of a data base. The highest level model is an abstract data model such as the third or fourth normal form of the relational model (Date, 1981) or the entry-relationship data model (Chen, 1980). An abstract data model is used to define the information to be stored in a data base without regard for how the data will actually be organised in the data base. The next lower level of model is the data base management system model, the model of data supported by a particular data base management system. The data base management system model normally requires that the information in a data base be defined with a specific structure, such as a hierarchy or a network, and may impose restrictions on the order in which information may be accessed. Related pieces of information are normally grouped together and referred to as a segment. The lowest level of data model is the storage model which defines how the data base management system model is to be implemented, that is, it defines the access paths to be used to link segments together and the access methods to be used to manipulate the data base segments.

During the design of a system, the data base administrator (DBA) should define the information to be included in the data base using an abstract data model. This model allows the DBA to concentrate on the logical organisation of the information instead of on the details of how the information is to be stored in the data base. Once the basic design of the data base is complete, the DBA restructures the information in the abstract model so that it conforms to the structure of the data base management system model being used. It is this data base management system model that is used in the remainder of the design and in the implementation of the system. However, if a data base interface is placed between the application programs and the data base management system, the interface can support the abstract model of the data base by translating requests defined in terms of the abstract model into the equivalent requests for the data base management system model. Thus, it becomes possible to use the abstract model throughout the system, not just during a portion of the design phase. The use of an abstract data model makes the design of the system significantly easier

*Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada. Manuscript received October 1981.

since the analysts concentrate on the manipulation of the information instead of the manipulation of the information as it would be stored in a specific data base.

The use of an abstract data model decreases the dependence of the application programs on the data base management system being used: should it become necessary, the data base management system model could be changed and the data base reorganised without affecting the application programs. Such changes should require only a modification of the interface, not of the application programs.

The data base interface can be used to support different views of the abstract data model if the associated data base management system does not support such views. For example, segments that are not required by certain application programs would not be included in their views of the data base. The view of the data model could also include field-level independence if this feature is not supported by the data base management system. With field-level independence, the fields to be returned and the order of those fields are defined in each view. If this ordering is not the same as the ordering supported by the data base management system, the interface rearranges the fields so that the application program retains its independence of the data base.

The abstract data model is particularly useful when a system must be maintained on several machines with different data base management systems. For example, in a health-information environment, hospitals may use different machines and data base management systems but perform the same processing on the same type of information. A common data model could be used by each hospital if the appropriate data base interface is written for each data base management system. A common model makes the interchanging of information between machines much easier. If a common application programming language is supported on some or all of the machines, it may even be possible to transfer application programs from one installation to another even though the underlying data base management systems are not the same. Supporting a common data model on different machines by using a data base interface is much easier than attempting to design a machine-independent data base management system that would run on each machine being used or attempting to find one data base management system that is supported on each machine but that provides the required data base facilities. As networks of machines become more common, the ability to share information will become increasingly important.

3. SYSTEM IMPLEMENTATION

The use of an abstract data model, supported by a data base interface, makes the implementation as well as the design of a system easier. When an abstract model is used, the application programmers who implement a system can concentrate on the algorithms required to manipulate the information instead of being concerned with the mechanics of manipulating a particular data base. For example, some data base management systems require the application programmers to be aware of and to manipulate the access paths that link segments together. This low-level processing can easily be performed in the data base interface instead of in the application program. As the complexity of the application programs is reduced, not only is the time required to write the application programs reduced, but the number of errors in the application programs should also be reduced.

The data base interface can also be used to extend the data manipulation facilities supported by the data base management system. Some data base management systems do not permit qualification statements to be included with a request to the data base management system, forcing the application programmer to include the logic to examine each segment returned until the desired segment is found. If this processing is moved into the interface, the number of statements required to implement the application program is again reduced.

The amount of exception-handling logic required in the application programs can also be reduced by using a data base interface. Moving some of the exception checking into the interface frees the application programs from having to compare the status code returned with a large list of possible codes. The interface can analyse the status code and determine whether or not the condition is serious enough to cause processing to terminate. Moving status code checking into the interface also makes it easier to adapt a system to new status codes returned by a new version of the data base management system since only the interface need be modified.

The data base interface can also compress the fields within segments if compression (removing trailing blanks, etcetera) is not supported automatically by the data base management system. The extra CPU time required to support compression can normally be justified as the size of data bases increases and the cost of CPU cycles decreases. If necessary, the interface could also change the representation of numeric fields by storing numeric values in the most efficient internal representation.

The interface can be used to enforce certain types of integrity constraints if they are not already supported by the data base management system. Integrity constraints define the relationships that must exist before a new segment can be inserted or an existing segment can be modified or deleted. The constraints may consist of the list of values that a particular field may take on, the relationships that must exist among fields within a segment, and the relationships that must exist among segments. In a system where only one application program inserts, modifies, and deletes segments, the integrity constraints can be included in that program; however, if many application programs are permitted to modify the data base, the checking of the integrity constraints could be performed in the interface in order to ensure that the checking is carried out correctly and completely.

While most data base management systems support at least a basic security system, the data base interface can be used to extend the security constraints provided by the data base management system. A basic level of security is already provided by the interface if it supports multiple views and field-level independence. This security can be improved if the interface maintains a profile for each user of the system. The profile of a user contains a list of the data base views that the user is permitted to access and a list of the access rights (read, update, etcetera) that the user has to those views. A simple extension to the user's profile would permit the interface to accept or reject access to a view based on field values. For example, in a personnel data base, each manager would be permitted to access only the segments of employees in his section and would be denied access to the segments of employees in other sections.

4. SYSTEM TESTING AND TUNING

The data base interface makes the testing of a new system much easier because all requests to the data base management system are passed through the interface, making it possible to trace the activity of each application program as it executes. It is even possible to test a system for which the associated data base has not yet been created by having the interface return dummy segments which have the same format as the segments that will eventually be stored in the data base.

A data base interface can be useful during the testing of a new version of the data base management system itself. Some data base management systems have bugs in new versions and must be tested extensively before being placed in production. One method of testing a new version of the data base management system is to create a copy of an existing data base using the new version of the system; then, as changes are made to the production version of the data base, the interface saves a copy of each change request; these changes are applied at a later time to the test data base using the new version of the data base management system. The test data base can then be compared with the production data base to ensure that they both contain the same data.

The data base interface is a useful tool for the DBA when he is evaluating and tuning a data base. Since all data base requests are passed through the interface, the interface can gather statistics on the types of requests being issued, the users who are accessing the data base, the areas of the data base being manipulated, etcetera. (These statistics would be used to supplement the statistics generated by the data base management system itself.) The statistics provide the DBA with some of the information required to evaluate the efficiency of a data base storage model.

The DBA can also use the interface to change the method used to process a particular type of request. For example, a request that is issued infrequently could be processed by the interface as a sequential search (of a reasonably small area of the data base), while, if the request is issued frequently, an appropriate index could be maintained by the data base management system and the interface could generate requests that take advantage of this index. Thus, the DBA has sufficient control of the system to be able to adapt the storage model of the data base to meet the changing needs of the users. The DBA can also use the interface to restrict certain types of processing based on the time of day: during periods of high use, the interface could reject expensive requests but during periods of medium to low use, the same request would be accepted.

For a system that is large and must be available 24 hours each day, a data base interface could be used to support a differential data base (Severence and Lohman, 1976) even though such a data base is not supported by the host data base management system. A differential data base contains all changes that are made to another, static data base. When a change is made to the data base, the interface stores the modified segment in the differential data base; the static data base is not modified once it has been created. When a segment is retrieved from the data base, the interface first searches the differential data base for the segment; if the segment is not found, the interface searches the static data base. (Severence, 1976, has shown how hashing can be used to eliminate most accesses to the differential data base when the desired segment is actually in the static data base.) The differential data base reduces many of the problems inherent in managing a large data base to the equivalent, but simpler problems for a small data base.

5. FISHERIES INFORMATION SYSTEM

A data base interface has been used very successfully in a research project being carried out by A.N. Arnason of the Department of Computer Science, University of Manitoba (Arnason *et al.*, 1981). The project involves the development of an integrated management information system (MIS) and deterministic simulator system. The MIS manages the information generated by an experimental fish hatchery in Manitoba. The hatchery records information concerning the size of the fish and the conditions under which the fish are raised (water temperature, feeding levels, etcetera). The MIS receives, edits, and stores the information generated by the hatchery. Information is then extracted from the system quite easily using a specially-developed language; the information is normally presented in time-date order with the various pieces of information having been correlated automatically by the MIS. The simulator system is being used to test growth strategies at the fish hatchery. The user of the simulator system carries on a dialogue with the simulator, specifying the conditions to be used during an experiment, and the simulator prints the size of the fish at specified intervals during the experiment. The use of the simulator to test management strategies is obviously cheaper and faster than experimenting with the real hatchery since a bad strategy does not cause fish to be lost and good strategies can be refined quite quickly.

The integrated system is being run on the University of Manitoba's AMDAHL 470/V7 computer, and IBM's Information Management System (IMS) is the data base management system used by the MIS. The IMS data base required to store the hatcheries information and the simulated information is reasonably complex, involving several logical relationships and special indicators in the segments. However, only the DBA for the system is aware of the structure or contents of the IMS data base; all application programmers view the data base using the relational data model. Each type of segment in the data base is viewed as a separate relation, the segments (or tuples) of which can be accessed both sequentially and randomly. The translation of the relational data model into the IMS data model is performed by a data base interface. The interface also performs processing that is not provided by IMS.

All information that is required to issue a request to the interface is included in a data structure that the application programmer copies into the application program from a system library. This data structure defines the format of the segments that can be accessed and also contains command and key fields that are set in the application program and status fields that are interrogated by the application program. The application programmer sets the command field for each type of segment to be processed (more than one segment type can be processed with each request), and, if a particular segment is to be retrieved randomly, sets the key field of the segment. Special commands were defined to permit the application programmer to retrieve the first segment with a key greater than, or the first segment with a key less than, the specified key since much of the processing is in time-date order and the application programmer may not know the exact starting time of an experiment.

Among the features provided by the interface but not supported by IMS is backwards processing. The segments in the IMS data base are stored in chronological order since the application programs normally access segments chronologically. However, some application programs require access to segments in reverse-chronological order. Since this type of processing is not supported by IMS and is reasonably complex, the logic required to access segments in reverse order was added to the interface instead of to the application programs. Thus, the application programs remain free of intricate data base processing and can be implemented much faster.

An additional advantage of using the data base interface is that several random-access files that are not stored in the data base can be accessed by the application program as though they were in the data base. The application programmer views the information in these files with the same model as he views the data base; however, when a request involving this information is issued, the interface accesses the appropriate random-access file instead of the data base. Thus it is not necessary for the user to be aware of the actual location of the information being processed. With this organisation of information, a change in the location of information requires a modification only to the interface, not to the application programs. (The information was stored in the random-access files in order to permit the DBA to modify the information more easily; as a result of storing the information in these files, the complexity of the IMS data base was reduced since the information in the files could not have been stored in the data base without a significant amount of redundancy.)

The data base interface has made the development and the integration of the management information system and the simulator system much easier because application programmers have been able to concentrate on the information being manipulated without having to know the details of how the information is stored. (In fact, the application programmers were quite happy not to have to fight with IMS in order to get their programs running.) The interface has also made it possible to change the format of the data base (and this was done twice) when dictated by new circumstances without having to modify any of the application programs.

6. CONCLUSIONS

The two major reasons for using an interface are first, to simplify the design of application systems by supporting a higher level data model than is supported by the data base management system being used, and secondly, to simplify the implementation of application systems by moving the low-level data base manipulation out of the application programs. The overall effect of the interface is to reduce the time required to implement a system and to ensure that it is functioning correctly.

The major disadvantage of a data base interface is the extra CPU time required at program execution time by the interface to translate the abstract model requests into data base requests. An equivalent translation could be performed at compile time instead of at execution time by writing a preprocessor that translates the abstract requests into in-line data base requests. However, the use of a preprocessor does not preserve program isolation to the same degree as the interface: changes to the preprocessor force the recompilation of the application programs while changes to an interface require the recompilation of only the interface. Also, a preprocessor can not support the same level of complexity of extra data manipulation that the interface can because the extra statements are generated in the application programs each time that they are used instead of only once in the interface. A careful design of the abstract data model used with the interface should minimise the translation overhead but still maintain program/data base independence.

The data base interface will be useful as long as the production-oriented data base management systems force application programmers to be aware of internal storage models and fail to support the high-level data manipulation facilities that are required in many application systems.

REFERENCES

ARNASON, A.N., SCHWARZ, C.J., and SCUSE, D.H. (1981): "An On-line Simulator and Database System for the Management of a Commercial Fish Farm", Presented at the Winter Simulation Conference, Atlanta, Georgia, December, 1981.

CHEN, P. (ed.) (1980): *Entity-Relationship Approach to Systems Analysis and Design,* Proceedings of the International Conference on the Entity-Relationship Approach to Systems Analysis and Design, Los Angeles, 1979, North-Holland Publishing Company, Amsterdam.

DATE, C.J. (1981): *An Introduction to Data Base Systems,* Third Edition. Addison Wesley Publishing Company, pp. 237-264.

SEVERENCE, D.G. and LOHMAN, G.M. (1976): Differential Files: Their Application to the Maintenance of Large Data Bases, *ACM Trans. on Data Base Systems,* 1, 3, pp. 256-267.

Letters to the Editor

COMMENT ON QUERY LANGUAGE ARTICLE

I was intrigued to read an article in the November 1981 issue of your journal entitled "A Review of Data Base Languages" by M.A. Robinson. In it the author refers to some twenty-seven query languages, many of which are either for specific purposes or out of date with current technology.

My greatest concern though regarding the article is that the author failed to mention our product EASYTRIEVE which has almost 150 users in Australia and New Zealand alone. This figure is almost more than the number of users of all the packages mentioned in the article in the same territory. It is difficult therefore to accept the validity of the article as a whole when one of the most popular query languages in the world is not mentioned.

I.J. Farrell,
Pansophic Systems A/sia,
North Sydney, NSW 2060

AUTHOR'S REPLY

In response to the letter by I.J. Farrell, I should point out that my primary aim was to outline what features should exist in a generalised query language. The inclusion of details about various languages was to illustrate that these features do, in fact, exist in some languages. The decision as to which languages to include and which ones to omit was difficult. In general, languages were included because of the availability of information, and more importantly, because I believed that their inclusion contributed to my article.

The number of users in Australia or New Zealand was not one of my criteria. If it had been then languages such as GPLAN or RENDEZVOUS which have no users in either Australia or New Zealand and very few throughout the world, would not have been included. As both GPLAN and RENDEZVOUS contribute to the state of the art, I view their inclusion as mandatory. On the other hand, EASYTRIEVE, which I believe to be a report generator for a file management system rather than a data base query language, could be omitted because of its similarity to other languages which are included.

It is of interest that the British Computer Society recently published a monograph on "Query Languages". This monograph lists about 140 languages, 16 of which are included in my article, but EASYTRIEVE is not included.

I have also received a letter from Mr. George Nichols of Computer Sciences of Australia who has expressed interest in seeing a further review at a future date. Computer Sciences of Australia is the Australian distributor for SYSTEM 2000, and Mr. Nichols has helpfully supplied details of the latest improvements to the system. I would be pleased to hear from any other persons or organisations who feel that they can help me with additional information.

M.A. Robinson,
Chisholm Institute of Technology,
(formerly Caulfield Institute of Technology),
Caulfield East, Vic. 3145

PROCEDURE FOR PROVIDING A SINGULAR VALUE DECOMPOSITION OF A LARGE MATRIX, ON A MINICOMPUTER FOR SPECTROPHOTOMETRIC ANALYSIS

The singular value decomposition of matrix D may be given as $D = U\Sigma V$ (see for example, Golub and Reinisch, 1970) where U consists of orthonormalised eigenvectors associated with the eigenvalues of DD^T and the matrix V^T consists of orthonormalised eigenvectors associated with the eigenvalues of D^TD. The elements of the diagonal matrix Σ are the positive square roots of the eigenvalues of D^TD.

A data matrix D of M rows of N readings may be partitioned

$$D = (D_1 D_2 \ldots D_s \ldots D_r)$$

such that $N_s \leqslant M$ where N_s is the number of columns for the sth partition. We have

$$D_s = U_s \begin{bmatrix} \Sigma_s & 0 \\ 0 & 0 \end{bmatrix} V_s$$

Because of the high correlation of spectrophotometric data, the number of terms which are nonzero in the diagonal matrix are relatively few. Letting the superscript define the step, the factorising of the data becomes in the first step:

$$D = (U_1^1\,\Sigma_1^1\;\; U_2^1\,\Sigma_2^1\;\; \ldots\ldots\ldots\;\; U_{r^1}^1\;\; \Sigma_{r^1}^1) \times \begin{bmatrix} V_1^1 & 0 \ldots\ldots\ldots & 0 \\ 0 & V_2^1 \ldots\ldots\ldots & 0 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 0 & 0 & V_{r^1}^1 \end{bmatrix}$$

If we iterate the process by repartitioning the left hand side of the product and factorise that repeatedly, since the basis of the component waveforms in the spectra is sufficiently small, partitioning is eventually not needed. Thus we arrive after k steps at

$$D = U \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix} (V^k) \times \begin{bmatrix} V_1^{k-1} & 0 & 0 \\ 0 & V_2^{k-1} & \\ 0 & \ddots & V_{r^{k-1}}^{k-1} \end{bmatrix}$$

$$\ldots \begin{bmatrix} V_1^1 & 0 \ldots & 0 \\ 0 & V_2^1 & 0 \\ \cdot & \cdot & \\ 0 & 0 & V_{r^1}^1 \end{bmatrix}$$

Rather than save and bother to multiply the V transform it is easily computed from

$V = (\Sigma)^{-1} U^T D$

This algorithm has been used to factorise a spectral matrix from 143 samples each having 700 readings; scores for calibration were reduced to a matrix of 143 by 30 uncorrelated scores on a Data General Eclipse S140 with 256k bytes of memory. The procedure takes a number of hours but makes it possible for one to envisage its incorporation in software on mini-spectrocomputer systems, because it requires little program space in addition to the well tried procedure by Golub.

REFERENCE

GOLUB, G.H. and REINSCH, C., (1970): Singular value decomposition and least squares solutions, *Numer. Math.*, 14, pp.403-420.

A.E. Stearn,
CSIRO Division of Textile Physics,
Ryde, NSW 2112

PROCEDURE FORMATION NOTATION

We refer to the paper "Procedure Formation – Derivation of Procedures for Users by Users" by Clive Finkelstein in the November 1981 issue of this Journal.

On page 132 under the heading "Stage 4: Procedure Derivation" is a reference to figure 4, and the statement:

"This figure indicates that event A is executed first, *followed by event B then C*" (emphasis added).

The implication here is that event C must follow B, hence that the procedure has a tree structure and cannot be exited from other than the root event, event A.

This is an interesting idea. It appears, however, to be in contradiction to figure 7, shown on page 134, wherein the exit from the procedure occurs from event E40, which is not the root event. It also appears to be in contradiction to the statement:

"Figure 2 indicates that event A is executed first. *Either event B or C* is executed after event A" (emphasis added).

This statement appears in Finkelstein (1981) (figure 2 in this publication is the same as figure 4 in the ACJ article).

Two conflicting notations for procedure maps have apparently been suggested: one involving a tree from which an exit can only be made from the root event after evaluating all branches; the other involving a network of events from which different exits are possible.

S.A. Vitlin,
A. McBurnie,
University Library,
University of New South Wales

Reference:

FINKELSTEIN, C. (1981): Series of articles published in the United States edition of *Computerworld*, issues dated 11 May 1981, 25 May 1981, 1 June 1981, 8 June 1981 and 15 June 1981.

SPECIAL ISSUE

ON PROGRAMMING LANGUAGES

The Australian Computer Journal will publish a special issue on "Programming Languages" in Feburary, 1983. Research papers, tutorial articles and industry case studies on all aspects of the subject will be welcome, and both full papers and short communications will be considered.

Prospective authors should write as soon as possible to:

Professor J.B. Hext,
ACJ Guest Editor,
School of Mathematics and Physics,
Macquarie University,
North Ryde, NSW 2113

to notify him of their intention to submit material for the issue and provide a brief summary of their intended contribution.

In order to allow adequate time for refereeing and editorial review, complete manuscripts will be required no later than 15 September 1982.

Papers should be prepared in accordance with the guidelines published in the May 1980 issue of the Journal. Authors are requested to pay particular regard to the Journal's prefered style for references.

CORPORATE ADVERTISING SCHEME FOR VIDEOTEX

A service announced recently now makes it easier and cheaper to use videotex as a public relations/advertising medium in Australia.

The new service is being offered by Ilmar Taimre Pty Ltd, a Melbourne-based public relations firm, and will be marketed under the name "Profiles". It will operate on the national videotex system launched a few months ago by Computer Power.

Until now, an organisation wishing to publish its material on videotex had to register directly with Computer Power as an "information provider (IP)". However, the minimum space allocation of 100 frames, the initial registration fee of $12,000, and the need to invest in an editing terminal, has tended to discourage individual advertisers whose space requirements are usually small.

In the Profiles scheme, advertisers and other information providers will be able to rent space on the videotex database in multiples of 10 frames, without the requirement to register directly as an IP. In effect, Ilmar Taimre Pty Ltd will act as an "umbrella publisher" for a number of different sub-IPs, as well as providing editorial, frame design, indexing and updating services.

The initial cost of renting a 10 frame section in Profiles is $2,500 per annum, which includes the complete design and creation of the section. Additional lots of 10 frames will cost $2,000 per annum. In other words, Profiles offers potential IPs a relatively inexpensive, low-risk, fast and easy way to disseminate information on videotex.

At this stage, Profiles will cater for three main categories of information provider — business, government, and non-profit organisations. Because the main users of videotex are in the business and government sectors, information such as the latest press releases, financial results, chairman's statement and corporate background is particularly suited to the Profiles/videotex combination.

Further details of the Profiles service are included in a report "Corporate Advertising Opportunities on Videotex" published by Ilmar Taimre Pty Ltd. The report is available on request — telephone (03) 876-3950.

OTC UPGRADES MIDAS

In April 1982 the Overseas Telecommunications Commission replaced its original Midas node located in Sydney with a new Tymnet 'Engine' which will enable new features to be added to the service with relatively small effort.

OTC believes it will be able to respond more quickly to market demands in a field where technological developments demand adaptability.

The original processor was installed three years ago using equipment also manufactured by Tymnet. But technological developments and market demands have meant that the existing processor cannot provide all the facilities that existing and potential users of Midas, OTC's packet switching data transmission service, now require.

A major limitation of the present Midas Service is the inability of Australian Data Bases (Hosts) to be accessed from overseas data networks. The consequence of this limitation has been a one-way flow of information into Australia, and an inability of Australian information venders to offer services internationally.

By mid-1982 the Midas service will be able to connect Australian synchronous data terminal equipment (DTE) by tie-line using the CCITT X25 LAP B interface at speeds of up to 9600bps thus enabling the establishment of calls from overseas to Australian host computers.

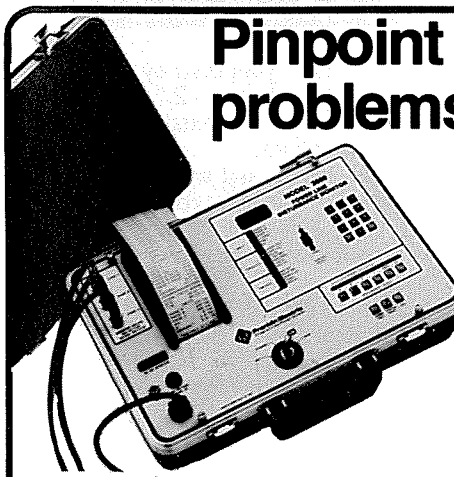
Facilities will also be provided to connect hose computers to Midas via tie-lines operating asynchronously at speeds up to 1200bps thus enabling calls to be directed to these computers. Further development will enable calls to be estabished both into and from Australian asynchronous DTE's at speeds up to 1200bps.

MANAGE YOUR PRACTICE WITH CARE!

"Within five years 80 per cent of the 25,000 medical practices in Australia will be equipped with computers." This is the prediction of Maurie Stang, the Marketing Director of Regional Data Systems who last October released Care, a powerful computerised system for medical practice management.

Already installed in a number of practices, Care offers a unique total solution to the requirements of practice management. It features full open-item accounting that includes instant or deferred billing and the production of monthly accounts, daily banking and the production of bank deposit slips and comprehensive practice management reports. For patient management it provides a powerful patient detail retrieval, the history of all previous services given to the patient and patient registration for private patients, pensioners, repatriation patients and those for

Pinpoint power problems . . .



fast!

Portable Model 3600 Powerline Disturbance Monitor can track three AC phases and one DC Channel simultaneously, for fluctuations likely tocause problems in sensitive equipment. Push button keyboard for programming parameters . . LED display . . Large thermal print out tape. . RS232 interface . . all mounted in shock protected briefcase.

Franklin Electric
Victoria Tel: (03) 792 9431. Telex: 37416
N.S.W. Tel: (02) 439 6633 Qld. Tel: (07) 524 059 FE 002

NOW! FROM SANYO

CRX1000: ONLY $979*

- High-resolution display of up to 1920 (8 x 12 dot) characters, 80 per line in 24 lines, on a 12" screen.
- Extra simplicity, extra low cost with incorporation of a microprocessor and LSI modules.
- ASCII keyboard with typewriter-style shift and shift-lock keys for capital letters and blocktype arrangement for easy operation. Teletypewriter key format for alphanumerics, separate numeric pad.
- External interface employs EIA RS-232C interface and 20mA current loop.
- Hard copy interface on RS-232C AUX port output enables print-out of data from host computer.
- Designed for easy servicing with function-related modular construction and keyboard check system for memories and data transfer functions.
- Audible alarm when control is received or 72nd character of a line. Also warns of depletion of paper supply when optional printer is employed.
- Compact unit measures 410 (W) x 555 (D) x 320 (H) mm and weighs only 17.5 kg including keyboard.

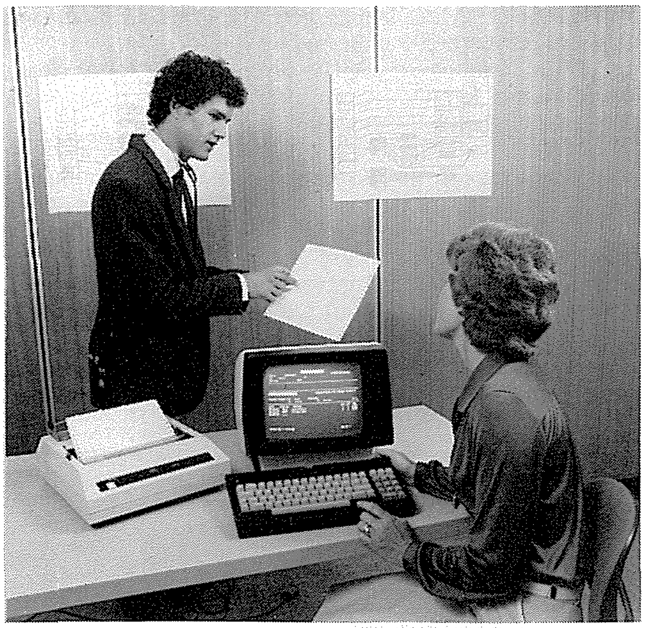
EMULATIONS: ADDS RANGE ** VISUAL 100, 200 ** ETC.
Dealer enquiries welcome

SANYO

SANYO OFFICE MACHINES PTY. LTD.
127 Walker St., North Sydney 2000. Tel: 929 4644
VIC: 67 5501; SA: 51 3946; ACT: 80 5854; BRIS: 36 7588;
DARWIN: 81 5794.

*Plus tax if applicable. **Additional $120 per chip.

SOM-229


The Care computerised system for medical practice management.

whom accounts are sent to employers, insurance companies or the Health Department. Care has the ability to have one or more users billing, receipting, enquiring or processing reports at one time.

Regional Data Systems was formed in 1981 by the joining together of Dalton Sallis and Associates, a firm of leading consultants in the Computer and Management field, and the Regional Medical Group, an Australia-wide organisation specialising in health care. The company employs seven top people all selected for their knowledge and years of experience and the technical team is headed by David Dalton and Dr. Philip Sallis. Dr. Sallis is a well-known consultant and lecturer in Management Information Systems and David Dalton was until recently National Software Manager for Prime Computers and has had more than ten years experience in the industry.

Care is written in Cobol and runs on the Onyx range of computers. Onyx was chosen because the computers will run in a reasonable office environment, because they have built a Winchester sealed fixed disk and a built in tape for back up and because they can be expanded to allow the system to be upgraded as the practice grows and the number of doctors increases.

In the future the company sees its computers hooked up to a national data network in which a central data bank will provide in-depth medical data to the General Practitioner and the Practitioner through his computer will provide to the data bank information on such matters as the prevalence in his area of specific illnesses or allergies.

Maurie Stang is excited by the prospect of these developments and believes the day will soon come when it will be commonplace for a doctor to have a computer.

Maurie says, "in future years a major part of the Regional Group of activities will be in the growing areas of practice management systems and installation of the integrated electronic office concept into health care."

CONTROL DATA HOSTS CHINESE DELEGATION

Recent visitors to Australia were members of the Computer Application and Management Study Group of the State Scientific and Technological Commission of the People's Republic of China.

The group, pictured here outside Control Data's Australian Computing Centre at Knox, Victoria, was headed by Mr. Su Zhengwu (centre). They were hosted on a tour of the Australian Computing Centre by Mr. Rob Hain (right), CDA's Regional Manager for Computer Systems.

The party met with various organisations, including ACS, during their study tour, which was planned to assist their investigations into applications, networks, software development, computer centre management and training for technical personnel, and were especially interested in

QEGB

TECHNICIANS

$15,722 to $18,644

GLADSTONE POWER STATION

Applications are invited from suitably qualified and experienced persons for appointment to the above positions located at the Gladstone Power Station.

RESPONSIBILITIES: The appointees will be responsible for:—

- Fault finding and maintenance of boiler and turbine instrumentation and controls.
- Fault finding and maintenance of computer systems.
- Repair of electronic control cards and modules.

QUALIFICATIONS: Applicants must hold a Queensland Electrical Workers' and Contractors' Board Certificate of Competency as an Electrical Fitter Mechanic and a Certificate, Associate Diploma or Diploma in Electrical Engineering or equivalent from a College of Advanced Education or other technical qualifications which, in the opinion of the Board, are equivalent in standard or a combination of experience and qualifications which the Board deems to be acceptable.

Experience in instrumentation, analogue and digital controls and computers would be an advantage.

CONDITIONS: Salary will be negotiable up to $18,644 depending on qualifications and experience. Relocation expenses and housing assistance where applicable will be discussed at interview. General conditions are progressive and include a nine day fortnight.

TO APPLY: Application Forms and further information may be obtained by contacting the Personnel Office (079) 761203. Applications should be lodged with The Personnel Officer, Gladstone Power Station, P.O. Box 1108, Gladstone, Q. 4680.

the centralised facilities and equipment displayed at the Knox centre.

The group was also provided with a demonstration of the Plato system for a computer-based education (CBE) at Knox, and expressed a keen interest in the possibilities for such a system in China.

AUSTRALIAN MARKET EAGER FOR ITT COLOUR TERMINALS

A new four-colour visual display terminal supplied by STC's Computer Product Division, has been released in the Australian market.

Designed for compatibility with the larger IBM computers, the 2790 first landed here in December. By the end of March, close to 200 units had been sold.

The four-colour display presents data more clearly and concisely on the screen, and so makes it easier for operators to absorb information.

Colour is generated by the existing field attribute byte, and makes no demand on mainframe computer memory.

The ITT Courier system operates through advanced terminal controllers (ACTs) in two sizes – for up to sixteen, or up to 32 attached terminals. Models of the larger controllers offer 100 per cent redundancy.

The 2790-2A terminals offer 1920-character display on a 20cm by 27cm high-resolution screen.

CSIRO

RESEARCH & DEVELOPMENT

GROUP LEADER

COMPUTER SYSTEMS DEVELOPMENT
$39,850 — $45,500 p.a.

DIVISION OF COMPUTING RESEARCH, CANBERRA ACT

The Division of Computing Research operates an Australia-wide computer network to provide a computing service to CSIRO, government and other users throughout the country. Major host computers include a CDC Cyber 76, a CDC 730, a FACOM M-190, two FACOM M-150's and a two Pi V32, and services are offered using the CDC operating systems SCOPE 2 and NOS/BE. FACOM OSIV/54 and IBM VM370/CMS. Host computers are connected to each other and to the long distance packet-switched network by a local high speed contention network using NSC's HYPERchannel.

In addition to its service role, the Division undertakes innovative research and development in: Operating Systems, Communications, Image Processing and Graphics, Systems Modelling, Data Base Systems and VLSI architecture.

As leader of a Research & Development Group, the appointee will assist the Chief in the guidance and co-ordination of various research and development projects. The appointee will be expected to take a leading role in the formulation, initiation and conduct of research within selected projects.

Applicants should have a PhD degree or equivalent in computing science or a related field, a substantial record of research achievement and the ability to provide scientific leadership to a number of research groups. Extensive experience in communications and systems software development would be an advantage.

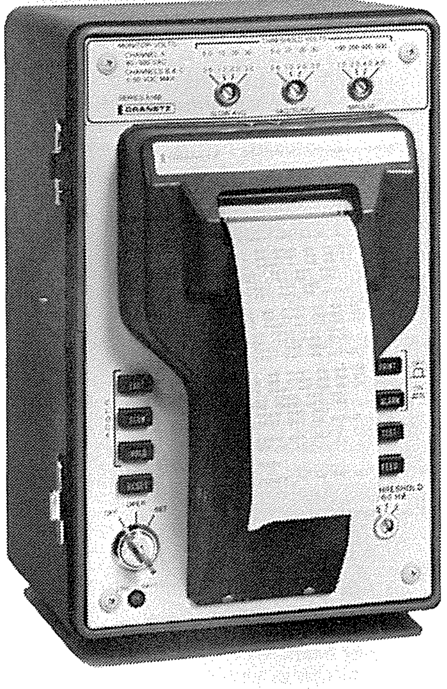
This is an indefinite appointment which carries Australian Government superannuation benefits, subject to normal conditions.

Dr. P. J. Claringbold, Chief, Division of Computing Research would be pleased to discuss the position with potential applicants and receive advice concerning this appointment from people with a particular interest in it.

Applications in writing and stating full personal and professional details with the names of at least three referees, quoting reference No. A1977 should reach:

The Chief
Division of Computing Research
CSIRO
PO Box 1800
CANBERRA CITY ACT 2601

by two weeks after publication date.



POOR POWER LINE QUALITY... how much does it cost you?

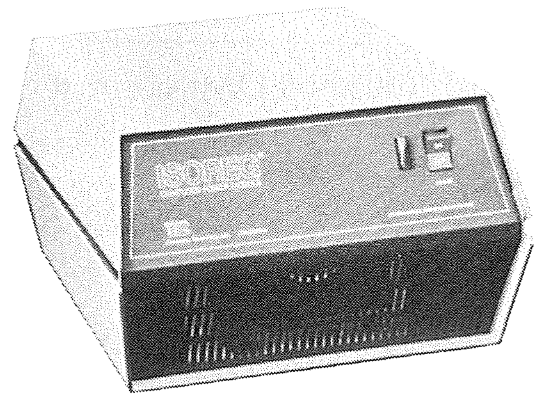
When the quality of mains power deteriorates, so does system performance. Sags, surges, drifts and spikes can cause errors, loss of data, equipment damage and system shutdowns. Often, the diagnosis is wrong and the system is thought to be faulty.

The Dranetz Power Line Disturbance Analyser detects and records power line conditions and tells you what you need to know to select the right power conditioning for your installation.

Isoreg power line conditioning modules...

... can eliminate costly down-time and errors caused by mains disturbances.

Put these power problems behind you by contacting the sales staff of Electrical Equipment's Industrial Division for details.



Electrical Equipment Limited

Industrial Division

Sydney 597-1155 Melbourne 555-1611 Brisbane 44-4801 Perth 381-2866
Agents: Adelaide 51-6718 Hobart 34-2233

The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

SUBSCRIPTIONS: The annual subscription is $20.00. All subscriptions to the Journal are payable in advance and should be sent *(in Australian currency)* to the Australian Computer Society Inc., PO Box N26, Grosvenor Street, Sydney, 2000. A subscription form may be found below.

PRICE TO NON-MEMBERS: There are now four issues per annum. The price of individual copies of back issues still available is $2.00. Some already out of print. Issues for the current year are available at $5.00 per copy. All of these may be obtained from the National Secretariat, PO Box N26, Grosvenor Street, Sydney, 2000. No trade discounts are given, and agents should recover their own handling charges.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining partway through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $2.00 per copy. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly of any change of address.

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: PO Box 446, Canberra City, ACT, 2601. NSW: Science House, 35-43 Clarence St, Sydney, NSW, 2000. Qld: Box 1484, GPO, Brisbane, Qld, 4001. SA: Box 2423, GPO, Adelaide, SA, 5001. WA: Box F320, GPO, Perth, WA, 6001. Vic: PO Box 98, East Melbourne, Vic, 3002. Tas: PO Box 216, Sandy Bay, Tas, 7005.

AUSTRALIAN COMPUTER JOURNAL

Subscription/Change of Address Form

Name. . . . .

Current Address. . . . .

. . . .

☐ Please enrol me as subscriber for 1982, I enclose $20.00.

☐ Please record my new address as shown above. I attach below the mailing label for the last received issue.

ATTACH LABEL HERE

Send all correspondence regarding subscriptions to PO Box N26, Grosvenor Street, Sydney, 2000, Australia. Photocopies of this form acceptable.

CONTRIBUTIONS: All material for publication should be sent to: Editor, Australian Computer Journal, PO Box N26, Grosvenor Street, Sydney, 2000. Prospective authors may wish to consult manuscript preparation guidelines published in the May 1980 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. Descriptions of new software packages are also published to facilitate free distribution.

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 2.5cm 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, Keyword 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	Service
ACM	Bibliography and Subject Index of Current Computing Literature.
ACM	Computing Reviews.
AMS	Mathematical Reviews.
CSA	Computer and Information Systems Abstracts. Data Processing Digest.
ENGINEERING INDEX INC.	Engineering Index.
INSPEC	Computer and Control Abstracts.
INSPEC	Electrical and Electronic Abstracts.
ISI	Current Contents/CompuMath
ISI	CompuMath Citation Index.
SPRINGER-VERLAG	Zentralblatt fur Mathematick und ihre Grenzgebiete.

Copyright © 1982. Australian Computer Society Inc.

Production Management: Associated Business Publications, Room 104, 3 Smail Street, Ultimo, NSW 2007 (PO Box 440, Broadway, NSW 2007). Tel: 212-2780, 212-3780.
All advertising enquiries should be referred to the above address.

Printed by: Publicity Press (NSW) Pty Ltd, 66 O'Riordan Street, Alexandria, NSW 2015.