ISSN 0105-8517

Proceedings of the 1978 Aarhus Workshop on Software Engineering

Software Engineering: Tools and Methods

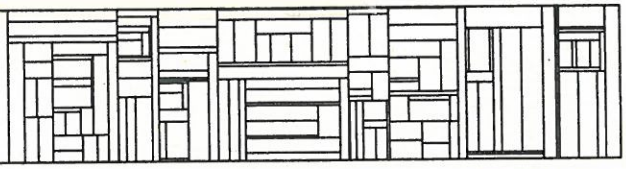
Edited by:

Nigel Derrett
Karen Møller
Mike Spier
J. Michael Bennett

Daimi PB-125
August 1980

This work has been supported by the Danish Natural Science Research Council Grant No. 511-8508.

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



This is the record of a meeting held at the University of Aarhus in Denmark, 23-26 May 1978. The meeting was a workshop concerned with Software Engineering. A mixed group of Computer Scientists met for 4 days with the intention of exchanging knowledge and the hope of becoming wiser. Herein is a small glint of their wisdom.

The workshop was organised as a mixture of formal presentations and discussions. The organisers had given titles to the various sessions but these were only used as guidelines, and the discussions in particular ranged far and wide, sometimes heated, and sometimes rather undisciplined.

I have reorganised the order of the presentations and the discussions in an attempt to introduce a more logical structure.

The workshop was held under the auspices of the Computer Science Department, Mathematics Institute, Aarhus University, with financial support from the University and from the Danish Natural Science Research Council, grant no. 511-8508.

Nigel Derrett

The 1978 Aarhus Workshop on Software Engineering

Software Engineering: Tools & Methods

Aarhus, 23-26 May, 1978

List of Participants

Robert S. Barton
Burroughs Corporation
USA

E. Henry Beitz
Computer Consultant
USA

J. Michael Bennett
Department of Computer Science
University of Western Ontario
Canada

Dr. Klaus Berkling
Institute for Infosystems Research
GMD, Bonn
West Germany

Becky J. Clark
Burroughs Corporation
USA

Nigel Derrett
Computer Science Department
Aarhus University
Denmark

Robert J. Flynn
Department of Mathematics
Polytechnic Institute of New York
USA

Janusz Górski
Institute of Informatics
Technical University of Gdansk
Poland

Chr. Gram
Department of Computer Science
Technical University of Denmark
Denmark

Herbert R. J. Grosch
President, ACM
USA

Søren Lauesen
Computer Science Department
University of Copenhagen
Denmark

John A.N. Lee
Department of Computer Science
Virginia Polytechnic Institute
USA

Peter Marks
Digital Equipment Corporation
USA

Peter Møller-Nielsen
Computer Science Department
Aarhus University
Denmark

Elliott I. Organick
University of Utah
USA

Graham Pratten
International Computers Ltd.
England

Nick Shelness
Department of Computer Science
University of Edinburgh
Scotland

Mike Spier
Computer Science Department
Aarhus University
Denmark

Joy Stoy
Oxford University Computing Laboratory
Programming Research Group
England

Ib Holm Sørensen
Computer Science Department
Aarhus University
Denmark

Contents

SESSION F	SOFTWARE MANAGEMENT AND ECONOMICS	
F.1	G. Pratten	Developing Large Systems
F.2	R.J. Flynn	Goal-Driven Software Engineering
SESSION G	HARDWARE FOR SOFTWARE	
G.1	R.S. Barton	A System Based on Functional Programming
G.2	K. Berkling	Computer Architecture for Correct Programming
SESSION H	WHAT HAVE WE LEARNED SINCE ...	
H.1	M.J. Spier	A Personal Peek into the Future

The 1978 Aarhus Workshop on Software Engineering

Some Food for Thought

Mike Spier
Nigel Derrett

The title of this workshop is "*Software Engineering: Tools & Methods*". The workshop is a forum for discussing the state of software engineering today: experiences gained, insights gleaned, measurements made. We hope that the presentations and discussions will demonstrate where we are, and where we are headed, professionally.

As the title of the workshop indicates, we want to focus attention on *industrial* programming: the Tools and Methods employed in producing software products, the effects of using these techniques, and recommendations for future technology.

This document sketches certain topics which we consider to be important, and which we would like to hear discussed. There are, of course, many others.

7 November, 1977
Aarhus University

1. Concerning Classification

"Software Engineering" is the antithesis of "The Art of Programming". Given the importance of computers and computer programs in our society today, it is desirable that we understand the practice of industrial programming. Making programs professionally should be an engineering discipline whose practicality is equivalent to that of other engineering disciplines.

An important feature of other engineering disciplines is that they use general scientific knowledge to devise specific technical solutions to known practical problems (*cooking recipes*). We need to acquire similar techniques in our branch of engineering — by identifying classes of problems, and developing tools to deal with them.

Essential to this goal is the identification and enumeration of things as they are, and of their importance and their possible inter–dependencies. The realities of the programming industry are a far cry from the elegant models promulgated by our *savants*; indeed much of reality was never admitted into the models because of its vulgarity. Perhaps, instead of ignoring uncouth subjects we should treat them as classifiable "natural phenomena". Then, having taken stock of the situation, we may manage to deal with it.

Classification of Virtues

We should be able to specify what the properties of "good" software are. A discussion on this topic should take place on a less trivial level than that of whether or not *GOTO* statements should be allowed in programs. Here are some features of software "goodness":

- *Being maintainable* — Writing a clean program is one thing. Keeping it clean throughout a long lifetime, during which it is continually being modified, is another.

- *Being fault tolerant* — Software products should not crash upon the slightest provocation. They should not crash at all! They should produce sensible diagnostic or error messages, and they should contain procedures for meaningful continuation after failure.

- *Being friendly to users* — It is not easy to make software products which interact with people in a sensible way. For example it may be necessary to propagate response signals up through successive levels of interpretation, and then to present them to the user in terms which are meaningful to him.

- *Being host system independent* — The useful life of a commercial application program typically is longer than the useful (or cost effective) life of the computer on which it was first implemented. The cost of transferring the program to a new machine may be so great that it outweighs that savings which using the new machine could give.

- *Being documented* — There may be documentation specifying what the program should do; documentation specifying what the program does, how it works, how to use it, how to modify it; documentation of the program's history, and of its planned future. There may be user manuals, system manuals, training manuals; and more. On the other hand, there may be none of these.

- *Being complete and self consistent* — How do we incorporate all of the relevant functions into the product? More importantly, how do we restrain ourselves from incorporating miscellaneous "features" or "options"?

- *Being correct and reliable* — How do we write programs so that we can be sure that they do what they were intended to do? How do we modify extant programs so that we can be sure that they have the new intended behavior?

- *Being cheap* — Making software products is extremely expensive. Maintaining present products is even more expensive.

We would like to see a list containing further features of software "goodness". More important, we would like to see each virtue given a precise context, and a set of criteria for the evaluation of its "goodness".

Classification of Activities

There are many branches of engineering — civil, electronic, nautical, and so on. Their practitioners often use common techniques to solve their problems, using the same slide rules and drawing boards. Nevertheless, the differences between these engineering branches is obvious.

We have tended to regard "Software Engineering" as a monolithic subject, because its practitioners go through the motions of programming. Upon closer inspection it is doubtful whether the professional who makes a high–precision floating point library, and the one who makes an inventory control system, have much in common.

It is important to recognise and clearly identify the various branches of software engineering, and to consider the distinct requirements of each of them.

Here is a list of some programming activities which get very little attention in the literature.

- *Program Maintenance* — Large numbers of programmers spend all of their time maintaining and modifying programs. They may not be able to rewrite these programs; often they are explicitly forbidden from doing so. Out of sheer need to survive, these programmers have developed some very sophisticated tools and procedures.

- *Product Customisation* — Commercial programs are often designed to be used by many customers, no two of whom have identical systems or requirements. The problems of tailoring systems for the individual users are considerable.

- *Diagnostic Programming* — This is the production of test programs for hardware or software systems. The hardware test procedures themselves are firmly rooted in known combinatory logic, but the arrangement of sequences of tests into larger test packages is an obscure art. The art of making test procedures for software systems appears to be closely related to Astrology.

- *Customer support* — When the system is obviously jinxed during production run, and the Field Service Engineer insists that the hardware functions perfectly, we urgently summon our friendly Software Support Specialist.

- *Quality Assurance* — Also known as "Release Engineering", quality assurance is the profession of certifying that programs work properly, before they are released. It involves looking for bugs, and checking whether programs do what their authors claim they do. In the case of a modified program, "regression testing" is called for: checking whether the (supposedly) untouched parts of the program still work in the way that they used to.

We have listed some branches of software engineering which are important because they have to do with real–life problems. The list is far from complete. These branches comprise a large percentage of all the programming being done. By treating them as respectable disciplines we may help to develope the techniques required by each.

Classification of Tools

Most engineering tools are well designed precision instruments. The programmer's toolkit is a pitiful collection of odds–and–ends. Inadequate to the task, unmatched with one another, of doubtful origin, and lacking most of the above enumerated virtues, these are the means with which quality programs are supposed to be crafted. Here is a mixed bag of some of the tools available to software engineers:

- *Language Processor* — It is one thing to define a glorious new language. It is another thing to write a decent compiler.

- *Text Editor* — Interactive programming is becoming increasingly common. The interactive programmer spends half of his time talking to his text editor. Do you know of a decent text editing language?

- *Debugger* — The debugger should be thought of as an "un–compiler". We would like our debugger to talk back to us in the language we wrote our program in; the rascal typically talks back in octal or hexadecimal.

- *Macro Processor* — Macro processing and conditional compilation are among the system programmer's most potent tools. Not enough thought is given to their design.

- *Job Control Processor* — The job control processor is the means by which we instruct (or request) the computer to do work for us. Job control languages are usually illogical and incomprehensible.

- *Command Processor* — This is the on–line version of the job control processor. It does not need to have the fancy flow–of–control facilities of its brother, but it needs to be user–friendly.

- *System Generator* — Those of us who have ever had the misfortune of performing a SYSGEN, know! Let them who are innocent of such deeds, be spared the knowledge.

- *Interactive Interpreter* — The interactive programming language interpreter, *viz.* BASIC or APL, has great appeal as the compleat programmer's tool. But the service features (editing, filing, debugging etc.) in existing interpreters are impossible to use, and the problem of efficiency during production runs cannot be dismissed.

2. Concerning Languages

Tools have to be wielded. In our profession this is done by talking to them in some language. The most universal tools available to us are the general–purpose computing machines, with which we converse in "programming languages" (high level languages, assembly language, machine code). It is an important feature of these tools that the languages are first designed in the abstract; and these are (in the opinions of their designers, at least) clean and coherent in syntax and semantics. The tools themselves (compilers, assemblers, interpreters, hardware) are then built in order to implement these languages; and the design of the tools is subsidiary to, and entirely dependent on, the design of the languages.

However in the case of more specialised tools (such as the text editor, debugger, macro processor, job control program, etc.) the order of events is reversed: A tool is designed for a specific purpose; it needs to be talked to in some language, but little thought is given to the design of that language. Indeed it is remarkable how little of the research done on "high level languages" is applied to these specialised languages. As the tool acquires extra features, the language becomes more and more complicated and muddled. The tool is essential; it can only be wielded by using the language; and the language is a mess. How can we expect the tool to be wielded in a workmanlike manner?

Cases are known where only a chosen few wizards can safely negotiate some linguistic maze. In time, they acquire the stature of thaumaturgists whose job security is made invulnerable by the sinister Voodoo that they practice.

3. Concerning Measurement

Assuming that we have observed and identified various facts and practices, it is not unreasonable to inquire into their relative importance. Software engineering is a multi–billion dollar industry, and money could be a very convincing measure of such importance. "Performance" or "efficiency" are other measures, although recent technological and economical developments have substantially unbalanced traditional standards. Yet other known measures are "man–month", "lines of code per day", "installed customer base", "number of bugs reported", etc.

It may prove invaluable to have a classification of identified subjects (a measure of goodness, a technique, a tool, a professional branch), properly quantified by measured facts. For example, we suspect that program modification is a more prevalent programming activity than original program coding. If we knew for a fact what the importance of modification is, and the reason for that importance, we would then be in a better position to improve upon the state of the art.

Also, we may solve the wrong problems and not know it, or else solve the right problem in an unpracticable manner. We may be constructing elegant jeweller's tools, when the man in the field is using his improvised sledge hammer for lack of the sophisticated power–assisted contraption that we should have provided in the first place. Or else we may be overhauling the entire engine when a judiciously administered drop of oil would have sufficed.

And talking of facts: these must be obtained; the art of measurement is rather infantile, and further confused by the difficulty of identifying the thing that is to be measured.

4. The Problem Revisited

We all agree that there needs be a methodology for the predictable, efficient production of good software. Considering how little we know about the discipline and the tools, it is not surprising that we cannot make sensible pronouncements about the methods.

Software engineering – as distinct from the art of making elegant little algorithms – is the discipline of coping with massive, chaotic and irrational complexity. When we program the computer to emulate an elegant mathematical relation, we can easily demonstrate the superiority of *DO WHILE* over *GOTO*. When we program our computer to emulate our income tax laws, it is unclear what, if anything, could be demonstrated about the code.

Another interesting point is that mathematical relations, being invariant, can be reprogrammed and refined for increased elegance or coherence or efficiency. The tax laws, on the other hand, are not at all invariant. They are guaranteed to change from year to year, and the changes are not necessarily insular but quite possibly have subtle consequences. And the reprogrammed system will not acquire elegance or coherence or efficiency; we consider ourselves lucky if it simply fails to acquire too many bugs.

For certain classes of problems, it is thus doubtful whether there should be any programming language at all, in the sense of ALGOL, BASIC, COBOL or FORTRAN. Alternative ways of specification, ranging from RPG through various pidgin–English application generators to the filling out of pre–printed forms, are in widespread commercial use. These approaches are little understood, and almost universally ignored by the more "serious" practitioners.

We are thus suggesting that there exist ample opportunities for exhaustive field work, in recognising and identifying real–life problems; and for devising honest workable solutions. We can see three important outcomes to such work:

- *Formalisms* — The cleanup and formalisation of those non–languages which today comprise the programmer's toolkit; or possibly the elimination of them by substituting a new, well conceived toolkit.

- *Methodologies* — Clear, unambiguous and foolproof "cooking recipes" custom tailored for specific situations.

- *Legitimacy* — Incorporation of Software Engineering into the body of "respectable" computer science. Possibly one reason for our difficulty is that when the creative young student is doing his studies he is learning to solve one class of problems; by the time he encounters the other classes of problems on the job, he may not have the time, or resources, or intellectual environment for a proper solution.

SESSION A

Program Goodness

What is program "goodness"? Is there a real distinction between "goodness" as perceived by the programmer, and "goodness" as perceived by the user? Is goodness a fixed quality, or does it deteriorate in time with the product's evolution, or does it increase as the product stabilises? How do we measure goodness? How do we establish its presence or absence?

Defensive Programming

Mike Spier

DEFENSIVE PROGRAMMING *

Mike Spier
Aarhus University

This paper addresses the subject of program robustness, which I see to be antithetic to the mathematician-inspired notion of program elegance. Too often is such elegance achieved through excessive reliance on unrealistic assumptions. Experience has shown program robustness to be directly proportionate to the amount of suspicious verification of assumptions coded into it. Whether or not we are doing ourselves a favour in fostering and encouraging the present fashion of elegance, and in condemning robust programs for vulgarity, is an interesting subject for contemplation.

---

* The text of this paper is directly copied from the notes for a course in software engineering, given during the fall '77-'78 semester. It is, however, reasonably independent of all other course notes. I hope that the reader will excuse the paper's somewhat loose style.

DEFENSIVE PROGRAMMING

There is a school of thought, according to which the elegance and clarity and goodness of code are - among others - evidenced by its lack of redundancy. It is thought that to write terse (or, colloquially, "tight") code is the mark of the expert programmer; that there is no elegance to code containing logical or algorithmic redundancy.

I dispute this. I believe that solid and reliable code has a different kind of beauty and elegance; by analogy, a passenger-bus or a truck can be as elegant in their way as a sports-car is in its. To make a truck that lives up to criteria of sports-car æsthetics is not elegant - merely foolish. As software engineers we are more often called upon to make a reliable "truck" than we are to make a fast "sports-car".

If we analyse samples of code proclaimed to be elegant, we will often find that this terse elegance was achieved through too heavy a reliance on assumptions. But such assumptions - appealing as they may be on paper - may not at all be so reliable in reality; or else they may be so complex or subtle that some other programmer, modifying the code in the future, will not be aware of them and failing to recognise and understand the assumptions will innocently violate them and thus destroy the code.

This tendency to rely on assumptions is something that we (that is, we programmers) have inherited from the mathematician. In mathematics, if there is a boolean variable whose value is known not to be true, then its value is known,absolutely, to be false. And in programming, we naturally follow similar reasoning

```
if boolvar then <condition is true>
           else <condition is false>
```

But there is a fundamental difference between the mathematician's boolean variable, and the programmer's. The programmer's boolean variable has a material existence and is boolean only under suitable interpretation. Typically it is a computer memory word, under an interpretation such as

integer(true) = 1
integer(false) = 0

Now consider the following code

if boolvar then <condition is true>
else if not boolvar then <condition is false>
else abort(boolvar)

where directive abort is defined to display the value of its argument in some universally meaningful representation (e.g., octal), after which it displays the program's state and stops the program's execution.

The above sequence of code is obviously illogical, mathematically. It may none the less be perfectly realistic, possibly resulting in an abort printout of boolvar

005203717002

[Whether or not such a condition may ever arise is very much dependent on one's compiler's implementation. The proper generated code should test for boolvar's truth value by testing the single lowest significant bit. Under this arrangement the value of boolvar could never be paradoxical, i.e.,

boolvar $\neq$ true $\cap$ boolvar $\neq$ false

But on some machines the only means of testing that bit may be by (1) copying the value elsewhere, (2) masking off all other bits in the word containing the copied value, and (3) testing the word for integer values 0 or 1 (or some other circuitous method). Under such circumstances, the efficiency-motivated compiler writer may choose to test for integers 0 or 1 directly, or to test for a zero/non-zero distinction, and then paradoxes of the kind just mentioned are possible.]

There are, of course, other well known examples of such potential problems:

- using an unchecked index value when accessing an element of an array, risking an out-of-bounds reference.
- using an unchecked index value in a case statement without having provided a default case.
- assuming the value of a character to be something that can always be input, or output, safely (where the possibility exists for a special character to be interpreted by some device as a function code that turns the device on or off, etc.)*

Obviously all these potential problems could be eliminated with additional code that verifies (and in the process of verifying, explicitly documents!) our assumption. For example:

```
manifest  size = 100;
let table = vec size;
      :
if (i ge 0 ∩ i le size) then access (table! i)
      else abort      // index i out of bounds
```

Just as obviously, to do so systematically for all assumptions would result in unnecessarily tedious and meaningless code. Still, not to do it at all is a very bad programming practice, for to build one's program upon unverified assumptions is a very foolish thing to do. In other areas of our life, the thing analogous to writing terse code based on agreeable assumptions would be, for example: having houses without a fire escape, having cars without seat belts and with a single brake system only, having ships without lifeboats, or having an electric circuit without a fuse.

This last analogy deserves closer attention. Complementary to our profession of software engineering is the profession of hardware engineering. There the identification of "system", "module" and "value" is naturally

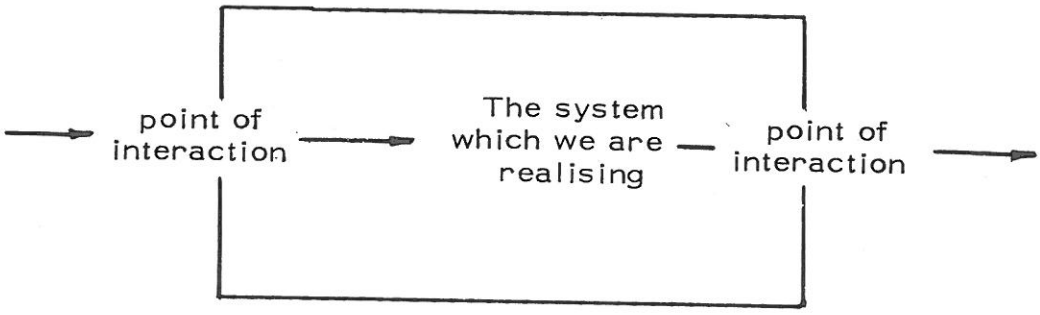
---

* For a local example, inputing a control-P into RECAU's terminal system causes the line to be disconnected.

obvious, whereas in our case we need an extensive body of abstract reasoning plus analogies and examples (see part 1, principles) to obtain a corresponding identification. The hardware engineer has various physical modules (e.g., IC's , resistors, capacitors, switches, wires, etc.) which he puts together into a hardware system. The system's behaviour is the pattern of electrical impulses going through the component devices. The hardware engineer makes two major assumptions, (1) that the physical components behave as specified, and (2) that the electrical current used is within very narrow bounds of voltage, rectitude and amperage. There is little he can do about the first assumption: if the devices do not work, he has to get others. But the second assumption is controllable and verifiable: using transformers, rectifiers, voltage regulators and fuses, he makes certain of having his system operating under the proper electrical conditions, or else of not operating at all.

I propose that we make it an elementary discipline to apply similar common sense at the software level.

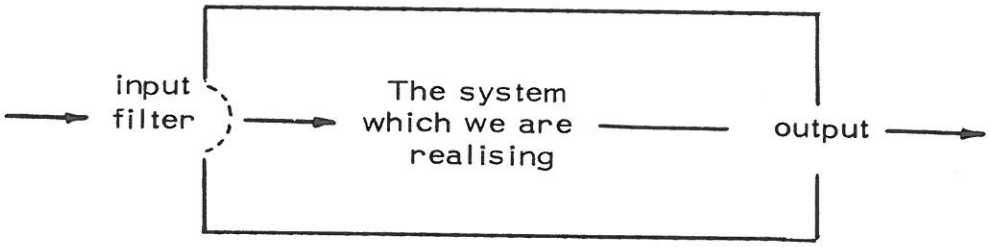
Let us draw the distinction separating the inside of the system we are realising, from its outside. Let us also suitably indicate the points of interaction between our system and the rest of the universe.



We only have to worry about those points of interaction where our system's behaviour is being influenced from the outside, because according to the principle of localisation of knowledge it is perfectly reasonable to let

the rest of the universe worry about the ways it is influenced by the output from our system.

So as we consider the state of the world from the local perspective of our system, we recognise that even though all external assumptions are beyond our control, they need not be beyond our ability of verification. Let us construct filters (or "firewalls") to stop all undesirable influences:



What is such a filter? It is code to verify as many as possible of the assumptions that we are making about the influences coming from the outside. For example:

- that an actual call made to a gate of our system contains the proper number of input and output arguments (see section 1.IX: Usefulness of the independent module, for further technical details).
- that data items supplied from the outside are of the proper data type: i.e., that a boolean is indeed a true or false value; or that the floating point number, or the array of pointers, or the character string, etc., indeed are just that.
- that a data item of the proper type also has the proper value, if there are further restrictions on value: e.g., that an integer variable can assume values $\{0..6, 27, 9999\}$; or that a character string variable must contain a legal file name, etc.
- that a pointer variable supplied from the outside for the purpose of indirect reference from inside our system indeed points to some reasonable place.

How do we code such a filter? Very carefully! Two aspects have to be considered,

- the logic of the filter algorithm, and
- its existential conditions.

The logic of the filter algorithm may or may not be obvious, and there may be serious restrictions on what is or is not possible. These depend very much on the language used, and on the specific language processor implementation technicalities. There are languages and runtime systems where it is possible to obtain specifics about the variable type, but in many cases it is not possible. It may be easier to check the value range of a boolean variable than it is to check the validity of the value of a file-name string variable. However, we should check as best we can. To apply circuitous reasoning, by knowing that assumptions should be checked, we could preventively design our system or module to have an interface that is more easily verifiable. If we have any influence upon our language processor, it could be modified - if necessary - to provide a run-time reflection of the type of variables; the information is easily enough obtainable from the symbol table.

The existential conditions for the filter code are twofold. First, the file code must be properly segregated and identified, textually, so that the reader of the code is not led into confusion when trying to understand what the program is really supposed to do. The filter code may well be thought of as "logical noise". I strongly advocate putting it into the program to give the program the robustness necessary to resist various potential error conditions; however, I also strongly advocate the removal or segregation of all gratuitous code, for the sake of lucidity. Some compromise must be found to accomodate these conflicting goals. I suggest to use the procedure mechanism. You define a procedure to do the argument validation, and you clearly invoke it before entering the program's essential logic. For example:

```
SYSTEM
let validate([I1,I2,...,Ii,Ii+1,...,Ii+o],[ ]) be
        { < argument validation logic > }

modname: GATE([I1,I2,...,Ii],[O1,O2,...,Oo])
        validate([I1,I2,...,Ii,O1,O2,...,Oo],[ ])*

        <Essential logic of program >

ENDSYSTEM
```

The second existential condition of the filter code is that it must absolutely not influence the program's essential logic. The filter code must be completely neutral, behaviourally. This means that the filter code (1) must not write into any of the argument variables, and (2) must not write into any local variable that is used by the system. The filter code must behave as an absolutely neutral and independent memoryless module that can be put in or taken out arbitrarily without affecting the system's behaviour other than in the following two aspects: (1) the system may execute faster in the absence of the filter code, and (2) the system will be more vulnerable to unexpected influences in the absence of the filter code. There must be no other behavioural effects associated with the insertion or removal of such filter code.

The example above clearly satisfies both conditions.

Where do we install such a filter? The filter should be placed between the code of the system that we are realising, and all values that come from outside. As shown in the example, the system's formal parameters are an obvious source of influence. But other such sources are: a character stream input from a keyboard, a record input from a file, a record input from a removable storage medium such as magnetic tape or punched cards, etc. The case of the magnetic tape (and the such) is especially serious, because here

---

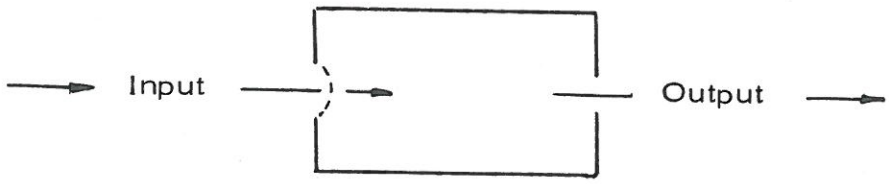
* The validation of an output argument O_j is for the purpose of ascertaining that it actually corresponds to a variable of the proper type and dimension. Possibly validation should be made of ref(O_j), or possibly there is no need to validate any O_j at all.

the choice of proper input is at the mercy of an operator who just mounts some reel of tape on the tape drive. Less acute but still highly possible is the case where files get mixed up or partially destroyed in a permanent online file system, thus any verification of the goodness of input is still advised even in the case of a permanent file system.

When designing the system (or the appropriate module) it is best to take the issue of file validation very seriously, and to design into the file itself a sufficient number of validation codes and formats with which future checks can be performed. These would include: file header and trailer records; record header and trailer fields; record, character or bit counts; checksums, etc. These techniques are well documented in the literature.

What about efficiency? Are we not going to end up with extremely large and slow running programs which spend most of their time re-checking assumptions that have already been checked before, a thousand times over?

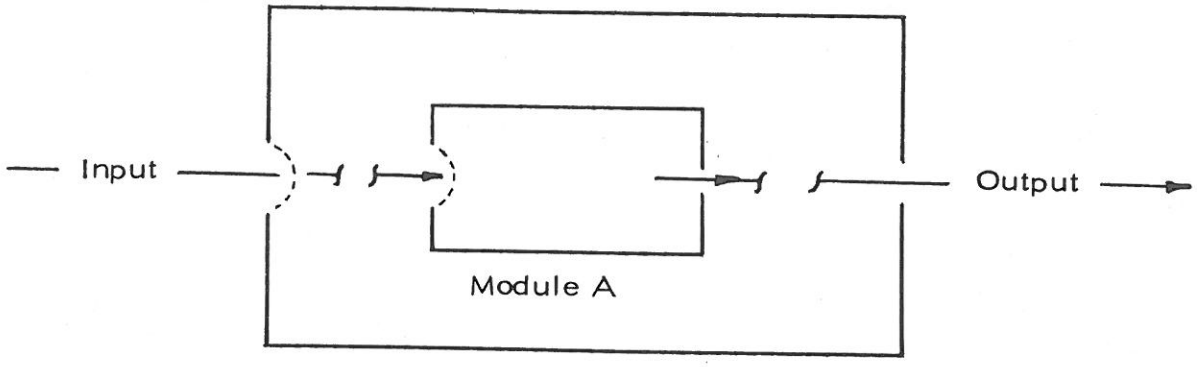
As things stand presently, the answer is yes. And this situation is not satisfactory at all. To understand the situation, and to see the remedy, let us illustrate the process of system realisation. Suppose that we realise a module A



Module A

as illustrated. I use representation  to indicate an input-value filter. Now we test the module and debug it independently, all the while using the filter. In fact, an important step in debugging and testing is to supply the module with erroneous inputs and to verify that the filter actually works - after all, it too is an algorithm that may have a bug! When the

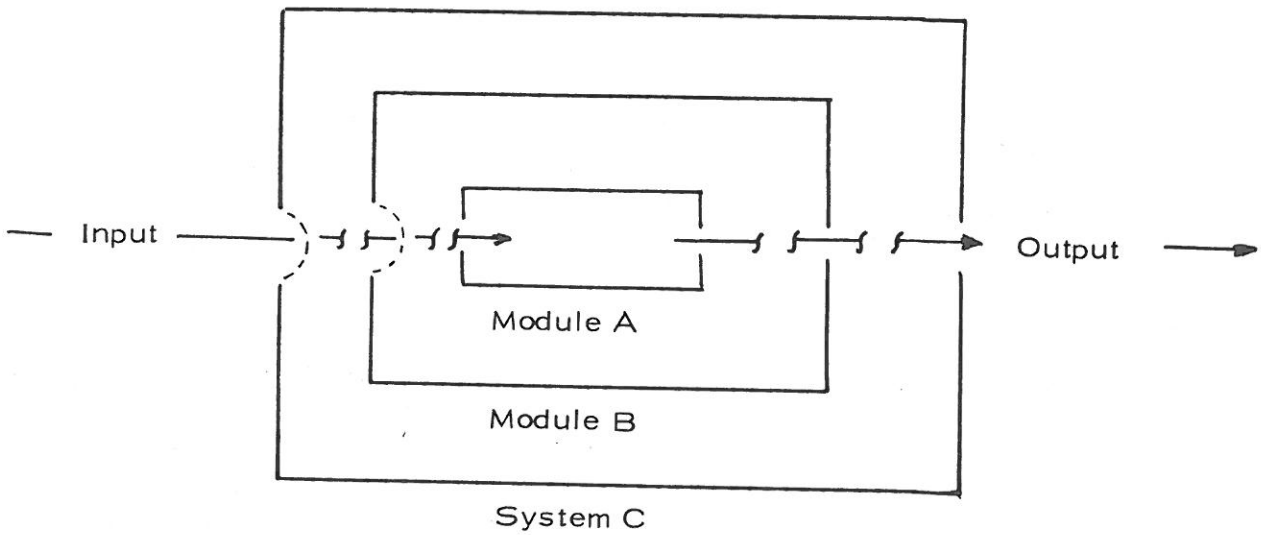
module is finished, we proceed to realise a higher-level system B, of which A is a component:



System B

We debug and test B in the same manner that we did A. It can be seen that the filter in A may well serve as an effective debugging tool for B, because if filter A traps an erroneous input, that input can only be produced by B, and that is a definite bug in B and needs correcting. As we did with A, we must also provide B with erroneous input to verify the proper behaviour of B's filter. If while we do this A's filter traps the erroneous data, then there is a bug in B, for properly B should provide A with complete protection from external unknowns.

Having finished B, we realise a system C that contains B (and indirectly A) as a component.



System C

There is no further need for a filter in module A, and the illustration above shows an arrangement where only B and C have filters, but A no longer does. When a finished module is two levels deep, in the path of control, from a source of unreliable input, that module needs no filter. Note that this applies to levels within the path of control. If module A can be called from either B or D

$$\{C \rightarrow B \rightarrow A\} \quad \text{or} \quad \{D \rightarrow A\}$$

and D and C are in the same level pertaining to input assumptions, then A must retain its filter even though in one activation case (namely $\{C \rightarrow B \rightarrow A\}$) the filter represents pure overhead.

How is the filter to be removed from A? Before we settle this technical question, I just want to note that by now it should be clear why I formulated the second existential condition for filter code, whereby it was required that the system's behaviour be unaffected by the arbitrary removal or insertion of a filter. For evidently it is intended that we remove filters as soon as the system they serve is nested sufficiently deep within the program that we are realising. An obvious thing to do is to edit all the filter code out of the program text, and then to recompile the edited program.

I strongly recommend against this technique.

First, as a rule, one should not modify a program text that has been tested and verified. If one does modify it, then one should retest and validate the program (and for that one would like to have the filter code in again!). Second, it took time and effort to design and code and validate the filter algorithm. It seems a wise precaution to somehow keep it around if ever it is needed, but at the same time to remove it from the system's runtime behaviour, for efficiency reasons.

```
SYSTEM
      manifest  {filter = false}
      if filter let validate([<inputs>],[ ]) be
            {<argument validation logic> }
modname: GATE([<inputs>],[<outputs>])
      if filter validate([<validation parameters>],[ ])

      <Essential logic of program >

ENDSYSTEM
```

The example code above shows how the filter may be removed or re-inserted arbitrarily. Any decent modern compiler has some facility for conditional compilation. In the example I assume that when the compiler encounters a statement of the form

```
if false <action>
```

then it automatically removes the entire statement from the program, in the sense of not compiling any object code for <action>, because obviously such code could never execute. By the simple expedient of redefining "filter" to be either false or true, and by recompiling the program text, we control the removal or insertion of the filter's behaviour. This still calls for the editing of the program's text, even though such minimal editing can be undertaken quite safely.

Certain language processors have an explicit compile-time expression evaluation facility which specifically controls what code does or does not get compiled (and even what code does or does not get displayed on the compiler-produced listing). A desirable ability of such compilers is to declare compile-time variables or supply values from the command language interface, so that one could control the conditional compilation of a filter as follows:

```
COMPILE  program.source(let filter = false)
```

or

```
COMPILE  program.source(let filter=true)
```

where we gain the desired effect without ever having to touch the program's source code (I here assume that the declaration

$$\underline{\text{manifest}}\ \{\text{filter} = \left[\begin{array}{l}\underline{\text{true}}\\ \underline{\text{false}}\end{array}\right]\ \}$$

is no longer part of the program text).

You have of course other possibilities of realising the conditional compilation of the filter code. This depends very much on the language processor available to you. If needed you can use a macro facility or an include facility (in this case you should better make the filter code into a single begin...end block that is a distinct physical text file and that is included in the proper place; to turn the filter off you substitute a null-content include file), or even some privately made text preprocessor. In the most pessimistic case, you will have to remove the filter code through actual editing of the program; still you could do it in the following systematic way:

```
SYSTEM
// let validate([<inputs>),[ ]) be
//         <argument validation logic>

modname: GATE([<Inputs>],[<Outputs>])
//          validate([<validation parameters>],[ ])

          <Essential logic of program>
ENDSYSTEM
```

This way, by removing the filter through commenting, the filter's logic remains perfectly documented and available in the source text. Remove the comments, and the filter is effective again. Indeed, even this can be further automated. Let the pattern "//Filter" be the unique comment that disables the filter. Then the editing command

```
substitute all "//Filter" with "/*//Filter*/"
```

will re-activate the filter (I assume BCPL commenting conventions), and the editing command

substitute all "/*//Filter*/" with "//Filter"

will de-activate it.

Why should we worry about filter re-activation? Because we must never assume a program to be finished, never to be changed again; and because we must never assume a program to be used in a certain way only, and not to be used in the future in ways presently unforseen; and because we must never assume a program to be used in perfect knowledge of its internal composition.

We are called upon to code our programs defensively, for the benefit of future users or modifiers. It is best to exemplify the circumstances under which the re-activation of dormant filters is useful, by adopting the point of view whereby we are the beneficiaries of this facility. Suppose EXIST to be the name of an existing program, of subroutine form. Let us look at some cases:

- We are supposed to modify a program containing calls to EXIST (as part of our modification we may even have to code additional calls to EXIST, which we may do in perfect ignorance by mimicking existing ones). While we are testing the new code, it is advantageous to have EXIST's filters re-activated.

- We are supposed to modify EXIST. It is not clear which other programs are calling it, and how. In this case it is not only advantageous, but indeed necessary to activate EXIST's filters and to leave them in the activated state for a reasonable useage period, until we have ascertained (or are at least reasonably convinced) that all useage of EXIST is consistent with it.

- We are in need of certain services, and are told that there is a subroutine named EXIST (available from a public library, or publicly available but of uncertain origin) which provides such services. We are told that there is a certain way of calling EXIST. Either we have no way of knowing any more about it (in which case somebody committed the sin of not documenting EXIST when it should have been - but that

is a matter to concern us elsewhere), or we simply do not wish to know any more about it even though it is amply documented. In either of these cases of ignorance (and even in the case where we took the trouble to study EXIST's inside) it is highly reassuring to know that EXIST can be recompiled with its filters re-activated, that it will not tolerate misuse, and that it will help point out such misuse so that it can be corrected.

The above points apply more generally to the collection of modules that is our program. We should have the ability to re-activate all filters in all of the code whenever it is being modified or used in a new fashion. Recognizing that the only serious objection to the filter is based on its runtime inefficiency, the program should be used without filter under the following compound condition: (1) a certain pattern of useage, with active filter, is pronounced satisfactory, and (2) that same pattern of useage can be satisfactorily repeated with the filter de-activated, and (3) a consistently measurable improvement in efficiency is observable when the filter is removed, and (4) the program has not been modified, and (5) the pattern of useage has not been changed.

Of these, sub-condition (3) is of special interest. Lamentably, it is common for programmers to introduce "tunnel-vision" optimisations: optimisations which can be seen as such only from a very narrow and local and unrealistic point of view, and which are practically unmeasurable when the efficiency of the total software package is being considered. Very often such optimisations are achieved at the expense of algorithmic robustness, and may cause difficulties of the kind that we are trying to avert through the use of filters and similar devices. Most definitely, he who writes such tunnel-vision optimised code does not put into his code any such non-productive excess baggage as a filter. But it is very possible that not only is the tunnel-vision optimisation unmeasurable and can thus best be left undone, but that even the addition of filters and other defensive code will remain totally unmeasurable insofar as efficiency goes. *

---

* There once was a very sophisticated online information storage and retrival system, using a very intricate data base facility. It was made by one of those tunnel visionaries, who optimised it so well that random data items of critical importance would dissipate into oblivion under unpredictable circumstances. I was "volunteered" to remedy the situation.

(continued on next page)

Thus it is always advisable to apply measurement techniques to the finished product, and to ascertain that the de-activation of a filter is indeed measurably contributing to the product's efficiency; for if it is not then the filter is best left active.

To further justify the usefulness of having filters that can be re-activated, let me propose an even more radical technology for the de-activation and re-activation of filters. Using this technology, when the filter is de-activated then there is practically no penalty in either execution time or in program memory useage. Yet the filter can be re-activated at will, without recompilation; in fact it can arbitrarily be de-activated and re-activated in the middle of the program's execution! Let the technology reside in a compiler which can recognise the semantics of

```
filter
        <body of filter>
endfilter
```

The compiler generates the body of the filter such that its behaviour has absolutely no side effects on the logic external to it; it places the body of the filter in a locality that is physically external to the program's object code. In-line it only generates a very fast jump, or subroutine call, or processor trap to a general filter handler sequence. The filter handler is in either two modes

- filter enabled in which case the filter activation is allowed to proceed to execute the appropriate filter body code, or
- filter disabled in which case an immediate return is made to the point of activation, and the normal program logic continues, uninterrupted.

---

* (continued)
That I did by replacing the system's innards with code containing the most heavy-handed defensive and suspicious logic imaginable. This extraneous code was put in the most heavily-used control paths of the system, and is permanent! As a point of honour, I finished the job by increasing the system's response efficiency by 40% compared to what it used to be; I did that by re-working a 200-line section of code that interfaced with the physical disk. Not only had the tunnel visionary missed an obvious optimisation possibility, but his effective optimisation was measured in form of tens of thousands of dollars of actual loss.

We can even make this device so that the de-activated filter incurs absolutely no execution time penalty at all. Namely, let the facility reside in the processor hardware in form of an instruction

filter([<filter parameters>,<filter code>],[ ])

where <filter parameter> and <filter code> are pointers to physically remote memory, and where filter is a conditional instruction code: when the processor is in filter enabled mode, the instruction is executed, when it is in filter disabled mode the instruction is skipped.

Why do I go to such great lengths discussing this technique?

Because there are cases when it is of the utmost importance to trap various conditions leading to program misbehaviour, and where the ability to switch to some filter mode in the course of program useage is most desirable. Specifically, let us suppose that there is a massive program, made for some customer. We test it to the best of our ability and find it satisfactory. We then remove all superfluous inefficiency-causing filters (e.g., through recompilation), retest the program and still find it satisfactory, then deliver it to the customer. On customer site the program operates well, except for those occasions when it does not. The customer complains. We test the program even more thoroughly; we improve certain suspected parts of it; we do all sorts of reasonable things to it; we deliver it to the customer, where it normally operates well, except for those occasions when it does not. The customer complains. We now obtain copies of all of the customer's actual data (his privacy considerations permitting) and use that for testing, etc. Yet on customer site the program still misbehaves. The point of the story is that there is sometimes an obvious use for being able to activate, on customer site and during his normal production run, the most heavy-handed and measurably inefficient filter code, if that is the only way in which a very elusive error condition can be trapped. And this is one more important reason for having filters that can be activated, of which I wanted to make mention.

The filter is a device for verifying assumptions over which we have no control. The assumptions local to the program that we are realising are, of course, under our control. Once we know that all input influences can be trusted to conform to their specification, then there is no further reason for not writing tersely elegant code.

In general, that is true. We write the code; we understand all the assumptions that we invent; we are (hopefully!) conscientious enough to document the program's logic and the assumptions that we make, for the benefit of our followers; therefore, in general, we need not resort to any heavy-handed coding techniques.

But there is one particular case where a highly defensive attitude towards our own code is called for. That is when we code a complex management program for some static storage system (e.g., a storage allocation system, a file system, a data base, a symbol table manager, a queueing facility, etc.). These systems all have in common the property that the state of memory is constantly evolving, and that its present state is a consequence of the sequence of all of its previous states. There may be bugs in our code which can only be activated under some complicated sequence of events, and which can normally not be tested for; more disturbing, such bugs often cause error manifestations that are so far removed, in time, from their actual cause, that it is an almost impossible task to identify and correct the real problem. *

In such cases it is most advisable to design our algorithm in such a way that it is self-checking. As a rule, these algorithms have a number of complementary functions, such as: make a new entry, delete an existing entry; insert an item, remove an item, look an item up; remove the first item from

---

* E.g., see previous footnote concerning the bug in the information retrival system, which was of this nature - caused by remote consequences of an overly "efficient" hash-coding algorithm. For another example, I once worked on an operating system scheduler, where the system would "forget", hence lose, all of its 50 user jobs by an attrition process requiring up to 36 hours of continuous up-time; the only way to discover what was happening was by inserting very inefficient testing and verification code into the process switching module, and running the system normally for 36 hours, gathering specific data on the system's state as processes were disappearing.

queue, append a new item to end of queue; etc. We normally make a highly efficient subroutine which locates the desired item, then write the necessary code to perform the desired function. The danger here is that by abstracting the accessing logic in form of a single subroutine, we may have trouble in identifying bugs. It is possible for the accessing subroutine to be erroneous, but so long as it is consistently erroneous, it will allow us to test the code by, for example, inserting an item in a wrong place, then locating it again in the same wrong place. As we test the program, it appears to us as if access to the static storage area works satisfactorily, when in fact it does not. *

It is best to illustrate such a defensive technique. Let us assume a list-processing system where there are n lists L_1 through L_n, and a free list L_f. The lists are all double-threaded and have head and tail pointers H_i and T_i. For each list we have an item count C_i; I'll assume all items to be uniform, so that there is always a fixed number nitems of them in the system, albeit variously distributed among the lists. We code one function that removes an item from a list (including the free list) so that the item is on no list, and another function that inserts an item into a list (including the free list). An item is identified by a reference to it, a list by its name L_i.

The logic of remove([L_i,<item designation>],[itemref]) is to access the designated item by proceding forward along the list from its head H_i. When the operation is performed, remove updates C_i.

The logic of insert([L_i,<place designator>,itemref],[ ]) is to access the designated place in the list by proceding backwards along it from its tail T_i. When the operation is performed, insert updates C_i.

---

* E.g., still concerning that data base system, one of the bugs in it was that occasionally it would overlay an item with another, depending on how full the table was. It behaved very consistently on a per-item basis. Who would think of verifying that an old item named "xyz" was still there if moreover it so happened that there were over 3741 items in a category and that there were more than 11 items beginning with "x.."?

The last thing remove and insert do before returning to their caller is to invoke

validate([L_i],[ ])

which does the following:

[1] Follows the entire thread of L_i, counting individual items, and verifies the value of C_i.

[2] Follows the entire thread of L_f, counting individual items, and verifies the value of C_f.

[3] Adds up the values of all counters C and verifies that the total is the exact number of list items in the system, nitems.

[4] As an enhancement, depending on some well distributed binary random factor (e.g., real time clock register having an odd or even value), validate follows lists L_i and L_f either forwards or backwards.

[5] Under special circumstances, e.g., the setting of a special validation mode value, validate does its checking on all lists.

Whether or not validate is to do its work always, depends on whether or not there is a measurable efficiency penalty to doing so. In any case this question is of little importance given that we possess so many techniques for the conditional de-activation and re-activation of validate's behaviour. What is important is to design the list processing system's formats and structures such that there is information with which validation can be attempted. It is not always the case: I have seen "tightly-coded" systems which had absolutely no provision for control data such as counters, and where it would have been impossible to do any validation.

Understanding the need for such defensive tactics, the experienced designer designs the system's basic structures and algorithms already with validation in mind. For example, another difficult to identify error condition is when - for whatever reason - an item gets put into the wrong list. If we design the item format to include the identification of the list to which it is supposed to belong, then validate can check that too as it follows the list's thread. Of course such identification has to be set by a piece of code other than insert, or else it is practically useless.

Lastly, the use of a table-driven automaton is a very powerful technique to apply at areas where modifications can reasonably be expected in the future. Unfortunately, not all applications lend themselves to be programmed in this way. The advantage of this technique is that the main algorithm is independent of the program's actual logic, because it only serves to execute the table. The table represents the program's actual logic expressed in static form. There exist many techniques for constructing tables, or for verifying statically that the table is of some normal form.

An interesting consequence of using such table directed logic is that occasionally one attempts to modify the program's behaviour, only to discover that there is no obvious way in which the new behaviour can be expressed in terms of the table's language. This is a most useful warning signal which should make one stop by reflex , considering the meaning and merits of the contemplated change. Had the program been coded normally, then chances are that the modification would have been incorporated, that nobody would have been aware of its incompatibility with present logic, and that the seeds might have been sown for future program deterioration. [In reference to section 1.XII: Levels of machines, we understand the table-driven automaton to be a level of total interpretation, hence its useful property of disallowing the expression of a semantics that is foreign to the automaton's language.]

Robustness and Elegance

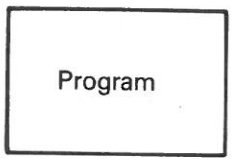
Spier My paper deals with the technique that I call *"defensive programming"*. The name is inspired by the Massachusetts department of motor vehicles which advocates *"defensive driving"* whereby, when you drive, you constantly watch out for what other drivers might be doing. The idea is that if you watch out for the others instead of simply assuming that they are driving reasonably, then there will be less chance of your being involved in an accident.

Shelness The assumption is that everybody else will do the worst they possibly can, and so you concentrate in advance on possible escape routes to be taken if indeed they perform their worst.

Spier I advocate structuring one's programs defensively, in a way that would yield what I call *"robust"* programs. Robust programs are strong and indestructible. For example, one very important problem is keeping operating system monitors or telephone switching systems working indefinitely. These systems have the opposite of the halting problem: they terminate, when in principle they never should.

The difficulty lies in our tendency to rely on too many assumptions, especially assumptions concerning a supposedly benevolent runtime environment.

An illustration of the concept is made by drawing a box.



Rest of universe

You program the insides, and know perfectly what you are doing. You make assumptions about the outside, and by choosing your assumptions conveniently, you can make the inside look aesthetically appealing, or elegant. A robust program, constructed according to my suggested technique of defensive programming, will not be aesthetically pleasing inside because it will contain a lot of code to verify assumptions made about the surrounding universe. You must consider this outside universe to be *a priori* hostile.

I am suspicious of *"elegant"* programs. Most *"elegant"* programs are based on so many assumptions, that under real life conditions the probability of all these assumptions being jointly satisfied is low. This gets you into trouble.

Gram You put something different into the word *"elegant"* than we do.

Grosch I think we have a problem of definition. I am sure that nobody here is confusing *"elegant"* with *"lacey"* or *"intricate"* or *"baroque"*. I may be putting words in Spier's mouth, but I think that what he is trying to say is that by the time you have ground down your parts to an exact minimum and put them together, you may find out that you have been so miserly with your pieces that the system leaks. Like those new metal buildings. Some are so stripped down, to perform an exact function, that the windows pop out in Boston because somebody has not thought about *all* the affecting forces. Spier says that even if the wind does blow a little hard, the windows need not automatically pop out.

Flynn What is elegant to the engineer may be lousy to the user. What is elegant to the programmer may be lousy to the user.

I think that one important ingredient in a program is the degree of stupidity and predictability of the user. I think that is what you talk about. By the time you program in defenses against stupidity, to make the program robust, it may not be elegant, using the current concepts of program elegance. Making an accounts receivable package is, however, quite different from making a subroutine for solving differential equations.

Grosch Concerning differential equations, you are of course entitled to code them elegantly and tuck them away somewhere where you know nobody will trifle with them. You don't in principle need to put in a lot of these robustness artifacts because you trust your system. But suppose there is a little twitch in the power supply and a bit creeps in somewhere. The trouble with the differential equations is that you won't know something has gone wrong. That problem always bothered me, this kind of lack of robustness.

Shelness I don't see how code could be elegant. The design may be, but the code will always have to reflect compromises made with reality. In some sense it seems that civil engineering is a constant conflict between elegance and robustness.

Spier Allow me to show you an example of defensive programming. We have a two-value variable *BOOL*. The mathematically-elegant use of it is

if BOOL **then** <*true action*>
else <*false action*>

while the defensive use would be

if BOOL = **true then** <*true action*>
else if BOOL = **false then** <*false action*>
else <*blow the fuse*>

where the last alternative is logically impossible, but practically admissible as in the case, for example, where the value of BOOL is neither 0 nor 1 but −67, or where the compiler is either stupid or inconsistent.

Stoy So it is not a boolean variable.

Spier That is what we just found out. We only thought of it as boolean. I had a very interesting bug once where two modules had to communicate through a boolean variable. They were compiled years apart with different versions of a single compiler, whose definition of boolean had been changed well after the compilation of module *A* and before the compilation of module *B*. So according to one module **true** = 1 and **false** = **not** 1, and according to the other module **true** = **not** 0 and **false** =0. I must add that the compiler changes were motivated by object-code efficiency; consequently the compiler itself generated no defensive code whatsoever.

Beitz I can tell you what you did wrong. You used more than a single bit to represent a boolean.

Spier I didn't use anything. I only discovered the thing. But I talked to the compiler writer, and he was convinced of having done something extremely elegant and efficient.

Grosch It is not elegant to have two definitions of **true** in a single program.

Spier I agree, but defensive programming is the discipline of laying traps for such inconsistencies, and in general verifying one's assumptions. In hardware we use fuses to check our assumptions about current intensity. Why not learn from the hardware engineers and put some fuses into the software? I can tell you why — because it is inelegant by any definition.

Lee You still must assume the compiler to work. You can't protect yourself against the compiler.

Spier No you can't. But there are ways of counter-checking one's logic so that at some point you'll blow the fuse. These have to be designed into your program, and this is the technique I'm advocating.

Staunstrup Why don't you verify that 2+2 gives 4!

Derrett The idea is not to be absolutely sure everything works, but to be more sure than we are at present. The question is where the checks should be placed, and Spier is advocating putting very strong checks at module boundaries.

Spier Another point. Industrial software has to be finished some time. One can't continue refining it indefinitely. Consider Dijkstra's book. Who can tell me how long Dijkstra sat there, pondering his algorithms? It may have taken years. The question is whether anyone would have paid a programmer's salary for years just so that he could come up with his most elegant solution; when in fact it was necessary to have some solution — any solution — within 3 weeks.

Also the money spent on writing programs is negligible in comparison with the money spent on maintaining existing software and keeping it alive. Programs get modified. I have yet to see an elegant program that gracefully submits to a single modification, let alone to many years of constant evolution. If it survives, then its original elegance is long gone.

I may build a program that behaves like a rocket, and maybe it is a good rocket. But then the user comes along and complains and modifies it and in the end he is using it as a submarine. But he still expects it to work. That is what I mean by *robustness*.

Shelness The structure of any program must reflect its function. It is unreasonable to expect programs to perform completely different functions from those for which they were written.

Spier That is right but my requirement for a robust program is that it either functions correctly as a submarine or refuses to work at all. The program should itself check whether it can function correctly in its environment — it should not simply assume that the environment is benevolent.

Let me further exemplify robustness: I have often assigned some problem to a beginning programmer. He comes back to demonstrate the thing to me. He shows me a computer terminal and tells me to type in a 3-digit number. What I do instead is lean with my elbow on the keyboard. Under such conditions, the behaviour of non-robust programs is rather unpredictable. The programmer usually defends himself by saying *"you were not supposed to do that!"*. You see, he had made wrong assumptions of benevolence about my own behaviour. And I turned out to be hostile.

Beitz Elegance and robustness don't preclude each other.

Spier You are right, but we must get our priorities right — I am interested in programs which work, even if I lean my elbow on the keyboard instead of typing in *"793"*. If they are also elegant then that is even nicer. I am not interested in elegant programs which crash at the least provocation.

Stoy If I want a compiler, I want a robust compiler that will work even if people lay their elbows on the terminal's keyboard. But I want to know that it is robust, and I don't want to take people's claims for granted simply because they did things that were supposed to make the compiler robust. Thus, I want to read the compiler, and the code has to be intelligible. I would not mind about it not being elegant, but it has to be intelligible so that I can read it.

Spier I, too, would like to read it. One thing I found out about elegance is, that elegance is in the eyes of he who writes such code. But the code may not be comprehensible to anyone else. It is difficult to read the stuff.

Derrett I think that this is the trouble with a lot of algorithms people call *"elegant"* — they are totally unintelligible.

University Training

Spier *"Elegance"* can mean many things. Sometimes we write a program for a well-known relationship which has been researched exhaustively in the past. Such a relationship is perfectly known. We reflect our perfect knowledge in the form mathematicians call elegance. But this applies only to the few relationships we know. I think it a crime to teach only those examples to our student programmers, and to claim that these examples embody the art of programming. To emphasize this point, I discussed the following hypothetical situation with my students last semester:

> Suppose that computers had been invented in the middle-ages, centuries before Newton or Leibnitz or any of the other founding-fathers of mathematics. Several centuries later, differential equations are invented. How much elegance would there be in a Fourier transform subroutine, and to what degree would coding it be different from today's coding of the accounts-receivable package?

What we call elegance is the collective knowledge passed down to us by generations of predecessors. We restate that knowledge in the notation of *Algol* and *FORTRAN*. Not having inherited perfect knowledge of accounts-receivable, we are reduced to coding the little we know inelegantly. I don't think that there is much to learn about the art of programming by looking at those elegant examples, because I believe that their elegance lies elsewhere than in the domain of programming. Students at universities are not taught how to solve other than the *Eight-Queens Problem* and its friends. They grow up under the assumption that every programming problem has an elegant solution and that if you just think long and hard enough it will unavoidably be reached.

Flynn Spier's point is well taken about mathematics. The stuff was studied for a long time, and there is some awfully nice mathematical software around, that has constantly been refined. But the basic algorithms have been around for ages, and people just stumbled upon the basic flaws in them one by one, as time went on.

Spier I am advocating that when we have to solve a new problem we should produce a robust solution that works. Then when this solution has been used for years and years, perhaps we will come to understand the problem better, and in the end we will be able to produce a solution which is elegant.

But universities tend to teach students things the wrong way round. They concentrate on producing elegant programs and ignore robustness.

Flynn People have different ideas about the function of universities in society. If we say that the universities' function is to train students for jobs, then robustness is clearly a number-one priority; but if the function is to produce new ideas, then elegance and robustness are equally important. On the whole it is easier to produce new examples of elegance than it is to produce new ideas of robustness.

Spier Students are also taught to think that all problems are well-defined. They are never made aware of the fact that a real program's specification is always approximate at best, and sometimes a straightforward lie. They are not taught to question the specification, which would make them aware of the hostile surrounding universe and of the validity of their assumptions.

Shelness. I intentionally teach with false or misleading or extremely vague program specifications. First the students complain, and then they learn to come and discuss the problems with me.

Spier When I hire a university graduate I don't expect him to do any useful work for the first six months. If he continues to be useless after that period he gets fired, but at the start nobody expects that he will do anything useful. Typically he has to unlearn a lot of the things he learned at university.

Beitz Maybe you shouldn't hire graduates.

Mrs. Grosch We have found graduates to be the worst possible programmers.

Gram I think Spier needs to do *defensive hiring*. There *are* universities which teach the skills he is asking for. My own is one of them.

Lee I think that the problem of teaching bad work habits is not confined to the universities. I am very concerned about what is going on in the high-schools. Some of the kids we get from the schools have absolutely no grasp of principles — they just know a lot of programming tricks.

Grosch When people ask me who to hire, I often tell them to hire graduates from disciplines other than computing: chemists, engineers, and the like.

Lee That is what went on 20 years ago!

Grosch Yes — but it worked 20 years ago.

Program Goodness

Peter Marks

Presentation by Marks

Program Goodness

Marks I have been looking at the problem of defining a commercial system that would allow the small unsophisticated customer who has no prior EDP experience to get his application running with little or no professional help. The small businessman wanting to use a computer — can he get away with not having to hire a programmer? Is it possible to program a commercial application without being an accomplished programmer?

I was looking for a system that would allow such a customer to do his payroll and inventory and accounts receivable, with a minimum of prior training requirements. The language had to have a minimum of inhuman, or formal-syntax, interfaces.

At present, the system analyst going to a customer to design the new computer application is really performing a translation function from the customer's current manual procedure into the planned future computerized procedure. This manual system exists and is bug-free. This is a most important point. We do not have to solve the inventory problem — we only have to simulate an extant and acceptable solution.

Spier I have had that experience over and over again. One of the most complex commercial systems I was involved in designing was a large-scale computer system for the typesetting of daily newspapers. Newspaper production is an extremely complex, fine-tuned and perfectly debugged operation. Every morning, on the appointed hour, the trucks start rolling to distribute the morning paper. The system works like clockwork. Our design problem was to set up the computer system to do what was already being done, only faster and cheaper. It is a sobering experience to work on a very large system knowing that the overall algorithm is well defined and acceptably debugged and only has to be *"re-coded"*.

Shelness I do not believe these manual systems are perfectly debugged. What I think happens is that people are processing the data, and they intuitively recognise and correct big mistakes. The little inconsequential mistakes slip by, but are of no great consequence. Now you automate this and the machine lets slip by both big and small mistakes, because we have eliminated the checks previously performed by humans as they were manually processing the stuff.

Spier You are right. The transition from manual to computerized is usually slow and painful. The two procedures should continue for a while in parallel, before the switch is made. But by and large, the algorithm is well defined; at most the finer details have to be ferretted out painfully.

Grosch From my experience I can say that in many cases manual procedures are extremely sophisticated, because very often the people who perform them are very clever people. So just getting to understand what the manual system actually is, is itself a difficult undertaking. This is a well-known phenomenon that becomes painfully clear whenever the person performing such a sophisticated manual procedure resigns or retires.

I used to have an office next door to a man named George Richter, who was essentially the scheduler of production of the old kind of IBM machinery — collators, sorters, tabulators and the like. He had a whole series of techniques for deciding what got scheduled where — it depended on the length of the telephone call from the local branch manager, and whether Mr. Watson had ever visited the customer six years before, and a whole mass of other things. Based on this extremely complex and undocumented calculation he would decide whether to take an order out of turn and send it off early, to reorganise the work schedules, to delay an order, or other things.

They tried to simulate him on a computer. A young man was hired to sit beside him and watch him and take notes. George and I used to have a good laugh about this, because George misled him most of the time. I don't honestly know how they replaced George when he retired — but that was one of the most sophisticated manual systems I have ever seen.

Marks The real problem as I saw it was that the analyst's design typically calls for the user to learn the analyst's (or rather, the computer's) language. And I don't just mean FORTRAN or COBOL or JCL, but also action codes, and funny symbols, and right- or left-adjusted data in punched card fields, and what must or must not be specified before or after what else. In short, rigid syntax, rigid logic, and an excess of all of the computer's de-humanised manifestations. And the customer wonders: *"For twenty years I kept my books without needing such mumbo-jumbo. Why do I have to learn it now?"* And I am convinced that he does not have to.

This touches on the basic problem of program *"goodness"*. I see two aspects to it. The program has to work well, and it also has to be nice to work with. A system that intimidates its users by strange interfaces lacks goodness, regardless of how correct it may otherwise be. I've worked in a bank and remember how confused and unhappy employees such as tellers were because they had trouble relating what they were doing to the menacing and incomprehensible doings of the computer.

So I made a small simple model system and asked in-house people such as financial analysts, planners, book-keepers etc. to try and use it. I made it very simple. For example, all data is stored in tables because these people's habitual representation of data is in rows and columns. Inside, the data representation was relational, but that was my problem and I never let the user know or worry about that. Once they discovered that they could enter their ledgers and account sheets directly into the system in the habitual manner, they got to using and liking it. They became very adept at it. None of them had any programming knowledge or experience, nor, for that matter, gained any from using my system. They still use it and are happy with it.

After I had gained experience with the first system I made a second one. Again it was not very sophisticated, but it could be used for a wider range of applications and I found that many people started using it.

The system's main advantage is its lack of syntax. I had made a conscious attempt to present to them the ideas that they were used to dealing with in their own language. The user has, of course, to learn a few rudimentary commands to start off with. But once he embarks upon some transaction, the system interactively coaxes him and prompts him in the options available to him, literally leading him by the hand through the transaction procedure. This happens in form of a very English-looking verbal exchange.

Gram But once the user had learned to use your system, did he not come back complaining of verbosity and asking for a shorthand notation?

Marks As a matter of fact the opposite happened. The system was equipped with shorthand, in the sense that as soon as the user typed in enough of a word to allow for unambiguous recognition he could stop and the system would complete it for him so that the visible display was always in clear language. Well, as they became more adept with it, they got into the habit of typing in the entire token faster than the system could complete it for them. In effect they opted for longhand. I guess that people simply get bored. Whatever you offer them, after some time they'll ask for something different.

Lauesen How many in-house users do you have?

Marks About forty.

Lauesen Don't you get conflicting feedback from them, asking for divergent and mutually incompatible improvements, each for his local area of application?

Marks No. I received various kinds of feedback, much of which resulted in improvements or modifications. But I never acted on their literal request. They would express some intent the best way they knew how, and if they had a useful insight then I'd effect the necessary modification. So the feedback resulting in modification was very common-sensical and no conflict arose. But I did have to contend with all my programmer colleagues who offered their unsolicited expert advise.

A common mistake that we tend to make is to judge users in accordance with our own understanding of computers. So if the user is unfamiliar with our professional jargon we think of him as unsophisticated. From here on it is but a short step to thinking of him as being unclever. But the customer is clever — otherwise he would not remain in business or earn enough money to buy our computer. The kind of goodness I tried to instill into my system would recognise and support the user's cleverness by being a tool with which he could further and better exercise his talents.

Shelness What is the use of your system to the businessman — what can he do that he couldn't do before?

Marks He performs the same functions as before, using the computer terminal instead of a pencil and paper, but the computer provides all sorts of procedures for him — for example checking that the books balance.

Generality

Spier Talking of program goodness, and hearing Marks describe a very specialised system, I wish to raise the question of the goodness of generality. In my younger days I was convinced that generality is good. I was energetically looking for the ultimate programming language, or the ultimate operating-system monitor. The older I get, the more convinced I become of the inherent goodness of simplicity, specificity, single-purposeness.

I believe that our infatuation with generality is a major source of our software problems. We make something that works fine in a very specialised context. Then we proceed to generalise it so as to solve everybody's problems in all contexts. That is a sure way to get into trouble.

Grosch The PL/1 programming language is an example of this. It is designed for all sorts of computer applications and it isn't really good for any of them. It may be better than COBOL, but that's not saying much.

Lee I did some measurements on our system, to determine how many of the available facilities we actually use. Turns out the useage pattern requires only about 10% of what's available. The remaining 90% we never use. I wonder what it costs us to have this overkill in unused generality.

Marks I did not want to make my system too general. If you try to solve 100% of problems in all possible applications you loose a great deal in ease of use. So I insisted that there were certain things my system just couldn't do. That helped to keep it easy to use.

Derrett I think that we need generality, the trouble is that we insist on presenting all of that generality to the user. The user-interface of a system should not reflect the complexity underneath.

Spier When I use a tool I want to be ignorant of its technical details. — I want to do something, it preoccupies me; I have to use a tool; I do not want to have the tool distract me from my preoccupation because of an overly intricate or technical interface.

Beitz I see the trouble rooted in our lack of suitable primitives. I believe that once we identify a suitable small set of proper primitives, we shall get very good generality.

Lauesen In Denmark, Regnecentralen originally made very general application packages, parameterized by many variables. Today the philosophy is to make reasonable skeleton prototypes that are easily custom-tailored to specific requirements. So we see here a trend from the general to the specific. A trend towards the individual handmade solution.

Shelness The question of generality has been with us since the advent of the industrial revolution. We have the choice between on the one hand something that is general and mass-produced and inexpensive and not totally satisfactory, and on the other hand something custom-made and expensive and satisfactory. The former carries goodness in form of cheapness, the latter in form of exceptional usefulness. The choice is difficult to make.

Grosch If you look at small systems — for example the IBM Series 1, the idea is to get the customer to alter his requirements to suit the package instead of tailoring the package for the user. I think that this is very sensible. The manufacturer of a small system cannot afford the cost of massaging the system for each customer, and the customers themselves don't want to delve into the innards of the system to change it. So the solution is to sell the customer a standard package and then he will have to change his manual routines to suit the package. I think that this is the only way to go for thousands of small customers.

I recall meeting some librarians in California at the end of the 50's. They had a system for cataloging books for their local children's library. It turned out that the cost of cataloging library books was greater than the cost of buying them, so other libraries took their lists and bought the books to match! That was my first experience of a standardized computing system. People are prepared to confirm to a standard if they can save money.

Beitz Motor car manufacturers have known this for ages.

Derrett Grosch is presenting *Model-T Software* — you can have it in any colour as long as it's black; mass-production software which doesn't quite meet the users' needs but which is cheap. That idea bothers me.

Grosch The virtue of the *Model-T* was that it made motor cars available to people who otherwise couldn't have afforded them. The software I am describing will do the same — small businesses, corner shops and the like, will be able to afford computers which they would otherwise have to do without.

Derrett They might be better off without.

Grosch That is possible, but I would give them the choice.

Examples of Goodness

Spier As we philosophise here about software goodness, we do so in the abstract. I would like to see some concrete specifics. Would the chairman be willing to poll the participants and audience, and have each propose the name of just one software product that he deems *"good"*? This way we can form some intuitive idea of where goodness lies in the participants' opinion.

Shelness There is the editor we have at Edinburgh. Its goodness lies in its ability to run on any of the machines in our shop. Whichever machine I use, I find on it the same familiar editor.

Spier I would recommend *DEC*'s time-shared operating system for the *PDP-11*, whose name is *RSTS*. It is simple and convenient and sufficient for doing just about anything I would ever want to do. At worst it would do so inefficiently, but I find that that does not bother me any more.

Lauesen I like simple line-numbered text editors with no sophisticated commands. It is obvious what they do.

Derrett I would suggest the *BASIC* language. It is simple and dumb.

(Protests from Herb Grosch)

Lee I can't think of a system I was really satisfied with for the past 25 years. But then I don't necessarily regard this as a bad feature.

Flynn I can think of a specific mathematical package to solve differential equations which is released through Argonne National Laboratories to be useable on many machines, taking into account not only their native languages but also their arithmetic peculiarities. It is good because not only does it do its job, and is debugged and correct, but it is also completely documented. It is all there. That is where the goodness lies. A beautiful piece of published software that treats the user/reader as an intelligent being.

Bennett Why can that be done for numerical analysis? Is it because we understand the subject so much better? Why don't we publish compilers in that exact same fashion?

Spier Talking of compilers. Can anyone here mention a good compiler? Not a language, but a language processor program. I often find myself being biased towards the use of one language rather than another because the available compiler is so much more appealing.

Stoy I asked that question yesterday concerning any published compiler, and of course got no response at all.

Shelness I can't mention a specific compiler, but to me compiler goodness lies not in its ability to translate from language *A* to the more primitive language *B*. That I take for granted. Rather, in its ability to give meaningful and intelligible diagnostics when the source program is wrong. The world abounds with compilers that do marvels of compilation provided that the source program is impeccable.

Spier This is why I have come to prefer interpreters to compilers. The interpreter can always talk back to me when something goes wrong, and talk back in the exact same language that I wrote the program in. Especially useful in an interpreter is the immediate mode that lets me debug complex expressions dynamically.

Lee Then why don't interpreter writers attempt to engineer maximum goodness into their interpreters, by making these facilities as powerful as possible? I've always been driven crazy by *APL* which seems to have three error messages only: *domain, value,* and *rank.*

Nielsen I may be sticking my neck out, but I'd say that *FORTRAN* has goodness by virtue of its widespread usage and transportability. I think the language proper is awful. But its compilers by now are standardized across most available machines.

The fact that compilers are usually so divergent and machine-dependent prohibits us from devising clever tools that can be made to perfection once, and then be given widespread use. Instead, we are forced to re-program those tools over and over again, usually sloppily.

Mrs. Grosch *FORTRAN* is very portable — we've had good experience moving *FORTRAN* programs to different machines.

SESSION B

Program Design

What is program design? How do we specify what is to be designed, and how do we convince ourselves that we have designed what was specified? What methods exist for assuring us that we have made a design that is complete, self-consistent, implementable, etc.? To put it differently: what is "goodnes" of design, and what tools and methods exist to obtain, or check that goodness?

Software Goodness

Nigel Derrett

PROGRAM CORRECTNESS

by

Nigel Derrett

Introduction

The most widely discussed "virtue" of programs is correctness. However I suspect that many computer scientists do not really know what program correctness is, and in this paper I will discuss the subject. I will then go on to look at some other, more neglected, virtues of software.

Correctness

There are two stages in the production of a satisfactory program:

1. Stating the specifications.
2. Providing a program which meets these specifications.

These two stages are very different from the computer scientist's point of view. Not only are different people often responsible for the different stages, but, as we shall see, only the second stage admits of mathematical correctness proofs.

The difficulties of stating program specifications seem to be peculiar to computing. There are many ways of stating specifications with various degrees of precision. For example

"Write a payroll program"

is too vague, whereas

"Write a program that does the following"
- followed by 100 pages of assemblycode listing

is precise, but rather too precise.

In general a user's requirements will not be exactly stated. There are a large number of programs which will satisfy him to various degrees, and the programmer's job is to select one of them. If the user specifies his problem too precisely then the programmer's choice of programs is unnecessarily limited, and he may reject a simple solution which would have been satisfactory; whereas if the user does not state his problem precisely enough then the programmer may select an unsatisfactory program.

It is very difficult to write specifications with the right degree of precision. For example

"Print 100 prime numbers"

is an unambiguous specification. The user may be surprised if he gets back a sheet of paper with 100 "23"'s on it, but that would be a correct answer. If he wants different numbers he must say so:

"Print 100 prime numbers, no two of which are equal"

If he wants to be more precise he can say

"Print the first 100 prime numbers"

or

"Print the first 100 prime numbers in numerically increasing order"

These are all unambiguous program specifications, of varying degrees of precision and they may all lead to satisfactory programs for various applications. However

"Print the first 100 ACM members' names"

is an ambiguous problem statement, since it is not clear whether the user wants a list of the first 100 members or the first 100 names, and in any case there is no universally accepted enumeration of ACM members or their names.

An ambiguous specification is one which can be interpreted in more than one way. The real danger arises when neither the user nor the programmer realises that the specification is ambiguous - each having his own interpretation of it. Often, however, the programmer will realize that the specification is ambiguous, and will decide by himself which interpretation to use. All compiler writers have had this problem - language "definitions" are usually ambiguous.

If we are to avoid ambiguity then we must require that the specification is made in a language which cannot contain ambiguous statements. Computer languages are (fairly) unambiguous, but these are not usually suitable for stating the user's requirements. The main reason for this is that the user wants to specify functions - mappings from inputs to outputs, but high level languages can only be used for specifying algorithms - specific implementations of functions, given certain resources.

It is important that in our quest for an unambiguous problem statement we do not force the user to state his requirements in a language which is foreign to him, or in a form which is too detailed. Otherwise we merely exchange one potential source of error (ambiguity) for another (incorrect specifications) and the probability of producing a satisfactory program may be less than before.

We can talk about the "correctness" of a specification, and sometimes we can even check such correctness. For example if I want a brown book and I say

"Give me a purple book"

then I have made an incorrect request, whose fulfillment will not be satisfactory to me. An observant friend, who knows that I prefer brown to purple, might point out my mistake. However there can be no (mechanical) mathematical proof of the correctness of a specification. This is important to remember when we talk about "program correctness" - we can check whether a program correctly meets a specification, but we can never prove that the specification describes what the user wants. The best we can do is to provide a specification language which minimises the chance of making mistakes.

Even though we cannot prove that a specification is correct, we can sometimes prove that it is incorrect, by showing that it is inconsistent:

"Give me 2 books - a purple one, a green one, and a red one"

Very occasionally we may be able to prove that the functions the user has specified are non-computable (e.g. the halting problem) or of such an order of complexity that no possible implementation will ever give a result within the user's lifetime (e.g. a list of all possible chess games). Thus, while it is meaningless to talk about "proving" specifications, a means of checking them would be useful.

Given an unambiguous specification, we can check the correctness of a program which is claimed to meet the specification. We note again that the specification is given to some degree of precision and that there may be many "correct" implementations. Thus we must demonstrate that the given program is consistent (not inconsistent) with the specification.

A "proof" of an assertion is a demonstration of its correctness, to the satisfaction of the readers of the proof. The fact that a program has been "proved" does not guarantee that it is correct - the proof may be wrong, and some readers are more easily convinced than others; but the existence of proofs can make the chances of bugs in programs so small that we can regard them as correct. If the probability of a program going wrong is

much lower than the probability of the computer hardware going wrong then it does not really matter whether the program is "correct" or not.

It is not possible to "prove" a program in the mathematical sense. A mathematical proof is a demonstration that an assertion can be derived from other assertions given certain manipulative and logical rules. A computer program is usually an imperative. We can prove the assertion

"The square root of 2 is not a rational number"

but it is nonsense to talk about proving the imperative

"Shut the door"

or even the description

"A complex number will be represented as a pair of real numbers; complex numbers may be added by .....
....."

All that we can prove are assertions:

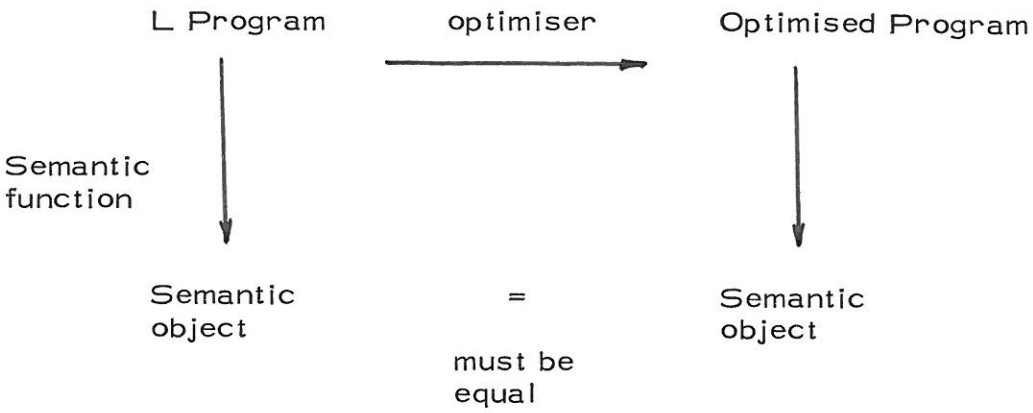
"After you have shut the door it will be closed"
"If you shut the door then the wind won't get in"
"This program terminates, whatever inputs it is given"
"This program executes in less than exponential time"

and so on.

If the specification is a set of requirements, then the assertion might be that these requirements are met at certain points during the program execution. In order to do this we may introduce new requirements (e.g. loop invariants) and claim that these are met at various other points during the program execution. We may then show that fulfillment of these new requirements will lead to fulfillment of the original ones. Now we must show that the new requirements are met for all legal program starting conditions. This is a widely advocated method for proving programs.

However this is not the only way to prove programs. For example the user requirements may be specified in another way:

> "Given a language L, with a semantic function which maps programs in L into a mathematical domain, and a measure of program efficiency, construct an "optimiser" which translates programs in L into semantically equivalent programs in L which are as efficient or more efficient than the original ones."



The proof here might be a demonstration that any language construct in L is mapped onto a semantically equivalent and no less efficient construct.

It is possible to "prove" a compiler in a similar way.

We note that any program which satisfies the criteria in the specification is a "correct" (bug-free) program. For example an optimiser which simply turns every program back into itself is correct. However the degree to which the user is satisfied with the optimiser will depend on how good an optimising job it does, and how long it takes to do it.

Usefulness

A correct program is not necessarily a useful one. A program may be useless because it provides a solution to a useless problem (for example the eight-queens problem) or because it provides an overly restrictive solution to a useful problem.

Much of the difficulty in writing good software arises in the provision of features which are not mentioned in the specifications. For example a compiler might produce a single error message:

"Source program is wrong - compilation abandoned"

This compiler might be provably correct, but it would not be of much use for getting programs to run. It is not "user friendly".

It is often the unspecified features of a program which make it pleasant to use. Conversely the lack of certain features may cause frustration and waste time. Some programs are like certain sports cars - they will run fast and reliably for long periods, but you have to be a contortionist to get into the driving seat, there is nowhere to put sticky sweet wrappers, and changing the oil requires a day's work and 5 special tools.

If a program is to be used and not merely admired, then it must have other virtues besides correctness. Indeed, these other virtues may be more important than correctness.

Program Modification

Some problems are well known, and can be precisely stated (for example the 8 queens problem). Making a new program to solve one of these is of little interest except as an illustration for teaching programming. Other problems are artificial and rather loosely stated (e.g. "make a THE operating system"). In this case it is unlikely that the specifications will be stated correctly and unambiguously before we are required to write a program to implement them. No-one can precisely describe in advance a system which will satisfy the needs of an unknown group of users for an unspecified time.

Thus we would like to write correct programs, but we would also like to be able to modify them later. This means that once a program is proved to be correct it should be possible to make "reasonable" changes to it without starting again from scratch.

Consider, as an example, a stack of size 64:

```
constant StackSize = 64
array a[1..StackSize]
procedure Push
        :
procedure Pop
        :
```

It is reasonable to expect that we can increase the size of the stack by altering the constant StackSize. However if the procedure Push contains a test of the form

```
if StackPointer ≥ 64 then Signal Overflow
```

then the program, even if it is provably correct, is a bad program, because it is not as modifiable as it should be. In this case the mistake lies in putting a single piece of knowledge in two places instead of one.

That was a simple example which indicates an important rule. What other rules are there for writing modifiable programs?

The above program would be worse if it contained a test of the form

if (StackPointer ∧ StackSize) ≠ 0 then Signal Overflow

In this case the programmer has made the assumption that StackSize is a power of 2, and that integers are represented in a specific way. (He is testing whether bit 6 is set.) The program is still correct (assuming that our language allows logical operations on integers) but its correctness depends on some extra knowledge. The use of this extra knowledge is only justifiable in special circumstances (for example if speed is very important).

A more practical example would be the modification of a teletype I/O driver to support a new type of terminal. A little thought leads us to another rule - split the program into logical units so that only a minimum need be changed. This is again a question of limiting the spread of knowledge as much as possible. Are the logical units we get by this approach the same as those which we would arrive at in a "structured" approach if we had not taken modification into account?

Up to now I have discussed the question of modifiable-ness rather loosely. This is inevitable - we must get a good feeling for the problem before we attempt to theorise about it. (This is the way that the theory of programming languages has grown up.) However we could try to be a little more formal. - Given a correct program, containing assertions from which its correctness can be deduced, what changes can be made to the program text, together with the assertions, such that the program may still be deduced to be correct? In other words we need a calculus of provable programs. Many of the rules of such a calculus would be familiar to us (for example, that consistent replacement of one variable name with another should not affect program correctness) but others may be new.

Reliability

Even though a computer system (hardware and software) is not known to be correct (or known not to be correct) we nevertheless want to design its constituent parts so that it will function satisfactorily.

There are two forms of incorrect program behaviour:

- The program may map inputs onto wrong outputs (i.e. the wrong function has been implemented).
- The program may "crash" when presented with certain inputs (i.e. only a partial function has been implemented).

In general the second sort of incorrect behaviour is to be preferred to the first; for example a compiler which crashes once in a while is far less dangerous than one which generates incorrect code. If possible the "crash" should not be too spectacular - the program should itself detect that something is wrong, and stop in a controlled way.

One of the most important aids to software reliability is the rule that functions should check their arguments. This sounds obvious, particularly to high level language programmers, but in my experience many nasty bugs could have been detected quickly in this way and removed.

Almost all functions in computer programs are partial functions - they only work correctly for a subset of possible inputs. Consider, for example, the factorial function

```
integer procedure Fact1(n);
    integer n;
    begin integer result, i;
        result := 1;
        for i := 1 to n do result := result * i;
        Fact1 := result;
    end;
```

Fact1 has been implemented as a total function - it gives a result whatever input it is given. However the factorial function is only defined for positive integers. If Fact1 is called with a parameter of -1 then something has gone wrong, but instead of detecting this, Fact1 will happily return a sensible-

looking result to the caller. In this way it may succeed in sweeping the bug under the carpet, where it can lie undetected.

What may be less obvious is that Fact1 only works for a limited number of positive numbers. For example on a 16-bit minicomputer Fact1 will only give correct answers for integers less than 8 (otherwise integer overflow occurs). Thus anyone who calculates ${}^{10}C_5$ as

```
Fact1(10) /(Fact1(5) * Fact1(5))
```

will get an answer which may look reasonable, but which is wrong.

Fact1 can easily be rewritten to check its parameter

```
integer procedure Fact2(n);
    integer n;
    begin integer result, i;
            if (n ≤ 0) ∨ (n > MAXFACT) then Error ( );
            result := 1;
            for i := 1 to n do result := result * i;
            Fact1 := result;
    end;
```

The usual argument for leaving such checks out is that they are expensive. This is a very weak argument - unless Fact1 is called several million times then the time spent on the extra check will not be noticable, and in this case more time will be saved by improving other features of Fact2 (for example calling n by value, which I forgot to do, or repeating the for loop from 2 to n instead of from 1 to n).

Of course none of us would ever be so silly as to call functions with the wrong arguments, but future modifiers of our programs may well do so.

Program Specification

Derrett I address myself to two related questions: First, how do you specify a design? Second, how do you convince yourself that what got implemented is what was specified?

If somebody tells me what he wants, and I design and build it, I can never prove that I have built what he wanted. All I can ever prove is that I have built what he told me, what I heard him say to me. But what he wanted and what I heard are not the same thing. That is one of the great gaps in computer science: the gap between what one wants and what one says one wants. We need some language to make specifications, but nobody knows what such a language should be like. When we consider the payroll-program example, nobody can really tell me what he wants. What happens is that I go to the firm that wants me to do their payroll and I spend three months there, or more likely three hours, and I decide what it is the firm wants. Then I go back and implement what *I* decided the customer wants. Alternatively, and perhaps more typically, I implement what my employer's salesman decided that the customer wants. Then I give the implemented system to the customer. There is no way for me to decide whether I made the right thing. What we need is languages with which the customer can write down what he wants, so that I can sit down and make the thing from that description. But again, even if we had such a language, we could never *prove* that the user has specified what he wants.

I'm unable ever to prove I know what the customer wants, so the best I can do is to try and give him a language with which he'll have the best possible chance of specifying what he wants in an unambiguous way. Now the problem is that a language in which he has a reasonable chance of specifying what he wants can never be unambiguous. Otherwise you end up with a computer language. But he can't use a computer language because he wants to specify functions, and all you can specify in computer languages are algorithms, which are specific implementations of functions. So we need languages in which we can specify functions and not algorithms.

Shelness There is something else you may want to do, and that is to make assertions. There may be certain commonality so that parts of those assertions may in fact be functions. But in general the customer may want to make assertions, such as how he pays his employees.

Derrett Yes. For example, the assertion that credit and debit have to balance in the books.

Shelness Or for example if you pay a mortgage, that you pay no tax on the first week of the month but pay more tax on the remaining three weeks of the month.

One can make it more formal by talking of the assertion in predicate calculus. The problem with this is that we don't really know how to turn an assertion into an algorithm.

I'm intrigued by the idea of using assertions. For the payroll problem you can come closest to specifying what you want to do by making assertions about how you want to do it. We need a way of mechanically going from the assertions to a program because we can change the assertions and deal with them far more easily than we can algorithms. Especially in the case of a payroll where every path for an employee may involve two or three exception paths.

In the case of telephone bills in Britain, every path for a bill involves on the average seven exception paths. And that has to be programmed. So of course you can say that the system being computerized is crazy. But maybe if you could come up with a set of assertions about how a telephone bill should be paid, you'd have a greater insight into this kind of system.

Flynn The question of what do you specify is something that haunts me constantly. When you buy software, even when you internally produce software for someone from your own environment, you commit yourself to a specification. That specification is really a legal instrument. It is very expensive to produce. It has to be tight. And you are contractually obliged to produce what is said in that document. But how do you know what it is you are supposed to specify?

Some problems come in when you ask the customer to speak your kind of language. Typically you may tell the customer *"I don't want you to tell me about your operation, I want you to tell me what to do."* And when the customer begins to tell you what it is he wants you to do, he is moving away from what he knows, the application, and towards the algorithm and its construction, about which he may understand absolutely nothing, or which he may express poorly.

Spier Once you have a legally valid specification, you have little choice but to conform to its letter. This is a universal problem that you get when you legally commit yourself to some future activity. You won't be rewarded for going beyond your commitment in meeting the spirit of it, but may well be penalized for not living up to its letter.

A dear friend of mine used to work for a large software house that does government contract work. They gave him a program specification, to be coded in COBOL. He worked very hard and conscientiously; it was his first job, having just graduated. He coded it to perfection, and tested it, and documented it, and made sure it is as good as he possibly could make it. He then proudly took it to his supervisor, to show *"daddy"* what a good employee he was. The supervisor quickly thumbed through the listing, then said *"You have only written 926 lines of code. The specification calls for 3000 lines. Go back and finish the program"*. And it hurts me to say so, but the supervisor had a point.

Flynn Some time ago I was involved in preparing a system specification that cost $120,000. More recently I was negotiator in, and witness to, the ugliest possible re-negotiation of that contract. $120,000 for a bidding instrument, a set of specifications of a system that someone says he'll produce for *x* dollars, and you have to come back and re-negotiate. The process is very ugly, and when it's all over you walk away with this subtle feeling that the instrument — the specification that cost $120,000 to prepare — is deficient in part. Should we have spent $240,000 instead to get more precision? Twice as much precision?

The ultimate constraint of software engineering is money. You don't produce something if you don't get paid for it. If computation were more expensive than doing it by hand then there would be no industry. Conversely, if as a customer you promise to pay money for a system then you expect to get one that does what you want, and you have difficulty understanding why the thing which was implemented does something else. This whole question of specification is frightening. I don't know what to specify. To what degree does the customer have to specify details of algorithms, of design? To what degree is the customer capable of doing that? We will all be better software engineers when we are able to live in a world of fixed-fee contracts.

Shelness What did it say? What was the legal content of this bidding instrument? Where was the difficulty?

Flynn We made some tacit assumptions. We assumed that people of good-will would understand that certain procedures had to be implemented in certain ways. All this refers to a very complex application. When the document was prepared we thought we had it all there; and only after this ugly experience did I go back trying to figure out what went wrong. (Two companies were almost bankrupted by this. We had a performance bond, which meant that not only would we not give them the money that was due to them on a particular day, we would in fact get back, from a surety house, all of the money that the contract had cost, and that would have bankrupted the

company.) I went back and re-read the specification a number of times, and I realised what it was that we did wrong. We never decided whether we were to specify the way the system was to work, or what the system was supposed to do. In an attempt to be clear we moved back and forth; slowly to this side and then slowly back to the other side. And yo-yo'ing back and forth we opened up holes concerning which even men of good-will could get into extraordinarily complex arguments.

The problem basically resulted from our going into design details in certain very complex areas of the application because we knew that there would be problems with them. This led our contractors to assume that we had given design details in other areas.

I don't think people really think about what the bidding instrument should actually do, aside from soliciting bids from contractors.

Perhaps there is a need to explore the role of the "chief purchaser" in software engineering projects. To what degree was the chief programmer supported in his singular view of the New York Times Information Retrieval System by a similar view of how the system was to be used (or designed) by someone at the New York Times?

This comes back to what Derrett says. You can at most prove what the man *says* he wants. But I don't know how to say what I want. I don't know, I really don't.

Organick There is an interesting case happening right now that illuminates many aspects of the subject of specification. That is the US government DOD specification of a programming language. It is a multi-year process involving many bidders, and even though the process is continuing, many people have their doubts about the whole thing.

Grosch What you do is you pass on the level of trust in a series of complex organisations. In the case of *Ironman* — the DOD language effort, the first responsible person in line is something like a deputy assistant secretary of defence who says something like: *"We are supporting too many different strategical and tactical programming languages. We should have fewer"*. Then he trusts someone below him to write this out in the form of a 20-page memo. This then is put out as a request for proposals, or a series of such requests. Some companies come in and offer to write the formal request for proposals for *Ironman*. There are levels of selection right there, and the payment for the contract at that stage may be several hundred thousand dollars. One of these outfits having been selected, it then writes the formal proposal, which is hundreds and hundreds of pages, and tries to make it logically and legally foolproof. In that they actually fail, of course. That, in turn, is put out as a request for bids to the large software houses which are then expected to finish the job for twenty, thirty, or fifty million dollars. By the time it is all done, the desire of the original deputy assistant secretary of defence has changed! It is impossible to go through that ponderous a system, through that many levels, and come out with anything useful. What you get in the end is one more unmaintainable and undesirable programming language so now you have ten instead of nine.

There is an assumption here about the delegation of trust. The assumption is that the people who come up with the language specification will not be permitted to do the actual work: *ergo* they will not make millions of dollars of profit, and they will be loyal to the person above them in the hierarchy. That doesn't really work. People have internal loyalties. They have private or local ambitions, such as getting the contract to do the next one of this sort. All this also perturbs the process.

What you have to do is remove one of those layers near the top and have a customer who pretty well understands his own requirements. He in turn may need a large specification, but there will be a genuine common interest and common semantics.

Shelness Let us keep in mind that those *Ironman* specs, among other things, specify how arithmetic is to be done. This way of doing arithmetic then absolutely dominates the way the language must be, the kind of control structures you can have. So the spec already defines too much about the language.

Talking to the Customer

Beitz I want to comment about a recent procurement that was put out by an agency of the government where to my knowledge they put in an eight man-year investment just preparing the bidding instrument. They didn't get a single viable bid. I was one of the people who were asked to bid. I took one look at it and saw that it was over-specified and contained an impossible wish-list. So I immediately put in an unsolicited bid to solve what I perceived the problem to be. (Which was another can of worms because they were not open to accept an unsolicited bid.) The interesting part of that monstrous bid was neither the over- nor the under-specification but the ridiculous size of the specification. The effort that they made. They started out not only telling you what to do but also how to do it and then how it will have to perform. That thing was absolutely impossible.

I question that you can ask the customer for a specification. I think that it is like drawing teeth. You have to get into a question-and-answer session with him and you have to ask him the things you have to know in order to resolve an ambiguity that you see. And when you have got the information out of him you have to play it back to him and make him agree that this is what needs to be done. And you can only agree to do something on which you have a concensus.

Shelness The specification of the sort of computerized solution that is required should be made jointly by both the customer and the computer expert. I see an area of conflict that perhaps relates to the subject of a professional code of practice. The customer says he wants such and such. The expert is of the opinion that the customer will better be served by something else.

Gram The expert always knows best! No matter what you think you want, he'll explain to you that you want something else! When I go to the bookstore I ask for a specific book. I don't tell the seller to give me a book, any book, about some subject. I definitely don't stand for his giving me book *B* when I specifically ask for book *A*.

Lee But that is a wrong analogy. For what I consider a better analogy: you don't just go to the doctor telling him summarily to remove your appendix. At least you'll give him the chance to examine you and to check upon your self-diagnosis. More typically, you'll just say *"I have this terrible pain in my tummy"*.

Spier A serious problem which I have observed is that the customer who knows an insufficient minimum about computers often specifies his request not in terms of his actual needs as he sees them but rather in terms of a tentative implementation, the way he thinks he would have understood and solved it had he been a software expert. And many software experts are gullible enough to take him literally and to build the thing for him. For example, the customer wants to indicate that there is an input. He thinks that he will make himself clearer if he talks about it as *"card"*; the software expert seems to hear an implied request for punched card I/O. And that's what gets delivered. Or the customer wants to talk about permanent storage, remembers the latest science fiction TV-program that he has seen where computers were spinning tapes like mad, and talks about the file concept using the word *"tape"* to make himself clear. And so he gets tape files instead of disk files. This may possibly entail a modification to the monitor to make the thing support tapes in the first place.

A second thing I observed is that we often talk about a customer's *"problem"*, such as the *"payroll problem"*, when in fact there is no problem. The application is well-defined and has been

performed manually for a long time. It only needs to be mechanised. Often we are too lazy to just spend the necessary time chasing down what actually is being done manually, and documenting it, and turning that into the first design draft.

Stoy So what would you suggest we do?

Spier About both problems, I suggest we get better at going and talking to the customer in his language. It is my responsibility as a system designer to talk the customer's language and to translate it into computerese. Another thing would be to make sure that the information you use comes not from the vice-president of operations of the customer firm but from the blokes who actually do the work.

Lee I see a big danger there, dating back to the early days of computing. As soon as they had a Von Neumann machine, people started asking themselves how they could use it. And they went into offices and saw all kinds of file cabinets and drawers and manilla folders. And they saw how it was done in the earlier unit-record equipment. Their first reflex was to emulate that, and thus never make any progress at all.

Shelness Coming from a long line of textile manufacturers, one of the things that always intrigued me is that the first generation of looms did mechanically mimic what was done by hand. Those early spinning and weaving machines did essentially what humans did. The second generation found better mechanical ways of doing it. The third generation machinery employs processes that have no manual analogue. This is generally true of mechanisation. I think this is happening to the computer mechanisation of manual processes. We should not start mechanising by inventing new processes. It never happened with other technologies.

Spier Yes, the first step must be a faithful but reliable simulation of manual processes. Especially because we must keep the door open for a mixed manual/automatic processing, for example during the computer phase-in. Also, you have to keep running in parallel for a while for verification, and have to be able to revert to manual if computer problems develop. And they always develop.

Gram Talking about meeting the customer halfways, it seems to me that perhaps there is need for another profession, namely the *"interpreting consultant"*. His job is not to make technical designs but to mediate between the customer and the technical designer, being responsible for there being a meaningful dialogue between the two.

Lauesen In commercial data processing you find such types.

Beitz I would like to get rid of intermediaries, and let the guy with the problem talk directly to the computer.

Derrett Sometimes the user does not understand what he wants; he only has a vague idea of what he wants. At other times he has a very good idea of what he wants, but lacks the language to communicate it. There is also the case where he knows what he wants, and he solved it to the best of his ability, and he asks you to do the rest. He has broken up the problem into *"sub-problems"* and he asks you to solve one of these. But the sub-problem may be more difficult to solve on a digital computer than the original problem was. This makes it awfully difficult to talk with the customer because at times you should take his technical suggestions very seriously and at other times not seriously at all. I think that the inevitable conclusion is that the software expert has first to have a good understanding of the customer's problem, before any technicalities are allowed to be considered.

Spier But here we come back to where we started. You negotiate, and pull teeth, and reach a concensus, and make the thing, and then you bring it to the customer. What if it is not what he thinks he wanted? And in the affirmative, how can you prove that this is a correct implementation of the specification?

Beitz I can only prove that the implementation does the specifics that I delineate. In other words, you have to have some exhaustive way of expressing the whole thing.

Lauesen I would like to comment on the question of software people making the specifications which the customer then has to confirm. This may solve the legal problem but it still leaves the problem that the customer will have to understand the specifications to which he is supposed to agree. Very few customers are able to do that. They are capable of understanding examples, but not the planned system. So they have to trust someone who is a professional.

Beitz I believe that if you have a simple notational instrument that the customer can understand then it is possible to communicate your intent. I will show you such an instrument in my presentation.

Writing Programs which may be seen to be Correct

Joe Stoy

Presentation by Joe Stoy

Writing Programs which may be seen to be Correct

Stoy We all know what *abstract* programs are, I now want to talk about *"concrete"* programs — reinforced-concrete programs. I want to talk about the work of a friend of ours called Brian Shearing. He is part of a software house called Alcock, Shearing and Partners. His firm has produced a sort of British *Consumers' Guide* to reinforced-concrete programs.

There is a British code of structural engineering practice, *CP110*, which tells you how big the beams have to be and where the reinforcements are, and so on. There are about seven proprietary computer program packages available to structural engineers for them to use to calculate these various sizes and positions and so on. There is one package from the Department of the Environment itself, and others from various software houses. This is too much of a good thing, and so Brian's firm thought that it would be good to do a consumer's guide. They could do tests on them, and, for example, run the same problems on all programs. These programs are fairly well documented with user manuals and all, and a chap at the end of a telephone you can ring up if you have any problems. So it was perfectly feasible to run the same problem on all programs. And of course they all gave different answers.

It turned out that most of them followed the code of practice, most of the time. And to be fair, most of the time when they did deviate from the code of practice they did so so dramatically that no engineer worth his salt would consider that solution at all. But some deviations were rather marginal. There were also all sorts of bugs in the programs.

So Shearing's firm gave the program producers the opportunity to correct their bugs, and all over the report there are little asterisks pointing to footnotes which say *"we understand that this is no longer the case in the current version"*. The other thing that they did was let the producers comment on the report, and printed their comments at the back. The Department of the Environment, which had commissioned the guide in the first place, said at the back that they had corrected the serious bugs in their own program. But, they went on to say, this case was a contrived example and it was worth noting that many bridges and beams and things designed using their program are in position all over the country!

Now, the question of course is: who is responsible if the bridge falls down or the building collapses? And the answer, in England anyway, is that responsibility lies with the engineer who chose to use the program. He is ultimately responsible for that beam. This means that the engineer has got to understand what the program is going to do. If he is going to make a conscientious decision to use a particular program then he has got to know about how it works, not only what it is supposed to do.

This is quite difficult nowadays, in part because the innards of some of these packages are proprietary secrets. But the firm reckons that this is a passing phenomenon; as software goes on, things that are proprietary secrets now will become published public knowledge. Secrecy is a temporary problem, but it is a trouble at the moment.

The moral of this story is that software programs have got to be readable. They also have got to be intelligible. The engineer who uses them has got to be convinced that the thing works.

Now you might say that he could have been convinced by reading the proof of correctness of the program, but that is a non-starter. Dijkstra said (I think it was in Aarhus, in 1973) that the proof of correctness of a program corresponds to the programmer's or reader's sense that it is correct in the same way that an Act of Parliament corresponds to one's sense of justice. There are

occasions when, because you are working with some fuzzy boundaries or because a particular decision is vitally important, you have to have recourse to the formal documents. That, usually, is a job for professionals — we unleash the lawyer to go out and hunt through the statutes. But if we have to do that too often then something is wrong, either with the Acts of Parliament or with the upbringing of our citizens.

The same thing should be true with programs. There might be very important programs or very intricate and subtle ones where we might need to have recourse to the whole mathematical panoply of proofs of correctness. But if we have to do that all the time in order to be convinced that the program is right then something is wrong either with the notation in which the program is written or with the way we have been brought up.

So what is the point of all the work that goes on about proofs of correctness of programs written in programming languages? I think that they provide a way of quantifying the simplicity, or alternatively the nastiness, of the programming languages concerned. I'll give you an example: if you write a little program, and to prove it correct takes 25 times the length of the program itself, then the language is awful, because someone who has written a program on the back of an envelope is not going to go through the 25-page argument. He's going to go through an intuitive argument, and if there are all sorts of hidden features of the language which get in the way of the intuitive argument then they are going to be forgotten.

Here is an example. This is the sort of axiom that the programmer who writes on the back of an envelope will have at the back of his mind (or perhaps even at the front of his mind) when he writes an assignment statement:

$$wp('x:=E', P) = P(E \text{ for } x)$$

This says that if you want the statement *'x:=E'* to terminate with some condition *P* true, you must ensure that a condition is true beforehand, which is like *P* except that every occurrence of *x* in *P* is replaced by the expression *E*.

This axiom is not always true in, for example, Algol 60. It is true unless:

1. *E* has side effects or fails to terminate.
2. *x* shares with another variable in *P*.
3. *x* shares with another variable in a procedure mentioned in *P*.
4. *x* affects any call-by-name parameter occurring in *P*.
5. *x* is a type-procedure identifier.
6. There is real-to-integer coercion.
7. There are rounding problems to affect the predicate.

This means that you can't prove any Algol 60 program that uses assignment, unless you go through every assignment and check that none of these possible pitfalls is relevant in that case. And of course the average programmer is not going to do that at all. Some of the trouble stems from the fact that Algol 60 is Algol 60; case number 5 is an obvious one. The call-by-name mechanism is also far more complicated than it need be. Almost always when one uses call-by-name parameters, call-by-reference would do. But the fact that call-by-name is there means that it cries *'Wolf!'* all the time and you have to check that none of the sophisticated problems is going to affect you. If you used call-by-reference you would have a simpler list of pitfalls to check for.

So one of the offshoots of this correctness proving activity should be that languages will get a whole lot simpler. So much simpler that this sort of pitfall won't occur any more, and that the formal proof of correctness need not normally be done: the language itself is a sufficiently

rigorous vehicle for the programmer's thought that he doesn't need to take the step backwards into the formal proof or the formal semantics. These do have to be there and available to verify that the language is all right, and the language designers have to make sure that the languages they are offering are going to be all right when used by the masses. But the masses need not have to worry about these things.

Now I'll show you a program written in a notation I heard of only last Thursday. This means that if you throw nasty questions at me, I can duck, and if you have criticisms of the language then I can say it's not my baby.

This is a notation, again by Brian Shearing, called *3R*, just like those famous three R's: *reading, 'riting* and *'rithmetic*. This is a notation for doing those things, in that order. It is a notation, principally, for reading programs written in it; because that is the most important purpose of a programming language notation. Secondarily, it is a notation for writing programs in. Lastly it is a notation to give to the computer to do something. Rather than go through the language feature by feature, I'll show you a sample program.

The block of program below is designed to search for a given value in a pre-sorted table of values. If a match is found, the position of the value within the table is to be delivered. If no match is found, the block is to fail.

The following example would set the status of execution **invalid** if the value were not found in table[1]..table[n], but otherwise would set "j" so that table[j] = value:

test j = find (value) in first (n) words of (table)

The program is as follows.

```
    let find in first words of be

    variable argument j is 1..1000
    invariable argument value is integer
    invariable argument n is 1..1000
    invariable argument table[1..1000] is integer
```

Because the values in the table are sorted the method of "binary searching" can be used, in which the range of values considered is repeatedly halved until a match is found. During the search the range of values to be inspected is table[first]..table[last]. The initial range is the full table.

```
    variable first is 1..1000
            first := 1
    variable last is 1..1000
            last := n
```

The main part of the block keeps searching until a match is found.

```
    repeat
            choose a value for j
    until table[j] = value
            adjust the range
    again


    where choose a value for j is
```

Assuming a fairly regular distribution of values in the table, the position to be inpected from the table is chosen to be that in the middle of the current page.

```
    j := (first+last)/2


    and where adjust the range is
```

If the value just inspected exceeds the current value then the new range is the lower half of the current range.

```
    if table[j] > value
            //first remains unchanged
            last := j - 1
```

If the inspected value is less than the given value then the new range is the upper half of the current range.

```
    if table[j] < value
            first := j + 1
            //last remains unchanged.
```

The blocklet *adjust the range* cannot be entered if table[j] is equal to the given value.

```
    otherwise fail          //computer failure
```

Before resuming the main loop, *first* is checked not to have overlapped *last* indicating that the value is not in the table (or that the table is not properly sorted!).

```
    if first ⩽ last
            pass
    otherwise fail
    end of find in first words of
```

That was the example. The first thing about it is that it has got a nice convention for comments: anything that begins at the left-hand margin is a comment. This makes it easy to intersperse the comment with the text. When Christopher Strachey and I published the operating system *OS6* we couldn't, we felt, put the commentary in with the text. This was partly because if you put it in with the text it obscured the structure of the programs. We relied on layout to show the structure, and if you intersperse commentary with the text then you lose the visual indication of the structure of the program — you can't show the nesting.

Well, Shearing has banned nesting. His position is that if you have some structural engineer who has got to understand the concrete program and you give him an

```
if ... then
   if ... then ...
         else ...
      else ...
```

he will not be able to work his way through the program. The trouble which the non-professionals get into with nested structures is the biggest hangup, according to Shearing, in getting them to understand programs.

Let us look at the loop in the middle of the program, which begins with **repeat** and goes on till **again**. That is the only form of loop that he has. All loops are in principle infinite. But somewhere in the infinite loop there can be a **while** or an **until** with some sort of condition. According to whether it is satisfied or not, you continue or you abandon the loop. And you see that where you might put in some nested structure *choose a value for j* and *adjust the range* you put a name there and that is called the name of a blocklet (the whole thing being a block) and you define the little blocklets by the **where** constructs which follow the main text of the block.

You can mention other blocklets from within a blocklet, and of course you tack on those extra **where**'s onto the list. There is probably some fuzziness to be cleared up about scopes of variables and so on, but it can no doubt be sorted out.

I told you of the only form of loop they allow. The only form of condition they allow can be seen in blocklet *adjust the range*. You have a set of conditions, introduced by **if**, and all **if** statements compulsorily end with the phrase **otherwise fail**. In this case you can get there only if *table*[*j*] is not less than, equal or greater than *value*; in other words, if something has gone wrong with the computer, because you could never have got here if the computer was right.

We might consider whether that is a good thing. The reason that Brian Shearing put this discipline in is not purely to guard against computer failures. He says that the discipline of having to consider each case explicitly whenever you compose an **if** statement (you notice that there is no default case and the exception is always an **otherwise fail**) catches more bugs in program composition than almost anything else you can think of. And that is the only form of conditional.

You can see that blocks have arguments, some of which are used for results. They are called **variables**. The others cannot be changed once they have been initialized and are called **invariables**. There are local **variables** and **invariables** as well.

I believe that we ought to be thinking about writing programs in notations as simple as this. The authors of *3R* do actually use it. For example they did a program to handle building contracts. Under conditions of inflation the normal ways of paying for a building contract break down, because when you make the initial estimate you can't predict what is going to happen. In Britain, at least, the government produces a set of indices every month about how the prices for various different bits of the contract have changed since last month. So they wrote a program for the

quantity surveyor to plug his contract into and be told how much he should be paid this month. The program's manual first of all describes the notation in which the program is written — the *3R* I've been talking about. The whole description takes only twenty pages, of which two are syntax charts. And then comes the program text, which is well interspersed with comments and forms the bulk of the book.

Then, they reckon, it is nice to put the program text through a word-processing system to remove all the comments, and that gives a much shorter program that they can take a computing engineering kind of look at and get some sort of a feeling for how the program is going to run.

Then they transliterate it into Fortran or Basic.

You see, this isn't a programming language with a compiler. It is a notation. They don't want to get into standardization and the like. They have not yet got a lexical analyzer or a syntax analyzer for the notation. They simply transliterate the text into the language their customer wants, but have the discipline that they don't allow any comment in the final text except for some cross-references to the 3R text. That forces any analysis of what's going on back onto the higher level version of the program.

They are a two-man software house, so they have no facilities to deal with maintenance problems. Therefore they have this way of making programs which obviously and believably do what they are supposed to do, especially in their customers' eyes.

The language is a little bit wordy, but not because of the comments. The comments are fine and are rooted in a respectable tradition — that is how you write mathematics textbooks.

The relevance of all this to program correctness is that here is a way of simplifying the job of proving programs out of existence. The programming language itself will be the notation in which the analysis can be done, an analysis that will be rigorously right. There is no point in somebody believing that his program is correct just because he's read it, when the language features that are underneath the surface make a chimaera of the apparent correctness of the text.

Natural Languages and Readability

Organick As you presented your arguments for readable programs, I kept thinking of COBOL. Don't you think that this is what Grace Hopper had in mind when she designed the language?

Stoy Yes, but I present a stronger requirement: that the notation has to be both intelligible and rigorous. There has to be some theory behind it. Regardless of how it is dressed up, it has to be mathematics.

Organick I question that it could be that, because you are using non-mathematical variables.

Stoy By mathematics I mean the kind of rigorous discipline that makes it possible to prove something; the discipline of thought whereby you can convince yourself that the program is doing what it is supposed to do. If you concentrate too much on the naturalness of the language, as is the case with COBOL, you are bound to get some of the vagueness and ambiguity which are the glory of natural languages.

Grosch Poetic richness, please. Not ambiguity.

Gram You talk about using concise language rather than natural language. I think it was Mike Woodger who said: *"You cannot talk about the real world in a very concise language."* There has to be some fuzziness if you want to describe the real world.

Professional Responsibility and Judgement

Grosch Stoy has raised the question of who is legally responsible for the reliability of software products. Ultimately, a code of professional practice, and professional certification, at least for some of us, will be unavoidable.

There is that professional civil engineer who signs-off that design and is legally responsible if the bridge falls down. He will do so without reading the program (to come back to Stoy's original scenario) if the program was signed-off by his counterpart in the software engineering profession whom he will be legally entitled to trust. Then both will be held responsible for the bridge and there will be much squabbling as to who is really to blame. But at least it will be more than we have at present.

Shelness But who in the software house do we trust to sign-off the program? In civil engineering firms there is usually a hierarchy, and it is a senior partner who signs on the dotted line. He really understands what is being built. We have the problem that many so-called computer experts are not competent to sign-off programs.

Grosch There are software organisations where the supervisor reads his juniors' code, and where the manager at least skims through the work passed on to him by the supervisor. Another important tool should be the intuitive professional judgement. In engineering places you have this guy who looks at people's work without really rechecking detail calculations and who says something like *"don't you think this needs a heftier bearing?"* because he has a feeling of wrongness just by looking at a design.

There ought to be more of that in our profession, we have been doing so much of it, for so long a time. We do have a remarkably developed sense of right or wrong. And you can look at some code and see how patently wrong it is. So why didn't somebody look at it when it could have been changed?

Shelness There is something about conventional (as distinct from software) engineering designs that allows professionals to gauge their rightness or wrongness intuitively. We don't have that in software. There may, as Grosch put it, be a feeling for obvious wrongness, but there is no feeling for a thing being right and if there is it should not be trusted blindly.

Stoy If we start publishing programs — as I am advocating, then we will be able to examine the works of great software engineers and perhaps learn something from them.

Stoy Somebody said the other day that when there is a billion-dollar suit against a motor car manufacturer because of the fatal car crashes caused by faulty software in the microprocessors that control the cars — then will the practical usefulness of correctness and reliability become an indisputable truth.

Beitz Then we'll have software malpractice insurance. And obtaining such insurance will depend on one's ability to prove correctness.

The Nature of Software Engineering

Lee Throughout this workshop we keep making an analogy between software and mechanical or civil engineering. They have two powerful tools available. First, they can test a thing to destruction. And second, they have recourse to analogues, either physical ones or computer simulations, which they can use for verification.

They have the discipline to test things in the light of at least two theories. For example in testing a bridge they can first use Hook's law and do an elastic approach, and then do a plastic approach. If they get identical or sufficiently close results, then they are in good shape.

Just concerning these dual approaches. I observed DOD computations which for reliability were run on two independent machines. But it was the same program, using the same approach. Why not run something else on the second machine so as to check on the software as well, comparing the results of two different approaches?

Beitz You forgot to mention that that engineer also has a safety factor of at least ten. He can allow himself to attach significance to the thing's failure to collapse under certain loads because the normal useage load is only a minute fraction thereof.

Gram It is wrong to compare programming with engineering. Programming and the output of programming are both intellectual in nature and compare much more closely with mathematics. The mathematician *does* have a feeling for the apparent rightness of a proof.

Lee No. An engineer is a person who designs bridges, he doesn't build them. Thus design is an intellectual exercise and is comparable to the exercise of program design.

Flynn The analogy between programming and mathematics disturbs me a lot. I believe the two to be very different. A theorem has to be true. It matters not at all whether the proof is inelegant or tedious or plain ugly, so long as it is impeccable and so long as the theorem is true. If it is false then it is not a theorem. A program has to have a meaning, and it has to run satisfactorily in the sense of correct outputs and of reasonable execution time and other such considerations. Because of that, as Stoy is saying, the program's text has to be readable and of obvious correctness. I cannot see the analogy between this text and the proof of a mathematical theorem.

Discussion B.2

Stoy If you compare computer proofs with mathematics, that analogy can become dangerous. For one thing, the mathematical proof and its theorem usually fits into a theory. The mathematicians go away and figure, within that theory's framework, that if that theorem is true then they may also prove such and such. This may lead them to an *"oh, I don't believe that!"*, after which they go back and take a closer look at the proof. Whereas with a program, it is at the sharp end. Once you have produced that program you run it. It doesn't fit into any larger theoretical framework. Well, sometimes you may look at the results and say *"oh, I don't believe that!"*, but very often there isn't anything in the light of which it can be examined. So you just put the program out. So it is more difficult to get the bugs out of program proofs.

Of course, these 20-page mathematical proofs themselves sometimes are wrong. There was a supposed theorem in algebraic geometry where there was first published a nice long algebraic proof; then a fairly complicated analytic proof; then a nice long sort of text-book proof; and then a counter-example! But proofs do get shorter, and if there had not been that counter-example we would no doubt have seen further effort towards making that proof more compact, and that might have helped to find the bug.

Gram Flynn and Stoy seem to take a very defeatist point of view. I agree much more with Lee in his saying that we may envy the mechanical engineers their different tools for checking constructions. But having no such means available we must resort to formalization and proof methods like those used in mathematics.

A Disciplined Approach to Solving Problems

E. Henry Beitz

A DISCIPLINED APPROACH TO SOLVING PROBLEMS

by

E. H. Beitz

Visiting Scholar, University of Utah, Salt Lake City, Utah, U.S.A.
Computer Systems Consultant, Saint Paul, Minnesota, U.S.A.

The process of designing a solution to a problem is difficult to describe. Prior experience and a disciplined review of the pertinent data and relevant alternatives can help to stimulate creative intuition. A mechanism for disciplining the reviewing process - in the form of decision tables - will be put forward. The same mechanism can be shown to be a useful guide to developing tests of the solution, enhancing and otherwise altering the system, and for communicating the intent of the solution to non data processing people. All of the attributes of well-structured programs are easily retained and there are numerous possible alternatives for generating the system that carries out the solution.

The most important benefit to be derived from using a computer is the uniformity and discipline that it brings to a procedure. Programs invariably fail when situations that "can't happen" do, or when conditions that were not considered arise, (assuming, of course, that what was considered, was handled correctly). The methodology presented in this paper does not guarantee that a user will be successful because ultimately the discipline is self-imposed. Neither is it a sure-fire recipe for solving problems; it is rather a tool - a tool to help problem solvers organize their approach to implementing their solutions.

We all tackle a problem with a rag-bag of ideas and some a priori experience. Brinch Hansen (1973) is one of the few designers who honestly admit that the elegant algorithms that they publish are really the very highly refined products of a rather complex process. (A process which is sometimes difficult to understand.)

Parnas (1971) has suggested that the data be considered as the major element in determining how a solution is partitioned. While essentially agreeing with him, it is often difficult to decide on the set of data elements that will be required until a particular representation and/or algorithm is chosen!

A procedure may be treated as a finite state automaton. In order to model the finite state automaton one must make an initial partitioning - the set of states, the input alphabet, the output alphabet, the transition mapping from state to state, the starting state, and the final state must all be chosen.

The methodology proposed below also requires an initial partitioning. The first attempt at partitioning may bear no resemblance to the final one; the important thing is that any partitioning that the designer feels might be important will suffice. An attempt to ensure that everything has been thought of will simply block the problem solving process.

INTRODUCTION TO THE METHODOLOGY

It is proposed that an unlimited entry decision table be used as the basic notation for representing the solution to a problem. The decision table may be thought of as an analogue for a finite state automaton. To ensure the completeness of the set of states some constraints will be introduced into the process of determining the state of the automaton. The input alphabet will consist of a set of equivalence classes, each one of which is a sequence of actions that may or may not alter the environment. The environment comprises a set of data elements. The transition from one state to the next cannot be represented as a simple mapping but must be determined by examining the environment. During the process of determining the state it is vital that the environment is not altered! Each state implies the invocation of one, and only one, of the equivalence classes in the input alphabet. The application of the sequence of actions of the equivalence class may alter the environment, which in turn will determine the next state, which in its turn implies the selection of the next sequence of actions to be executed. This cycle of events either does or does not terminate; but that depends on the nature of the problem.

CONDITIONS, RULES AND THE PROBLEM SPACE

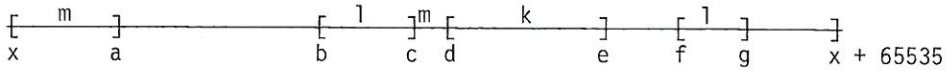
The first step is to try to define the set of states that will completely cover the problem. This set of states will be called the problem space. (This too is an exercise in partitioning.) One or more conditions need to be decided on. What a condition is, is best illustrated by an example: Suppose that a 16-bit variable named v represents a positive integer in the range x to x + 65535; suppose too, that depending on which of a number of intervals v is in, the procedure that is to be carried out differs. The defined intervals are:

k, between d and e inclusive;

l, between b and c inclusive and between f and g inclusive;

and m, between x and a inclusive and between c and d.

These could be represented as a number line as follows:



Note that it is a true partitioning in that the intervals are disjoint. But there are some values that v could take for which no defined interval has been named! This discontinuous interval must be named in order to ensure completeness. Let us call it n and define it as:

n, when v is not in either k or l or m.

It isn't necessary to know all the details about a condition in order to start solving the problem. Neither do all the conditions have to be known at the outset. For each condition that is named it is only necessary to provide:

- a name - to identify the condition,
- an exhaustive set of alternatives - regardless of whether the partitioning is too fine or not fine enough,
- an environment - the set of variables that must be referred to in order to determine which alternative applies,
- and a brief description of what it is that the condition determines.

Let us call the set of alternatives for the condition C1 the set CA1. Now, if m conditions are defined then the problem space is defined by the cartesian product of the sets of alternatives:

PS = CA1 X CA2 X ... X CAm.

Each state that is a member of the problem space is known as a rule. No matter how many new conditions or how many new alternatives are added to the problem space, it will always be possible to identify exactly which rules are affected! This means that those parts of the solution that have already been dealt with will in most cases not be altered when the new situations are discovered. The problem space is combinatorially complete and even when the conditions or alternatives are permuted every rule will still be present.

It is important that a strict enumeration of the rules be possible. To make this possible it is necessary to order each set of alternatives. This is simply achieved by mapping each set of alternatives onto a zero-based index set. This serves to give the resulting decision table representation of the solution a desireable regularity, which in turn makes it possible to recognize patterns by eye-balling the table.

ACTIONS AND EQUIVALENCE CLASSES

Before considering how the actions are related to the conditions and rules let us look at how an action is defined. In a manner similar to the conditions it is necessary to provide some information for each of the actions, as follows:

- a name - to identify the action,
- a reference environment - the set of variables that are read by the procedure that implements the action,
- an affected environment - the set of variables that might be altered by the procedure that implements the action,
- a brief description of what the action accomplishes,
- and the set of equivalence classes in which the action is included.

Each rule will require that a specific sequence of actions be executed. The similarity of the sequences for a number of different rules is precisely what makes the computer so useable. One of these action sequences defines an equivalence class, and the equivalence class itself is represented by two sets: the action sequence defining the equivalence class, and the set of rules for which that action sequence is invoked.

These action sequences may not in fact be strict sequences! Provided that two actions are not sequentially dependent and that the entire environment of each is disjoint with respect of the others affected environment, there is no reason why both of them cannot be executed concurrently. A notational device that allows this concurrency within an equivalence class to be represented will be found in Appendix A.

THE PROBLEM SOLVING PROCESS

In an unpublished working paper (Beitz 1974) I proposed that an interactive system for solving problems represented in decision table form, be implemented. The method may best be summarized as an attempt to permute the conditions, the sets of alternatives and the actions so that those which are similar are adjacent to one another. The problem space for most solutions is usually too large to represent on a single piece of paper in its complete form. However, after reducing the table by successive permutations of its constituents, most problem solutions seem to fit on a sheet of A4 paper (without having to resort to very small printing).

An example of a hypothetical problem solution will be presented below; first in its complete form and then in what might be called its final form. We will assume that the problem space is defined by the three sets of alternatives CA1, CA2 and CA3:

$CA1 = \langle d, e, f, x=\overline{dvevf} \rangle$

$CA2 = \langle h, j, k=\overline{hvj} \rangle$

$CA3 = \langle g, \bar{g} \rangle$

The resulting problem space will look like this:

C1	d						e						f						x					
C2	h		j		k		h		j		k		h		j		k		h		j		k	
C3	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$
RULE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Suppose that the set of actions over which the equivalence classes are defined is:

{A1,A2,A3,A4,A5,A6}

Let us also assume that whenever both A2 and A4 are members of the same equivalence class, A2 must be completed before A4 is begun; expressed as a constraint as follows:

<A2,A4>

The complete initial table might be:

C1	d						e						f						x					
C2	h		j		k		h		j		k		h		j		k		h		j		k	
C3	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$
RULE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A1	✓			✓			✓	✓		✓				✓		✓				✓		✓		
A2	✓	✓	1				✓		1				1	✓	1				1	1	1			
A3		✓	✓	✓					✓	✓				✓	✓	✓					✓	✓		
A4			2	✓				✓	2	✓			2		2	✓			2	2	2	✓		
A5	✓			✓			✓	✓		✓				✓		✓				✓		✓		
A6					✓	✓					✓	✓					✓	✓					✓	✓
EC	0	1	2	3	4	4	0	5	2	3	4	4	6	7	2	3	4	4	6	8	2	3	4	4

The process of reducing the table may now be described. The designer might realize while dealing with a specific rule that some refinement is possible. For example, suppose that the rules were presented to the designer in the order of the complete table (above). When rule 11 is presented the designer realizes that whenever C2 takes alternative k then action A6 is executed, and what is more, is the only action that is executed. This may be verified for those rules already dealt with - namely for rules 4,5 and 10. There are only two possibilities: either the designer's insight is substantiated by this check or it is refuted! (The confirmation does not necessarily imply that it is 'correct'; this decision is the designer's. Refutation, on the other hand, is far more significant!) If the hypothesis is refuted then it becomes necessary to examine the contradiction. In the particular case presented in this example, if the designer decides that the insight was indeed correct, then it will no longer be necessary to consider rules 16, 17, 22 and 23. These rules are essentially done.

Situations like the one just described are the rule rather than the exception. The six rules that are affected by the acceptance of the hypothesis in our example may be collected together. This is done by changing the order of the conditions. Simply making C2 the primary condition at the head of the table will suffice to make the six rules adjacent. Any change to either the order of the conditions or the order of the alternatives in a single set of alternatives, will change the order of the rules. To conform to the strict enumeration the rules may simply be renamed. There will be a 1 - 1 onto mapping from the old rule names to the new rule names. The following problem space is identical to the one in our example above and both the old and the new rule names are shown:

C2	h								k								j							
C1	x		e		f		d		x		e		f		d		x		e		f		d	
C3	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$	g	$\bar{g}$
OLD RULE	18	19	6	7	12	13	0	1	22	23	10	11	16	17	4	5	20	21	8	9	14	15	2	3
NEW RULE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

After permuting the conditions of our example the final table for the solution to the problem might look like this:

C2	h						j		k
C3	g		$\bar{g}$				g	$\bar{g}$	p∈CA3
C1	dVe	fVx	d	e	f	x	r∈CA1	r∈CA1	r∈CA1
RULE	0,1	2,3	4	5	6	7	8..11	12..15	16..23
A1	✓			✓	✓	✓		✓	
A5	✓			✓	✓	✓		✓	
A2	✓	1	✓		✓	1	1		
A3			✓		✓		✓	✓	
A4		2		✓		2	2	✓	
A6									✓
EC	0	6	1	5	7	8	2	3	4

It is quite evident that this final table is a lot simpler to comprehend than the complete table. The actions and the conditions and their alternatives may be at as high a level of abstraction as the designer wishes. Each action or condition may be represented by a decision table in its own right. This process will be known as elaboration. There is an important difference between the table representing the elaboration of a condition and that of an action. The condition's table may not include any actions! The elaboration of a condition serves to do nothing more than select an alternative. In the case of an action the elaboration is a decision table similar to that of the problem itself with the constraint that the environment of the action and its elaboration are identical.

When a table invokes an action whose elaboration is that table itself, we have recursion. The important aspect of the whole scheme is that the problem is treated as a unified object and is essentially a nested hierarchy of sub-problems. Any one of a number of different programs may be coded to represent the solution and testing also becomes an orderly procedure. Ultimately as was pointed out in the beginning, the discipline is self-imposed but the tables make it much easier for the designer to live with this imposition!

APPENDIX A

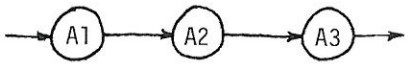
A NOTATIONAL DEVICE FOR REPRESENTING EQUIVALENCE CLASSES

A sequence of actions may sometimes have some strict order in which the individual actions must be executed. This is usually required when more than one action refers to the identical data environment. The designer must specify the order in which such actions are to be executed. The set of action names, delimited by commas, and enclosed in brackets of one form (say < and >) would indicate that those actions must be executed in sequence starting with the leftmost. When actions may be executed concurrently they will be enclosed in brackets of another form (say { and }). A single element in a sequenced action set may be a set of concurrent actions and similarly a single element in a set of concurrent actions may be a sequence of actions.

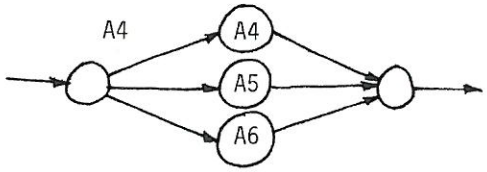
For example, if the action sequence for an equivalence class consists of A1 followed by A2 followed by A3 then the action sequence for the equivalence class is:

<A1,A2,A3>

or, graphically:



An example of a concurrent set of actions might be (graphically):



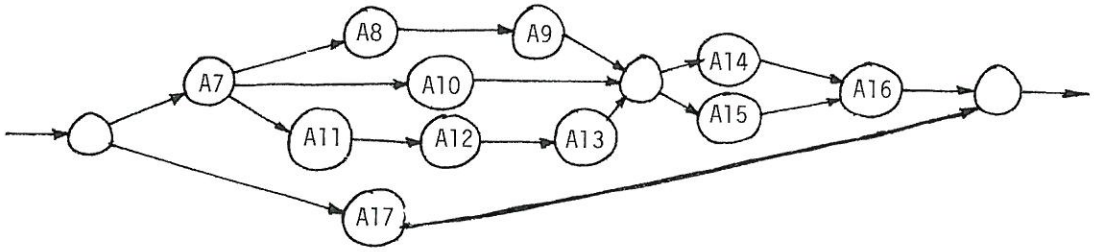
or, in our notation:

{A4,A5,A6}

which is identical to all of these:

{A4,A6,A5} {A5,A4,A6} {A5,A6,A4} {A6,A4,A5} {A6,A5,A4}

A more complex example combining both forms might be (graphically):



or using the notation:

{A17,<A7,{A10,<A8,A9>,<A11,A12,A13>},{A14,A15},A16>}

BIBLIOGRAPHY

Boitz, S. H., A proposed method for interactive problem solution using an exhaustive, combinatoric, decision table technique. Working paper - Decision Table Task Group of CODASYL, 1974.

Brinch Hansen, P., Operating System Principles. Prentice-Hall, 1973.

Parnas, D. L., Information distribution aspects of design methodology. In: Proceedings of IFIP Congress 1971, August, 1971. North Holland, 1972.

Discussion following Beitz' Talk

Beitz Now we want to turn this into a program. In order to do this we must be able to decide what state our program is in at each point in time. Making this decision must be an indivisible operation — no variable may be allowed to alter while we are deciding what state we are in. Once we have structured our problem in this way we can write a simple program to find out what state we are in and take the appropriate action.

When solving complex problems with time-dependent variables we have to be able to freeze the problem for a moment in time, if only conceptually, because it is only when we can freeze and look at a complex situation as a static object that we can make any decisions about what to do.

But in this case we must be sure that we consider every possible combination of conditions. I can imagine some of you saying that my case tables will become enormous. But I am not suggesting that you draw the table — you don't have to, that is why the sets are ordered. If every case was unique then your program would be too big to fit into the computer anyway, but we know that there will be a small number of different rules. Typically I find that a million different cases may be dealt with by 12 different rules.

Spier Sometimes it can be a very good idea actually to write out a whole decision table. It can lead to considerable insights into a problem, and maybe show us that the problem specification is inconsistant.

Bennett What practical experience have you had with this method?

Beitz I have used it for several large projects. For example I did a project which involved checking a computer program. The program contained about 180,000 cases and using a computer I could check them all in 16 milliseconds by treating each rule only once. A test program written by someone else ran for 300 minutes, and I found that it had only tested 20% of all cases. My test program produced a trace of what it was doing, and it usually identified a bug to within a few lines of code. The software was released in 1973, and only 2 bugs have been found since.

I find the tool difficult to teach. On the whole other people won't accept it until they are forced to by some terrible problem; but once they do accept it they become converts to the religion.

Spier This is a way of thinking. It is not a programming method which should be followed slavishly.

Beitz That's right, it is a tool for disciplining oneself.

SESSION C

Structuring Programs

What is a program's "structure"? Why do we structure programs? How do we structure programs? What benefits do we expect from structured programs, how can the benefits be measured, what measurements have been done?

User Defined Modifications in Dedicated Systems

Søren Lauesen

User Defined Modifications in Dedicated Systems

Søren Lauesen
DIKU, University of Copenhagen

Abstract

A dedicated computer system is designed for extreme reliability, which can be achieved if changes are kept to a minimum. However, users tend to ask for modifications prior to installation or while the system is running.

This paper compares several ways of implementing user changes without corrupting the vital parts of the dedicated system. The conclusion is that all available methods are inadequate. An improvement is outlined which reflects three independent purposes of tasks (job): to provide concurrent programming, to provide protection, and to provide dynamic exchangeability of program parts.

Contents

1. Introduction

A dedicated computer system serves a single purpose and runs continuously for months or years. Examples are computerized telephone exchanges, message switches, and power distribution systems. All of them are "real-time" systems.

In contrast, a real-time system for invoicing is not dedicated in the sense above, because it does not run continuously. In fact, it is stopped at the end of each day, and system modifications can be installed over night.

A dedicated system must have a high reliability. Vulnerable hardware components are duplicated, and the software must be well designed and well debugged. Although the system serves a single purpose, the user will require various modifications. Some of them are tailor-made in the first installation, but others should preferably be made while the system is running.

Any change to a dedicated system may degrade the reliability. This paper discusses various ways of installing the user's changes without corrupting the vital "dedicated" parts.

Figure 1 shows the model we will use of a dedicated system. The software consists of standard parts (the Operating System) and parts for the specific purpose (the Dedicated Part). An important task of the Dedicated Part is to maintain data structures which represent the surroundings supervised by the system. We will refer to these data structures as the Data Base, although they often reside entirely in the primary store, as opposed to commercial data bases.

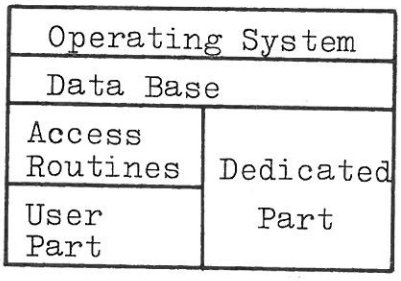


Figure 1. Components of a dedicated system.

The user's modifications are handled by the User Part which gets access to the Data Base or the Dedicated Part through a set of Access Routines.

We assume that the Dedicated Part and the Access Routines are virtually error free. This could be achieved in the long run if the user's wishes do not require modification of these vital parts.

Below we will discuss five methods for handling the User Part. The interpreter method allows the user to express himself in a restricted language interpreted by the Dedicated Part. The other methods represent the User Part as a task (a concurrent process) which can execute programs written in some general purpose language - even machine language. Of these methods, the shared area method gives the User Part direct access to the Data Base by means of hardware instructions. With the message method, all data is passed between User Part and Access Routines as message buffers. With the protected procedure method, the access routines are "protected procedures" which can be called from the User Part. Finally, the dynamic type method creates and deletes protected access procedures when the User Part "opens" and "closes" communication with the Dedicated Part.

We will evaluate each of the methods according to five criteria:

a) Flexibility
b) Protection
c) Dynamic exchangeability
d) Efficiency with traditional hardware
e) Commercial availability.

A flexible method must allow all kinds of user modifications. This clearly depends both on the language of the User Part and the set of Access Routines.

Protection depends basically on the Access Routines which must make sure that the Data Base is always consistent and that no User Part modifies data illegally. Protection could also be left to the language of the User Part, but

then the Access Routines are just considered part of the language.

Dynamic exchangeability covers two items: exchange of the User Part and exchange of the Access Routines while the system is running. The former is necessary to allow the user to add or correct modifications after installation. The latter is necessary if the Access Routines are not flexible enough, but exchanging them on the fly may of course degrade the reliability.

Efficiency with traditional hardware and commercial availability need not concern the computer scientist, but they are key criteria to the practician.

2. The Interpreter Method

With the interpreter method certain modules of the Dedicated Part are table driven (specialized interpreters). For instance, reports or pictures to be shown by the system are described in tables or files. By means of a problem oriented language the user can modify the tables thereby generating new reports or pictures.

The modifications are easy to make off-line, i.e. prior to system installation. However, the Dedicated Part could support an on-line version of the problem oriented language, which would give dynamic exchangeability of the User Parts.

The flexibility of the approach is low because the language is so specialized. On the other hand, the protection is excellent. The flexibility can be improved by providing a less specialized language, but from a certain point the user can legally express things in the language which are harmful to the integrity of the Data Base. Further, the efficiency of an interpreted language is much lower than the available hardware. For instance in power distribution systems, some user modifications may require large sets of linear equations to be solved. This clearly would be inefficient with a "general purpose" interpreted language.

The interpreter method with a specialized language can be rated as follows:

a) Flexibility: low.
b) Protection: excellent.
c) Dynamic exchangeability: possible for User Parts, not possible for Access Routines (i.e. the language cannot be exchanged on the fly).
d) Efficiency: good within the area supported by the language.
e) Commercial availability: (No special requirements to hardware or operating system).

The interpreter method with a less specialized language improves flexibility but reduces protection. The efficiency will be low when the flexibility is utilized.

3. The Shared Area Method

Many operating systems for real-time applications provide means for tasks (concurrent processes) to share an area of the primary store. In some systems a task can be restricted to read-only access to the shared area, but normally it gets both read and write access if it gets access at all (as an example, see Digital, 1976).

A dedicated system based on shared areas would look like this: The Dedicated Part would comprise a set of tasks maintaining the Data Base as a shared area. The User Part would be other tasks with direct access to the Data Base. With this scheme, it seems that we have protection because the user task cannot destroy the dedicated tasks. We also have complete flexibility because the user task can be programmed in a general purpose language and can access the Data Base in any way it likes. We shall see, however, that these claims are not true.

Protection is rudimentary, because ability to change a single field of the Data Base would allow the user task to destroy the entire Data Base. This would make the work of the dedicated tasks meaningless at best, and could even cause them to crash if they rely on consistency in the Data Base.

There are at least two limitations of the flexibility:
1) indivisible access to the Data Base is impossible,
2) peripheral devices used by the Dedicated Part are not available to user tasks.

The typical commercial use of shared areas seems to ignore the problems of indivisible access. For instance, programmers using such systems are surprised to learn that not even a double length integer can be read from the Data Base in a safe manner. The reason is that reading consists of two instructions, and if a dedicated task happens to get the CPU between them and updates the integer, the user task may get a meaningless result. If the user task attempts to update the Data Base, the lack of indivisible access may even corrupt a dedicated task relying on consistency.

If the user task wants to print a report, it can do so freely on devices not used by the Dedicated Part. Shared areas alone do not facilitate sharing of devices between dedicated tasks and erroneous user tasks.

The shared area method can be rated as follow:

a) Flexibility: excellent except for indivisible access and device sharing.

b) Protection: poor.

c) Dynamic exchangeability: good if task removal is supported by the operating system.

d) Efficiency: excellent.

e) Commercial availability: available in many systems.

4. The Message Method

With messages the User Part would again be programmed as tasks separate from the dedicated tasks. A user task gets access to items in the Data Base by sending messages to the dedicated tasks. These messages represent requests for data to be retrieved or stored. Retrieval will return data to the user task as an answer or in another message.

Referring to our general model, the Access Routines are now part of the dedicated tasks, and they are "called" by message passing.

The message method can overcome several limitations of the previous methods as we shall see.

Flexibility of the User Part is excellent because a general purpose language is used. Further, indivisible access and device sharing can be provided through suitable Access Routines matched with appropriate message conventions. For instance, a certain type of message could ask for a double length integer from the Data Base or ask for a consistent update of the Data Base. The corresponding Access Routines in the dedicated tasks make the indivisible access - assuming, of course, that indivisible operations can be implemented among the dedicated tasks.

Protection is potentially good because the user task is not allowed direct access to Data Base or dedicated tasks.

There is even a chance of providing dynamic exchangeability of the Access Routines. This would require that they are in a separate dedicated task with no vital duties. Provision could then be made for exchanging this task.

The weaknesses of the message method are related to dynamic exchangeability of user tasks, efficiency, and commercial availability; but different implementations have different weaknesses. The basic limitations of message passing are not obvious, and normally they are hidden by more serious design flaws in the particular implementation. Below, we will show typical design flaws first, and reveal the basic limitations later.

For our purpose, message passing involves four steps:

1) The sender (user task) calls the Operating System and asks for a message to be sent. As a result the message is queued up for the receiver (dedicated task).
2) The receiver calls the Operating System to receive the next message and as a result the message is made available in the receiver's store area.
3) The receiver calls the Operating System to return an answer. This can either be an explicit operation on the message received or sending of an endependent message (like step 1). The answer is queued up for the sender (user task).
4) The sender calls the Operating System to receive the answer. As a result the answer is made available in the sender's store area.

A common design flaw concerns handling of the case where the queue is empty when the receiver tries to receive a message. Some Operating Systems just return an indication that the queue was empty, leaving it to the task to suspend itself. The problem is that while the task decides to suspend itself, a message might arrive, but the task will suspend anyway. This problem is very difficult to program around, and it is often overlooked with time dependent errors as a result. The argument for the design flaw is that the task might have something else to do in case the queue is empty. However, a well-designed set of tasks should not have something else to do. Anyway, an indivisible operation "receive message or suspend" should be provided.

Another design flaw is to use a common pool of system buffers for message passing. The result is that an erroneous user task may use all buffers, thereby blocking other tasks from communicating. A simple solution is to give each task a separate pool from which buffers are drawn when the task sends messages. Both design flaws are present in for instance Digital's RSX-system (Digital, 1976).

Now we come to a more serious problem: Most implementations of message passing copy the message from the task

to a system buffer or vice versa in all the four steps above. This is not only a slow method, but it also makes an artificial restriction on the size of a message. For instance, if the user task wants the value of many variables, it must transfer them through many messages. If it wants an indivisible set of values, it will probably first have to issue a message "open indivisible access" and finally issue a message "close indivisible access". (Below we will discuss what happens if the user task forgets to close the indivisible access). The RC4000 message system has avoided the two design flaws, but still suffers from copying (Brinch Hansen, 1970).

It is possible to avoid copying of the message by passing a pointer to the message. To maintain protection with this implementation, we will either need special hardware or a mandatory high-level language. We also run into problems when the sending task is removed during message passing, because the receiver is left with an illegal pointer. (A high-level language solution is given in "Platon" (Staunstrup and Meiborg Sørensen, 1975)). Even if these problems are overcome, messages are slow compared to shared areas, because task switching must take place twice for each complete communication.

Now we come to a basic source of inefficiency: message identification. The dedicated tasks will normally be engaged in communication with several user tasks. When a dedicated task receives a message, the Operating System will typically identify the sender, but then the task has a rather heavy duty of translating this identification to a proper action to be taken on the message. One example is that some user tasks could be allowed to perform operations which are refused to other user tasks. Checking that the sender is legal must be carried out for every message. We will refer to this basic problem as the "identification problem".

The second basic problem is related to dynamic exchangeability of the user tasks: The dedicated tasks may need an indication of when a user task is removed. As an example,

consider the indivisible sequence of messages above, starting with "open indivisible access", followed by ordinary "retrieve" messages, and terminating with "close indivisible access". If the user task is removed during this sequence, the dedicated task never gets the "close". Of course, the dedicated task could use a time-out supervision (watch-dog timer) to detect missing "close" messages. However, if the user task is replaced by a new version, the ordinary "retrieve" messages of the new version may be taken for continuations of the indivisible sequence.
We will refer to this basic problem as the "termination problem". A solution must ensure that opening of a communication is always followed by closing of the same communication, and closing should be quaranteed when the communicating task is removed.

Notice again that the basic problems are obscured by the flaws and inefficiencies of contemporary message implementations. With this in mind, we can rate the message method as follows:

a) Flexibility: potentially good (depends on the Access Routines).

b) Protection: potentially excellent (depends on the Access Routines).

c) Dynamic exchangeability: possible for Access Routines, troublesome for User Part because of the termination problem.

d) Efficiency: good within User Part, but low for Access Routines - basically because of task switching and the identification problem.

e) Commercial availability: Many systems have flaws spoiling the protection and the dynamic exchangeability. The efficiency is normally rather low.

5. The Protected Procedure Method

Some of the efficiency problems with message passing originate from the binding of protection to concurrent processes. Thus in order to change from an address space where the Data Base is not accessible to an address space where it is accessible, we have to change processes (switch tasks).

Other efficiency problems originate from the binding of dynamic exchangeability to concurrent processes. As the sender and receiver can be removed independently of each other, most implementors choose to pass messages through a system pool, resulting in a high overhead and arbitrary restrictions on message size.

Protected procedures remove the binding between protection and concurrency. In our model, the Access Routines would be protected procedures which could be called by the user tasks. When inside these procedures, the address space includes the Data Base but also parameter areas specified by the calling task. As a result, the procedure can transfer data between Data Base and user task. Note that there need not be artificial size limitations on the parameter areas. Note also that no task switching takes place; instead the address space is changed twice: during call and during return.

Protected procedures require that dynamic exchange is independent of concurrency. As an example consider a user task which has entered a protected procedure and assume that the Operating System tries to remove the task. If the task is forced to leave the protected procedure, indivisible operations in the procedure can be corrupted. The solution is to remove the task code but allow the associated concurrent process to complete the protected procedure.

Dynamic exchangeability of the Access Routines would mean removal of the protected procedures. In this case the procedure code would be removed and possible processes executing the procedures would be forced to return.

A reasonably efficient implementation of protected procedures must rely on hardware protection or on a mandatory

high-level language. A high-level language implementation is provided by Concurrent Pascal (Brinch Hansen, 1976). Hardware protection must as a minimum supply a small set of base-limit registers, and a call of a protected procedure would then result in a change of the contents of base-limit registers. Proposals based on hardware protection are "capability systems" (Wulf et al., 1974) and "protected domains" (Spier, Hastings, Cutler, 1974). Neither in Concurrent Pascal, nor in the hardware based solutions is it clear how dynamic exchangeability should be implemented.

As far as I know, protected procudure methods with dynamic exchangeability have not been tested in real-life dedicated systems. So the following rating is tentative:

a) Flexibility: as for message methods (or better because Access Routines are not restricted by arbitrary parameter limitations).
b) Protection: as for message methods.
c) Dynamic exchangeability: seems possible, but the "termination problem" is not solved.
d) Efficiency: much better than message methods, but the "identification problem" is not solved.
e) Commercial availability: not available.

6. The Dynamic Type Method

In this section we will outline a variant of protected procedures which solves the "termination problem" and the "identification problem".

The idea is that when a user task opens a communication to the Dedicated Part, a private version of the protected procedure is created. Only the user task opening the communication will be able to call this version of the procedure, so the identification problem is solved.

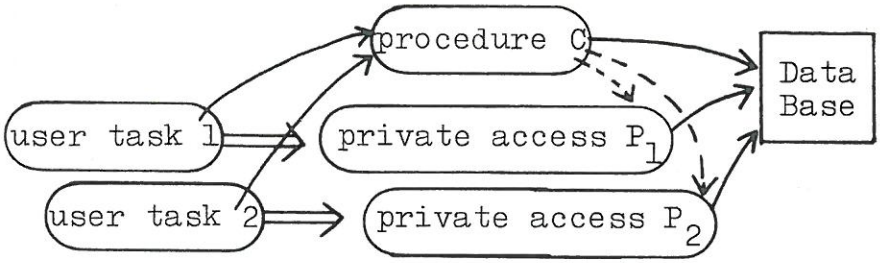
Actually, the communication is opened by calling one of the protected access procedures. It will do the elaborate checking and if access is granted, it will create the private protected procedure which from now on knows whom it is communicating with and about what.

The user task can close the communication by removing the private protected procedure. As part of the removal, a termination call of the procedure is performed (this corresponds to the "close message").

When the operating system removes the user task, it will also remove private protected procedures belonging to the task and not yet removed. Also in this case will the termination call be performed as part of the removal. This ensures that communications are always closed, and thus solves the termination problem.

Figure 2 shows part of a dedicated system based on these ideas. The user tasks have access right to procedure C (a protected access procedure), which has created private procedures P_1 and P_2. Both C, P_1, and P_2 can access the Data Base, but user tasks have to pass through one of these procedures. P_1 "knows" that it is only talking with task 1. P_1 and P_2 may have slight differences, for instance if task 1 is allowed to perform more operations than task 2.

The tasks "own" their private procedures in the sense that when a task is removed, the Operating System will also remove procedures owned by the task.



⟶ Access right
⟹ Ownership and access right
- - → Type instance

Figure 2. Part of a dedicated system based on dynamic types.

The relation between a private protected procedure "P" and the protected procedure "C" which created it, resembles the relation between a variable and its type. All private procedures created by C will perform similar operations, but they will work on different local data.

The type concept found in languages like Modula (Wirth, 1977) and Concurrent Pascal (Brinch Hansen, 1976) is illustrative. The creating procedure C would in these languages be a static type or "class" definition. The instances of the type would be the private protected procedures P. (It is somewhat misleading to use the term "procedure", because actually P would consist of local variables and a set of operations (procedures) working on them). However, in these languages the type is static (defined at compile time), whereas the creating procedure C above would do a dynamic check before creating a type instance. When the type instance is removed, a dynamic action is again involved (termination call). These differences are the reasons for using the term "dynamic type".

The principles of dynamic types are common in operating systems as we shall see. First, notice that a call to the operating system is a call of a protected procedure: The address space is changed to include access to data structures inside the operating system.

Next, notice that operating system calls like "open file" create something like a private protected procedure for the calling task. The local variables of the private procedure are found in the "system control block".

Finally, the task can remove the procedure by calling "close file". If it forgets it, the operating system performs the closing at task termination.

The dynamic type concept can be seen as an attempt to provide a similar mechanism for inter-task communication.

As far as I know, dynamic type systems do not exist at present, but it might be possible to implement them by means of "capabilities" or "protected domains".

It is rather meaningless trying to rate the dynamic type method. However, the aim is a method stronger than "protected procedures" because the identification problem and the termination problem is solved.

7. Conclusion

We have analyzed various ways of adding "unsafe" user code to a reliable system. The traditional methods were shown to have serious weaknesses, but the best is message passing - if properly implemented. We suggested that message passing could be improved by separating the concept of concurrency from the concepts of protection and dynamic exchangeability. The result was "protected procedures". A further improvement, called "dynamic types", would remove two basic limitations of message passing.

Acknowledgements

I am indebted to my colleagues at Brown Boveri for teaching me the problem of "user defined changes" in dedicated systems, and to O. Caprani and A.P. Ravn for our joint effort in trying to make a "dynamic type" system.

References

Brinch Hansen, P.: The nucleus of a multiprogramming system. Comm. ACM 13,4 (April 1970), pp 238-250.

Brinch Hansen, P.: The SOLO operating system. Software - Practice and Experience, Vol. 6 (1976), pp 141-200.

Digital Equipment Corporation: RSX-11M Executive Reference Manual, DEC-11-OMERA-C-D, November 1976.

Wulf, W. et al.: HYDRA: The kernel of a multiprocessor operating system. Comm. ACM, Vol. 17, NO. 6 (June 1974), pp 337-345.

Spier, M.J., Hastings, T.N., and Cutler, D.N.: A storage mapping technique for the implementation of protected domains. Software - Practice and Experience, Vol. 4 (1974), pp 215-230.

Wirth, N.: Modula: a language for modular multiprogramming. Software - Practice and Experience, Vol. 7 (1977), pp 3-36.

Staunstrup, J. and Meiborg Sørensen, S.: Platon: A high level language for systems programming. Recau, University of Aarhus (1975).

Discussion following Lauesen's Talk

Lauesen At the University of Copenhagen we are trying to design an operating system based on the idea of dynamic types. It is very difficult. There are a lot of problems to do with dynamic exchange of modules and it is not clear why they are there.

Grosch How big would your system be?

Lauesen If the kernel of our system is more than a couple of thousand instructions we won't implement it because it will be a bad design.

Grosch If we look to the future then the type of system you are talking about may be on several chips. Then the problem is to replace these chips on the fly without damaging the system.

Lauesen The problem changes if there are several processors because then there is concurrency and one needs a message-passing system to deal with the difficulties which that causes. In general we should try to avoid concurrency if possible.

For example, programmers usually ignore completely the problems of indivisible access to shared data. A programmer may access a 2-word object using 2 *load* instructions. He is very surprised when he finds out that the value could have changed between the two instructions. His typical reaction is to use a programming trick — read the first part; read the second part; and then read the first part again to check whether it has changed. This is because he doesn't have indivisible access.

Shelness Yes, and as soon as there are two independent clocks in a system then race conditions can occur. Life is much easier if there is only one.

Organick Is it reasonable to suggest that your dynamic type method can be regarded as a sophisticated message system where the private versions of procedures correspond to very cleverly formatted messages. Of course you don't have so many tasks, but that isn't essential.

Lauesen Yes, I would say that is right. But I want to enforce the termination messages.

Gram What is the difference between the Concurrent Pascal protected procedures and your dynamic types?

Lauesen When you create an instance of a class in Concurrent Pascal there is no means of checking whether or not that class may be created now. There is nothing to prevent you from creating it when you want to. In our dynamic types system you have the checking of access rights.

The type is a protected procedure itself. And it does some checking before it is willing to create an instance of itself. That is a crucial difference.

Møller-Nielsen This sounds very much like the class concept of Simula, where there is self-initialization.

Lauesen Yes, but initialization in Simula occurs *after* the instance is created — it is then too late not to create it. Also classes in Simula don't solve the termination problem.

Shelness Your private procedure idea seems to correspond to the idea of *streams* in OS6

Stoy Yes it is similar.

Lauesen The concept of dynamic types exists within most operating systems. If you want to open a file, you call the operating system which creates an open file for you. The operating system will from now on monitor your proper use of the file. If you don't close it upon termination, the operating system will do the closing. So the principle of dynamic typing is there. My complaint is that it is restricted to OS communications, and not applied more generally. What we need in order to structure systems generally into dedicated parts and user-specific parts is a similar inter-task communication mechanism.

Shelness In order to write programs using dynamic types we need a language which allows us to assign procedures to variables. There are very few languages which allow this, BCPL is one.

Shelness I wish to mention Carl Hewitt's *actor* model in which a message is something which at some point turns into an actor. This is really not much different from the ideas Lauesen expressed. I found out that if one considers systems to be basically message-switching systems, then one can build up on that model quite nicely. You can introduce parallelism by defining something called a *process* which contains a number of procedures that run to completion independently, communicating by messages. This simplifies the termination problem, because you have a termination procedure and you know that when it executes, nothing else can be going on in that process.

Lauesen That is: you issue messages, and they *"live"* each its own life. But what do you do for keeping track of memory if that resource is very limited? You can't just generate messages and let them live their own life. You need some sort of feedback to know whether or not there is memory available to support future activity.

Manthey The actor model is based on the assumption that there is no problem in having messages. You may have an infinite number of them. In order to make the thing work you have to arrange for it to look that way, using backing-storage if necessary. Doing this is the system implementer's problem.

Lauesen You need some central message pool. You can't just assume an infinite number of them because of the beauty of your model, blaming the system implementer for failing to provide infinite capacity.

Shelness In the end, the only things that really create messages in a system are external events. So we want to avoid a proliferation of messages as a result of a single event. In general, I think that we can deal with that. If there is an excess of messages the system may have to stop and something must be arranged, some sort of recovery, or maybe some discrimination that certain messages are more important than others.

Some Remarks on Software Systems Modifications

Janusz Górski

SOME REMARKS ON SOFTWARE SYSTEMS MODIFICATIONS

Janusz Górski
Institute of Informatics
Technical University of Gdańsk
80-952 Gdańsk, Poland

INTRODUCTION

Software systems exist as collections of representation levels. Two systems may have the same runtime representation but they remain essentially different because of the differences in other levels of their representations. Runtime representation is essential when we consider efficiency but it is much less important when considering for instance comprehensibility, flexibility or the useful decomposition allowing independent and parallel developing of parts of the system. Different system representations are constructed in order to form a "window" to the system which makes visible only several specific aspects in which we are actually interested. There are some basic concepts which proved to be especially useful in designing such the representations. The concept of the abstract machine enables us to represent the system as a hierarchy of programs written in abstract "machine languages" of different levels. This representation has proved its value in designing correct programs and in better comprehension of "what was it really done".
The concept of the process and the representation of the system as a collection of cooperating processes is very useful to study such problems as restricted resources utilization in operating systems. The concepts ofthe ownership and the priviledge are beneficial in representing the system as a collection of protective domains and are useful to consider problems of security and controlled information sharing.

In this paper we are interested in some problems related to changeability or modifiability of software systems. In order to study those problems more carefully we decided to base our discussion on the modular representation of the system as introduced in [1,2] . First we give the more precise motivation why we consider the modular representation as especially suitable to use when considering modifications. Then we argue that the supporting interface of a module has to be defined explicitely to allow checking the module interfaces against a redundant information and explicite localization which information is hidden inside the module. At the end we classify the possible module modifications into the three groups and discuss their impact on the extent of modifications in the whole system representation.

MODULAR REPRESENTATION OF SOFTWARE SYSTEM

We say that a given software system is modifiable if there exists such a representation of it that any we are interested in may be interested in may be introduced in such a way that

- it is easy to locate in the representation which has to be modified,
- there is no need to affect the remaining parts of the system representation,
- changes do not propagate through the system representation, i.e. the region of changes can be identified prior to the begining of modifications,
- there exists the method to guarantee that the changes introduced on that level of representation will have the desired effect in the runtime system behaviour, i.e. introduce the expected modifications to the runtime representation of the system.

In order to choose the representation especially suitable to satisfy the purposes as defined above, the first we have to decide on what criterion would we base the decomposition of the system. The most natural criterion is that of information hidding.

The whole system is represented as a collection of modules, each module hidding some important design decisions /such as algorithms, data structures, accessing methods, etc. / against the others and making visible only those informations which were consciously chosen to be exported outside the module. Such the decomposition enable us to encapsulate in a separate module every system function which is foreseed to be modified. As a result we obtain the modular representation of the system for which every modification is well localized inside an information module.

The basic problem related to the modular decomposition is that of a module interface definition. The module external interface defines the module to its "external world" and it is very important to specify that interface in such a way that it makes visible outside the module only those information which is necessary to use the module properly. All details about the module implementation ought to be completely hidden from that external world.

Modules are implemented using objects /functions, data/ offered /through the interfaces/ by the other modules and in that sense there exist intermodule dependencies which may be illustrated by the "makes use of" relation. Taking it in the mind we may talk about the supporting interface of a module, which is defined as the collection of objects /functions, data/ which are used in the actual implementation of the module and are exported through the interfaces of other modules. In our discussion of the modifiable software we decided to use the notion of the supporting interface explicitely. We define "supports" relation as the reverse of the "makes use of" relation. If a part of the external interface of the module A is used to support the implementation of the module B then we say that A supports B directly . If A is supported directly by B and B is supported directly by C then C supports A indirectly.

We consider modules as consisting of the three main parts, namely the external interface definition, the implementation and the supporting interface definition. The external interface definition part specifies which objects are exported outside the module and how to use them properly. The supporting interface definition

how to use them properly. The supporting interface definition part specifies which informations /exported by other modules to the public use/ are allowed to be used when implementing the module. The implementation part defines how objects exported through the external interface are realized in terms of objects supplied through the supporting interface of the module.

The explicit and precise definition of the supporting interface of a module may serve as a convenient tool to enforce the following restrictions on the overall structure of the system

restrictions on the structure of the graph defined by the "supports" relation. If we state that every module may be supported only by those modules which are not directly or indirectly supported by it, then the resulting graph represents the weak hierarchy of modules. If we assume additionally that each module supports directly only one other module then the strong hierarchy of modules results.

restrictions on the strength of interdependencies between modules /as defined by the "supports" relation/. Explicit definition of the supporting interface of the module enables us to preserve our control over what may be used to implement the module. As a result we may obtain more information about the actual intermodule dependencies. This information may be helpful in modifying the system in a controlled way, as will be discussed in the following sections.

MODIFICATIONS

Modifications of a single module may be divided into the two /not independent/ groups

implementation modifications,

interfaces modifications.

Implementation modifications

Of our primary interest are such the implementation modifications which are accomplished without any changes in both interfaces. It is one of the main designing goals to obtain such the modular decomposition of the whole system that its every future modification will touch only the implementation of a single module. Because it is impossible to predict all the modifications in advance, the sensible designing procedure is to decompose the system in such a way that

each task of the system predicted to be subject to modifications is enclosed in a separate module,

external module interface is as small as possible and allows to pass no information about the implementation details of the module,

supporting interface is as wide as possible allowing for a large variety of different implementations.

The modular representation of the system built up in accordance with those directions represents in fact the wide family of systems /in the sense of [3] / and that family is the one of our interest, i.e. it was defined by the anticipated system modifications. Passing from the one member of the family to the another one is accomplished by changing implementations of some modules.

Unfortunately, such the ideal situation is not necesairily true because in general it is not possible to predict all the system modifications in advance. Therefore we have to be ready that module interfaces will be modified as well as module implementations.

Interface modifications

Module interface modifications may be divided into the following three categories

interface extension

interface restriction

interface alteration

The extension of the interface rely upon adding new objects to be exported/supplied through the interface and/or extending /adding new facilities to/ the usage conventions of those objects without changing the existing part of the interface. The creation of the new module may be regarded as an /extreme/ example of a module interface extension. The restriction of the interface is a deleting of some objects and/or restricting the usage conventions of the interface. Removing a module is an example of the module interface restriction. The interface alteration may be regarded as a result of a combined use of the two preceding modifications.

The interface modifications are followed by the implementation modifications as an inevitable consequence. If we change /by extension, restriction or alteration/ the external interface of a given module, then some changes in its implementation are necessary. It is our wish to have such a situation that the size of changes in the implementation be proportional to the size of changes in the interface. The new implementation may require changes in the supporting interface as well. If such the changes are required, we have to examine if the information in demand is actually reachable in the system. If no, then some modifications of external interfaces of the existing modules or creation of new modules are necessary. In such a way changes introduced to a single module may propagate to other modules in the system representation.

Taking into account the propagation of changes we may divide single module modifications into the following groups

weak implementation modifications - require no changes in the module interfaces,

strong implementation modifications - require changes in the supporting interface of the module /make use of the new modules or exploit the existing ones in more extensive way/,

external interface modifications - in essence they require implementation modifications /strong or weak/ of the module as well as they change supporting interfaces /and therefore enforce the strong implementation modifications/ in all the modules supported by that external interface.

If we exclude the situation when two different modules mutually support each another /i.e. assume the hierarchical structure of modules/ then the weak implementation modifications are well restricted and touch only a single module. They may be regarded as the result of the activity to improve the efficiency of the module implementation. Such kind of modifications is welcomed and cause no troubles in the well designed modular system representation.

The strong implementation modifications requires changes in the supporting interface of the module. Those changes may have the following effects to the remaining modules

creation of new modules in order to support the modified interface ,

making use of those parts of the external interfaces of existing modules which were not exploited unlil now,

extending the external interfaces of some existing modules to obtain the desired information.

This kind of system modifications require much more careful examination of the net effect coused by the modification of a single module. Changes are not restricted to that module in this case. They may propagate to another modules and it is very important to localize them before the modification process start.

We may observe that if we have the hierarchy of modules /weak or strong/ then changes initialized by the strong implementation modification of a module propagate downwards in the hierarchy.

The external interface modifications cause the following effects to the other modules

> require a new implementation of the module which interface was changed. The old implementation of the module is modified strongly or weakly,
>
> change the supporting interfaces of all the modules which were supported by that one and enforce strong modifications of their implementations.

From the above it results that the external interface modifications propagate in both directions in the hierarchy of modules /downwards and upwards/. It is interesting to observe that in the strong hierarchy of modules the external interface modification of a given module affects only the level of the hierarchy immediately above the module. So, the extent of changes upwards of the hierarchy is restricted to one level only.

CONCLUSIONS

In the previous sections we have discussed the usability of the modular representation of a software system to introduce modifications in a controlled way. The overall system is represented as a collection of information modules, every module defined by its external and supporting interfaces and every module implementing its external definition /interface/ in terms of objects supplied by its supporting interface. We may put on a structure on that collection if we consider the relation "supports" between modules. In general such the relation must not define a hierarchy of modules. The situation when two different modules are mutually supported each by another may not be excluded because the relation "supports"

establishes the relationship between programs rather than modules. Therefore it is possible that some functions of a module A are implemented making use of some objects exported by a module B and other objects exported by A support the implementation of some functions of B. But in such the case the weak implementation modifications of A may propagate to B because the "supporting recursion" may occur between those modules. So, one of the main advantages of the hierarchical structure, the restriction of the weak implementation modifications to a single module, is lost in this situation. The other kinds of modifications /the strong implementation modifications and the external interface modifications/ also become much more complicated with regard to their net effect. The explicit supporting interface definition offers some means to overcome those difficulties and enable us to preserve our control over the propagation of modifications. For every module we may define the relationship between its external and supporting interfaces, illustrating which part of the supporting interface is responsible for the implementation of every object exported outside the module separately. This relation may be used to establish the net effect of every kind of a single module modification listed in the previous section. But the need for such the additional information illustrates the advantage of the hierarchical structure of the modular representation when considering its usability for system modifications.

To summarize, we argue that the explicit definition of the supporting interface of a module serves as the convenient tool to control and localize the region of modifications in the modular representation of a system. We recommend the hierarchical structure of the modular representation as more convenient to maintain when considering modifications.

REFERENCES

1 D.L.Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. CACM 15 12 1972 p.1053-1058 .

2 B.H.Liskov. A Design Methodology for Reliable Software Systems. AFIPS 1972 FJCC v. 41 p.191-199 .

3. D.L.Parnas. On the Design and Developement of Program Families. IEEE Trans. Soft. Eng. 2 1 March 1976.

Structuring Programs

Stoy What is the purpose of writing down the information dependencies?

Gorski If a system is to be modified, then the extent of the modification depends on the information dependencies in the system. If a system is small enough it is easy to see what these dependencies are, but once it gets large that is difficult. Therefore we need to write them down.

Stoy So your method tells you what modules must be changed when you make a modification.

Gorski Yes. But it also helps to suggest what are sensible modules when the system is being designed.

Spier These ideas seem closely connected with Lauesen's talk on structuring programs.

Lauesen I am not sure whether my talk had much to do with structuring programs.

Spier What do you yourself mean by *"structuring"* programs?

Lauesen *"Structuring"* a program means separating it into parts that can be managed. Now the kind of separation discussed in my paper is special because there are two different groups of people dealing with the two parts of the system. This is a special form of structuring.

Spier What about the problem of modification? This must be connected with structuring.

Lauesen Yes — we would like to separate the system into parts so that future modifications can be made by altering only one part at a time.

If we make a dedicated system without allowing the user to make changes to it then it may be possible to structure it so that we ourselves can add new facilities. If we are clever, and know what sorts of modifications our users will want, we may still be able to structure the system so that modifications are restricted to few modules. But the problem is that the users always come with requests for changes which will require the modification of many modules.

Lauesen [*to Gorski*] Have you had experience with modifying the system once it has gone into production use?

Gorski Yes — if we think back to what you were talking about in your presentation, then we have replaced both user and system parts of our systems. However, we don't do it on-the-fly as you do.

Lauesen There are always some modifications which affect many modules — does your method help here?

Gorski No, but in the worst cases I haven't seen anything else that does either. A modular design expedites a certain set of changes, but not all conceivable changes — there will always be modifications which affect many modules.

Derrett You say that the hierarchy is better than having mutual dependencies between modules. Have you come across situations where you cannot manage with a strict hierarchy?

...ski We have had situations where modules were interdependent, but after some thought we have usually managed to re-structure the system into a hierarchical one.

An Experiment in doing it again, but very well this time

Nick Shelness

AN EXPERIMENT IN DOING IT AGAIN, BUT VERY WELL THIS TIME

N.H. Shelness and D.J. Rees

Department of Computer Science, Edinburgh University

P.D. Stephens and J.K. Yarwood

Edinburgh Regional Computing Centre

Address:

The James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh
Scotland

ABSTRACT

This paper describes an operating systems re-implementation project. A rationale for re-implementing operating systems, and the structural changes being made to a particular large scale operating system (EMAS - The Edinburgh Multi-Access System) are described.

AN EXPERIMENT IN DOING IT AGAIN, BUT VERY WELL THIS TIME

> "Our problem is that we never do the same thing again. We get a lot of experience on our first simple system, and when it comes to doing the same thing again with better designed hardware, with all the tools we know we need, we try and produce something which is ten times more complicated and fall into exactly the same trap. We do not stabilize on something nice and simple and say "let's do it again, but do it very well this time."
>
> David Howarth [HO72]

INTRODUCTION

In the summer of 1976 a small group of teaching staff in the Edinburgh department of computer science decided, given the effect that they felt the ICL 2900 computer range was likely to have on the British computing scene, to examine and evaluate the system. The basis for this excercise, in addition to written information at various levels of detail, was an ICL 2970 computer that the department had agreed to house in its research laboratories on behalf of the university's computing centre in exchange for one hours hands on access a day during the prime shift. There were three aspects of the 2900 range that this group were particularly interested in studying.

1. The appropriateness of the order code as a target language for the compilation of one or more high level languages.

2. The appropriateness of the architecture as a vehicle for a modern virtual memory operating system.

3. The appropriateness of the both the order code and the architecture as a basis for a range of hardware implementations.

The group split informally into three overlapping sub-groups each of which dedicated itself to studying one of the three aspects. It was out of the second of these sub-groups that the project described in this paper grew.

The members of this second sub-group had a fairly clear idea of the sort of functions that a modern virtual memory operating system should provide, as well as the level of performance it should achieve based on almost ten years experience with these sorts of systems.

This experience had been gained primarily in the design, development, use and modification of the Edinburgh Multi-Access System, but was tempered also by the experience of others aquired both from the open literature and more directly in face to face discussion. Even with this experience, or perhaps because of it, the sub-group felt that the evaluation should consist of a study of the effect that the architecture had on a real operating system and not a study of the architecture in isolation. Three operating systems were identified as candidates for this evaluation.

1. The manufacturers operating system (VME/B).

2. The Manchester University Software System (MUSS 2900).

3. A re-implementation of EMAS for the ICL 2900.

The first candidate was rejected almost immediately largly because it was not self supporting, but also because its size and its performance made it difficult to detect any beneficial effects that the architecture might have had upon it.

The second candidate appeared initially most attractive. Versions of MUSS existed while a re-implementation of EMAS was merely an idea. It appeared to be well structured and efficient, and the use of MUSS seemed also to form a basis for collaborative research involving both the Edinburgh and the Manchester departments of computer science, but after much effort both in Edinburgh and Manchester it to was rejected as a candidate. There were three reasons for its rejection. The first was timing. At the time the Edinburgh sub-group needed it, the 2900 version of MUSS was still being implemented. The second was that the Edinburgh group felt uncomfertable with the software technology used in constructing MUSS, and with the impact this had on our ability to either modify or extend it.

Thus with some sadness at the failure of the MUSS approach, the task of re-implementing EMAS for the ICL 2900 was started. At this point the university computing centre expressed an interest in seeing EMAS re-implemented for the ICL 2900 as an alternative to the manufacturer's operating system. Without making any commitment to use a finished product, they were willing to commit staff to the project and a full scale re-implementation excercise was launched which had as its goal the development of a full production system and not merely an architectural evaluation.

RE-IMPLEMENTATION

By re-implementation, we mean re-programming and altering the structure of an existing system where necessary, so as to better implement the function of that system. We distinguish re-implementation from two other activities. These are the transportation of an existing system to new hardware, and the design of a new system. If we transport a system, we leave its function and

its structure unaltered. Re-implementation is not merely a combination of transportation and re-design, but is a unique activity in its own right. To paraphrase the title:- it is doing it again, but very well this time.

We believe that it was necessary to alter the structure of the existing EMAS system for three reasons:-

1. The existing implementation of EMAS on the ICL 4/75 is essentially a prototype. This is despite the fact that the system had existed as a product for over seven years. As with almost all prototypes, it had many rough edges, and parts of questionable quality. It should be possible, and in other branches of engineering it certainly is possible, to reconstruct these parts without altering the function of the system as a whole [WI76].

2. The function of the system had evolved over the last six years in ways that were not conceived of in the original design. This is not a unique phenomenon. As has been documented by Belady and Lehman [BE76], the function of many large programs evolves over time. This results eventually in a disastrous clash between the program's function and its structure. Belady and Lehman have indicated that they believe that this phenomenon is endemic to all large programs. Two examples of such change of function in EMAS are:-

   a. The transition of the system from being the centre of a computing utility to a node on a network.

   b. The transition of the system from possessing its own private file system, to sharing file systems with other EMAS systems [SH75].

3. Improvements in technology allow for simplifications in system structure. To take one example: in the existing implementation of EMAS for the ICL 4/75, drums are allocated on a page by page basis, and operate both as a read and a write cache for pages held on disk. There were two reasons for doing this. The first was that drums on the System 4/75 were relatively small (2 megabytes). The second was that the disks were very slow when compared with the drums (a factor of 20). Neither of these factors exist for ICL 2900 computing systems. The drums are larger (6 megabytes) and the disks are faster (only a factor of 4 times slower than the drums). Given these improvements in technology, we can allocate the drums in bigger chunks and dispense with the use of the drums as a write cache for the disks. We can use the same unit of allocation on the drums that we use on the disks, which though less efficient in the use of drum space considerably simplifies drum management. We can also eliminate the fiendish problems of inconsistency between the contents of the disks and the contents of the drums, and the complex code needed to move pages asynchronously from the drums

to the disks, at what we believe is the now acceptable cost of having to update the copy of a page held on disk each time we update the copy held on the drum.

THE PROJECT

The EMAS re-implementation project has been informal in organization, small in size, and relatively short in duration. It commenced in October, 1976 with the intent of demonstrating a system by June 1977, and generating a first release by April 1978.

The project has had ten members. A six man core consisting of four full time workers and two part time workers has implemented most of the system. This group have been augmented by a second group of about four, who were available to make specific contributions in an area of expertise (for example device handlers). The contribution of this second group has been very limited in time, to the extent that there have never been more than one or two of them active at the same time.

Why has it been possible to handle a project of this scale with such a small team and in such a relatively short period of time? We have been able to identify three factors. The first was that all the tools required by the project already existed, were effective, and could be trusted. The second and perhaps the most important was that all the individuals involved possessed a consistent and complete model of that which they were implementing. The third was the quality of the staff involved, who have on average considerably more than ten years system construction experience and a host of large projects to their credit.

GOALS

What did we hope to achieve beyond the construction of an EMAS system for the ICL 2900? This question has a simple answer. We wished to show the viability of re-implementation as an approach to operating systems construction. Historically, the construction of general purpose operating systems has been an extremely expensive and often unsuccessful exercise. We believe that several of the causes of high cost and low success are the inverse of those previously introduced as a basis for our intended success.

It is incredible that academics as well as manufacturers continue to implement production operating systems in new and often badly implemented languages. The implementation of MULTICS in PL/1, and EMAS in IMP are two historical examples. The implementation of the SUE operating systems in the SUE language, and ICL's SUPERVISOR B in S3 two more recent examples. Having ourselves once faced the problems of writing an operating system on the shifting sands of an initial language implementation, we did not wish to do so again, and believe it should be avoided at all costs. We have also observed

that a major problem in operating systems implementation, even with relatively small teams, is that team members do not share a common model of the system, individual parts of which they are implementing. One of the ways that this problem may be overcome is by first building prototype model systems, which can then be developed into products through re-implementation.

TOOLS

A brief section describing the tools used in the project seems necessary. In keeping with the philosophy of the project, there are no new tools being developed, but in contrast with many other operating system projects, the tools have existed from the first day. There are three factors that have made this possible. Firstly, we possess an existing EMAS system as a development base. Secondly, we had the advantage of two years software development on and for ICL 2900 computer systems, by several members of the project prior to the begining of the project. This resulted not only in their development of insight into the machine, but in the prior development and extensive checkout of compilers and diagnostic facilities. Thirdly we had the advantage of knowing which diagnostic facilities were likely to be useful and which were not. The advantages that accrue from these insights should not be under- estimated. There is an unrivalled opportunity in an operating system for the generation of a massive amount of diagnostic information which by its very scale becomes almost useless. As an aside, it has been our experience as teachers of undergraduates, that no matter how good students are at developing small to medium scale programs, they almost always fail in their first attempt at a large program, because they fail to build in the correct diagnostics. No amount of prior warning will convince all but the very best of them of the need.

The system is being implemented in the same dialect of the Edinburgh Implementation Language (IMP) [ST74] as was the initial EMAS system. This is being done despite the existence of more modern and consistent dialects of IMP [RO78] and the existence of languages such as CONCURRENT PASCAL [HA75] and MODULA [WI76]. There are three reasons for this choice. Firstly, we wish to transport some programs from the existing system. Secondly, we understood the language, both how to use it and how to compile it. Thirdly and perhaps most importantly it already existed, not only the language and a compiler, but source code formatters, trace packages and post-mortem diagnostic procedures among others.

We have already mentioned the need for diagnostic facilities to be embedded in large programs. The following is a brief description of those that we are including in the re-implemented system. These diagnostics can be divided into three classes: system tracing, language monitoring and post mortem error analysis.

All three forms of diagnostic are output from the lowest level of the system using the system print facility, which can output either

directly to a line printer or to files for later analysis or printing. The ability to output directly to a line printer is of the utmost importance during early system development, but diminishes as the system becomes robust enough to support its own development at which point the output can be analysed to provide error and event summaries for maintainence engineers and operations managers as well as system developers.

System tracing is extremely easy in any message-based system such as EMAS. All messages go through a single central mechanism, and therefore the c in EMAS are:-

a. The transition of the system from being the centre of a computing utility to a node on a network.

b. The transition of the system from possessing its own private file system, to sharing file systems with other EMAS systems [SH75].

3. Improvements in technology allow for simplifications in system structure. To take one example: in the existing implementation of EMAS for the ICL 4/75, drums are allocated on a page by page basis, and operate both as a read and a write cache for pages held on disk. There were two reasons for doing this. The first was that drums on the System 4/75 were relatively small (2 megabytes). The second was that the disks were very slow when compared with the drums (a factor of 20). Neither of these factors exist for ICL 2900 computing systems. The drums are larger (6 megabytes) and the disks are faster (only a factor of 4 times slower than the drums). Given these improvements in technology, we can allocate the drums in bigger chunks and dispense with the use of the drums as a write cache for the disks. We can use the same unit of allocation on the drums that we use on the disks, which though less efficient in the use of drum space considerably simplifies drum management. We can also eliminate the fiendish problems of inconsistency between the contents of the disks and the contents of the drums, and the complex code needed to move pages asynchronously from the drums to the disks, at what we believe is the now acceptable cost of having to update the copy of a page held on disk each time we update the copy held on the drum.

THE PROJECT

The EMAS re-implementation project has been informal in organization, small in size, and relatively short in duration. It commenced in October, 1976 with the intent of demonstrating a system by June 1977, and generating a first release by April 1978.

The project has had ten members. A six man core consisting of four full time workers and two part time workers has implemented most of the system. This group have been augmented by a second group of

about four, who were available to make specific contributions in an area of expertise (for example device handlers). The contribution of this second group has been very limited in time, to the extent that there have never been more than one or two of them active at the same time.

Why has it been possible to handle a project of this scale with such a small team and in such a relatively short period of time? We have been able to identify three factors. The first was that all the tools required by the project already existed, were effective, and could be trusted. The second and perhaps the most important was that all the individuals involved possessed a consistent and complete model of that which they were implementing. The third was the quality of the staff involved, who have on average considerably more than ten years system construction experience and a host of large projects to their credit.

GOALS

What did we hope to achieve beyond the construction of an EMAS system for the ICL 2900? This question has a simple answer. We wished to show the viability of re-implementation as an approach to operating systems construction. Historically, the construction of general purpose operating systems has been an extremely expensive and often unsuccessful exercise. We believe that several of the causes of high cost and low success are the inverse of those previously introduced as a basis for our intended success.

It is incredible that academics as well as manufacturers continue to implement production operating systems in new and often badly implemented languages. The implementation of MULTICS in PL/1, and EMAS in IMP are two historical examples. The implementation of the SUE operating systems in the SUE language, and ICL's SUPERVISOR B in S3 two more recent examples. Having ourselves once faced the problems of writing an operating system on the shifting sands of an initial language implementation, we did not wish to do so again, and believe it should be avoided at all costs. We have also observed that a major problem in operating systems implementation, even with relatively small teams, is that team members do not share a common model of the system, individual parts of which they are implementing. One of the ways that this problem may be overcome is by first building prototype model systems, which can then be developed into products through re-implementation.

TOOLS

A brief section describing the tools used in the project seems necessary. In keeping with the philosophy of the project, there are no new tools being developed, but in contrast with many other operating system projects, the tools have existed from the first day. There are three factors that have made this possible. Firstly, we

possess an existing EMAS system as a development base. Secondly, we had the advantage of two years software development on and for ICL 2900 computer systems, by several members of the project prior to the begining of the project. This resulted not only in their development of insight into the machine, but in the prior development and extensive checkout of compilers and diagnostic facilities. Thirdly we had the advantage of knowing which diagnostic facilities were likely to be useful and which were not. The advantages that accrue from these insights should not be under- estimated. There is an unrivalled opportunity in an operating system for the generation of a massive amount of diagnostic information which by its very scale becomes almost useless. As an aside, it has been our experience as teachers of undergraduates, that no matter how good students are at developing small to medium scale programs, they almost always fail in their first attempt at a large program, because they fail to build in the correct diagnostics. No amount of prior warning will convince all but the very best of them of the need.

The system is being implemented in the same dialect of the Edinburgh Implementation Language (IMP) [ST74] as was the initial EMAS system. This is being done despite the existence of more modern and consistent dialects of IMP [RO78] and the existence of languages such as CONCURRENT PASCAL [HA75] and MODULA [WI76]. There are three reasons for this choice. Firstly, we wish to transport some programs from the existing system. Secondly, we understood the language, both how to use it and how to compile it. Thirdly and perhaps most importantly it already existed, not only the language and a compiler, but source code formatters, trace packages and post-mortem diagnostic procedures among others.

We have already mentioned the need for diagnostic facilities to be embedded in large programs. The following is a brief description of those that we are including in the re-implemented system. These diagnostics can be divided into three classes: system tracing, language monitoring and post mortem error analysis.

All three forms of diagnostic are output from the lowest level of the system using the system print facility, which can output either directly to a line printer or to files for later analysis or printing. The ability to output directly to a line printer is of the utmost importance during early system development, but diminishes as the system becomes robust enough to support its own development at which point the output can be analysed to provide error and event summaries for maintainence engineers and operations managers as well as system developers.

System tracing is extremely easy in any message-based system such as EMAS. All messages go through a single central mechanism, and therefore the cost in code, if not in time, of tracing messages is small. As the flow of messages through the system is high, a certain selectivity is needed. Therefore one is able to nominate the subset of messages to be traced by specifying either their source or destination addresses. There are two other traces that the system

will generate. The first is a trace of the paging activity of nominated virtual processes. The second is a trace of CPU allocation to nominated virtual processes. All tracing can be enabled and disabled either from the operator's console or by any supervisory process. Nearly all of this was in the original version of EMAS and proved useful.

Facilities have always existed in IMP for monitoring programs in source language terms. (For reasons that we do not understand, it was believed, at the time EMAS was implemented, that such monitoring would be either impossible or too expensive in an operating system). This is not the case, and therefore source language monitoring is being included in the re-implemented system, though of course it will disabled by re-compilation in production systems. The facilities that source language monitoring provide should not be underestimated, as the alternative is a snap shot dump.

A similar approach was taken initially with respect to post-mortem analysis of errors. They may occur either after the failure of a process, or the failure of the entire system. In the case of process failure we again make use of diagnostic procedures that print the final state of the process in source language terms. In the case of system failure, we were able to produce a small post-mortem analysis without having to resort to a dump. This was achieved by using the restart facility provided by the hardware, to enter a post-mortem diagnostic procedure. Using activation records, and information provided when the system was built, this procedure was able to provide a limited printout. This consists of diagnostic cast in source language terms, register values, re-assembled machine orders from around the point of failure and annotated tables in a format which matches their structure rather than the width of the print line. In those rare instances when this is not enough, or the post-mortem procedure fail, we have to resort to the tedium of a full dump. We had the experience of de-bugging the first EMAS system using uninterpreted dumps as the main means of fault analysis; it was an unnecessarily slow and tedious process.

As the system became more robust it was felt that a two stage post-mortem procedure was more appropriate, both because there was more information needed which was increasing the size of the resident procedure, but also because greater selectivity was needed in chosing what to analyse. The first stage also triggered by invoking a restart dumps the store to tape fro which it can be analysed interactively at a later stage. While this is more appropriate in a production system. We would recomend the building in of immediate post-mortem analysis package to suport the early stages of system development.

EMAS

We do not propose in this paper to give a detailed introduction to the philosophy, design or initial implementation of EMAS. An

overview can be gained from [WH73] and information about various specific aspects of the system from [SH74, RE75, MI75, WI75, AD75, SH75]. It is nonetheless useful to note the various fundamental features that give the system its current appearance:-

1. Interactive working.
   The system is normally accessed from an interactive terminal, which it uses as its major source of control information. Initially this was a teletype directly linked to the system. Currently it may be one of a number of devices attached via a geographically distributed network.

2. Multiple virtual memories.
   Each user is allocated a 16 megabyte virtual address space.

3. Mapped files.
   Files are not accessed via a procedural interface (read block, write block, etc.), but by being associated with a range of virtual addresses and accessed as if they were memory.

4. Controlled sharing of information.
   Initially this was simultaneous file sharing in all modes of access between users of the same machine. More recently a limited form of simultaneous file sharing (read only files) has been introduced between users of a number of interconnected but disjoint systems [SH 75].

5. Transparent memory hierarchy.
   The system operates a four level memory hierarchy (tape, disk, drum, core), but the user is only aware of files and virtual addresses.

6. Minimal user constraints.
   The system attempts to restrain the user as little as possible in his or her use of files, languages, virtual addresses etc. There is a set of facilities provided by a standard subsystem, but the user may ignore them and easily provide his or her if he or she so wishes.

7. High user throuput under all loads.
   The system cannot allocate more resource than it possesses, but it attempts to allocate as much as possible to the users at all times. There is no thrashing due to use of local scheduling policies.

8. Repeatability and enforced fairness.
   The resources used by a job are dependent only upon its requirements, and not on other demands on the system. Therefore if a job is run again it uses the same resources. A user should get a fair share of the system determined by its own resource requirements and the number of users of the system. The greater the resources required the lower the priority.

STRUCTURAL CHANGES.

There are five structural changes being made to the system as part of the re-implementation exercise. They are:-

1. A decomposition of device control into three separate functional modules: device configuration, channel scheduling and the translation of logical transfer requests into physical device commands for each device type. The first two used to be included as sub-functions of a device handler whose major function was the third.

2. A re-modularization of virtual memory control from being partitioned primarily by function and secondarily by virtual process to being partitioned primarily by virtual process and secondarily by function.

3. Implementing DIRECTOR [RE75] as a set of priviledged procedures available in the user's virtual adress space as in MULTICS, rather than as a set of functions supplied by a separate process associated on a one for one basis with each user process.

4. Altering the connection and control of communications front-end processors and spooled devices so that communications spooled data transfers take place directly to and from paged virtual memory.

5. Giving each process two message addresses rather than one. One address being used for initial requests to a process and the other for replies to its own requests on other processes.

CHANGES IN I/O CONTROL

The need for the decomposition of device control functions became obvious on EMAS as the complexity of device interconnection increased. This complexity is considerably greater in ICL 2900 computer systems, for there is greater scope for dnamically re-binding peripheral device to particular controllers and device addresses.

In the existing system, a device handler would on initialization determine the configuration of devices for which it was responsible. It would build a data structure, descriptive of this configuration, for its own use in device and channel scheduling. In the re-implemented system both the configuration function and the channel scheduling function where one is applicable have been factored out of the device handler, leaving it to perform the conversion of multiple logical I/O commands into sequential physical device commands. (In the case of ICL 2900 drums, the hardware accepts logical I/O commands directly, and therefore no translation is required.) Figure 1 is a schematic of the old and the new structures.

In the re-implemented system, device configuration information appears in a read only segment which can be considered as part of the abstract machine on which the system runs. This is achieved by loading the system in two stages. Initially a small cut-down supervisor is loaded, which builds a data structure corresponding to the configuration. To do this it employs an inordinately complicated sequence of opoerations to determine the model and number of processors, memory boxes, memory modules, I/O controllers, the devices atrtached to them and all their possible interconnections. In addition to building this data structure the cut down supervisor responds to a small subset of operator commands, (ie. to perform device transfers). One of these commands causes the full system to be loaded from a nominated file passed the read only segment describing the configuration and then entered.

VIRTUAL MEMORY CONTROL STRUCTURES

The re-implementation of virtual memory control (paging) is perhaps the most fundamental change being made between the existing and re-implemented system. It does, however, only reflect in the structure of the re-implemented system, that which existed functionally in the existing system.

In EMAS, unlike other fully paged systems, page replacement is not a global function performed over all available page frames in the system, but a function performed separately for each virtual process in the multi-programming set, over a set of page frames allocated to each virtual process upon its entry to the multi-programming set. This functional organization is not manifest in the structure of the existing system, where page replacement and other virtual memory control functions are performed by modules in the resident supervisor, which are partitioned by function. In the re-implemented system, there is instead a module, called a local controller associated with each virtual process in the system. Though there is a local controller associated with each virtual process, it is only active while the virtual process with which it is associated is in the multi-programming set. virtual memory. Figure 3 is a schematic of the old and the new structures.

In the old structure each module was serially re-entarant and non-switchable. It was invoked by a message, and had to multiplex operations on behalf of many virtual processes.

In the new structure, each module is also serially re-entarant, though the code may be shared by a number of local controllers running in parallel. Each local controller is entered via a trap from the associated virtual process (a page fault is a synchronous event and not an interrupt) or upon a return from a call made by a local controller upon the global controller.

The global controller is effectively a monitor, or series of monitors, that handles page movement between the various levels of

the storage hierarchy, and the allocation of resources to a local controller for a period of process time. The global controller does not enforce this allocation, and depends upon the honesty of the local controllers in staying within their allocations. Local controllers must therefore be trusted programs and cannot be provided by a user at will.

NON-VIRTUAL I/O

The handling of non-virtual I/O primarily with interactive terminals but also with non-paged magnetic tapes, had always been a problem in EMAS. Spooled I/O, to line printers and from card readers etc., is not a problem as the user operates to a file interface. He nominates a file to be output, and accepts a file of input. The problem is to give the appearance of mapped I/O for interactive traffic.

Two approaches have been applied in the existing system. In the first, DIRECTOR copied data between a user's virtual memory and buffers in locked down real store. The transfer was performed in response to commands issued by the user's process and external I/O activity. This approach carried with it a high overhead, as a number of DIRECTOR pages were required to perform this operation, and if the data transfer was larger than the buffer size the process would be paged in and out of store merely to refill the buffer rather than to procede. This had the effect non only of incurring paging activity, but also of altering the identity of the user's working set.

This approach was modified with the advent of network operation, so that DIRECTOR now transfers data between virtual memory and inter-process messages, which are sent to or received from the communications front-end processor. As these messages are small all interactive traffic has to be fragmented into 18 character packets. As only limmited number of packets (10) may be sent ahead without acknowledgement, similar unneccessary paging activity is introduced for transfers of over 180 characters length.

In the re-implemented system we make an attempt to overcome this problem, and to integrate the handling of RJE traffic from the SPOOLing manager and inter-machine traffic within the same scheme. The SPOOLing manager currently multiplexes large blocks into the flow of inter-process messages to and from the front-end).

In the new scheme a special virtual memory is maintained by a special local controller called the 'communications controller'. This virtual memory is not accessed by a virtual processor, as is a user's virtual memory, but by the communications front-end processors to perform data transfers. The actual transfers are controlled by a protocol that guarantees that the requisit file pages are available in main memory. The transfer of these pages from secondary to main memory, and from main memory when not required is controlled by the communications controller through calls on the global controller. In

the case of interactive traffic these files (one for input and one for output) will also be connected into a user's virtual memory. The user will read input from the input file and write output to the output file, synchronization being performed by director calls which in turn send inter-process messages to the front end processor via communications controller. In the case of spooled I/O, the files will not be connected into another virtual memory. The advantages of this approach are that no pages need to be permanently tied down in main store, and that the unit of transfer between the front-end and the main-frame can be appropriate to that logical interface, while the unit of transfer handled by the system remains a page. To those who are worried that this might involve excessive transfer of a set of buffer pages, it should be pointed out that each interactive input transfer from the front end requires some action in the associated user's virtual process, because the front-end buffers messages, and will not transfer meaningless fragments. It is our intent, as all page sharing is handled by the global controller, that the user's local controller will request the relevant pages before the communications controller releases them. In the case of output, only the buffer pages are paged in and out as required and not the user and director processes as is currently the case.

TWO ADDRESSES PER PROCESS

The change to giving each process, both system processes and virtual processes, two message addresses is a simple one that perhaps does not require its own section, but the simplification that derives from this trivial change argues for its inclusion.

In the existing system, a process is invoked by a message addressed to it. If a process can receive a number of different messages,the body of the process will take the form of a case statement (Figure 3). As long as the process does not itself need to interact with another process this poses no problems. If it does, it will receive a reply multiplexed into the incoming message stream. As a system-process must relinquish control before another system process can be invoked, a single activity punctuated by a request on another process and the reception of a reply must be implemented by two arms of the case statement. If the process requires to maintain the context of the first arm while awaiting the reply, it must either queue all other messages received before the reply, or possess a means of saving multiple contexts. Neither approach is particularly satisfactory. By giving each process two addresses, we allow it to receive requests at one address and replies at the other. The system also provides a means of inhibiting and enabling a message stream by destination address. Using this facility a process may inhibit the arrival of requests while awaiting a reply, thereby using a simple global mechanism to eliminate the need for message queueing within the process or the handling of multiple contexts. Figure 4 is a schematic of the new structure.

CONCLUSION

We have described an operating systems re-implementation project, a rationale for re-implementing operating systems, and the structural changes that are being made to a particular system. At the time of writing (April 1978) the project is on schedule. All major functions have been implemented, and attention is now being paid to system performance evaluation and tuning.

REFERENCES

[AD75] Adams, J.C., et al.,
'Performance Measurement of the Edinburgh Multi-Access System',
Proceedings of the International Computing Symposium (1975).
Gelenbe & Potier eds., North-Holland

[BE76] Belady, L.A. and Lehman, M.M.
'A Model of Large Program Development',
IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 225-252

[HA75] Hansen, P.B.,
'Concurrent Pascal Report',
Cal. Tec. Report.

[HO72] Hoare, C.A.R. and Perrot, R.H., eds.,
Operating Systems Techniques,
Academic Press, 1972.

[MI75] Millard, G.E., et al.
'The Standard EMAS Subsystem',
Computer Journal, Vol. 18, No. 3, 1975, pp. 213-219.

[RE75] Rees, D.J.,
'The EMAS Director',
Computer Journal, Vol. 18, No. 2, 1975, pp. 122-130.

[RO78] Robertson, P.S.,
'The Portable IMP Compiler System',
to be published.

[SC77] Schroeder, M.D. et al.,
'A Hardware Architecture for Implementing Protection Rings',
CACM, Vol. 15, No. 3, 1977, pp. 157.

[SH74] Shelness, N.H. et al.,
'The Edinburgh Multi-Access System, Scheduling and Allocation Procedures in the Resident Supervisor',
RAIRO (Informatiomputer Science) B,3, Sept. 1975, pp. 29-45.

[SH75] Shelness, N.H. et al.,
'Dynamic File Sharing in an Interactive Computing Utility',
Proceedings of a workshop on data communications,
IIASA, Oct. 1975.

[ST74] Stephens, P.D.,
'The IMP Language and Compiler',
Computer Journal, Vol. 17, No. 3, 1974, pp. 216-223.

[WH73] Whitfield, H. and Wight, A.S.,
'The Edinburgh Multi-Access System',
Computer Journal, Vol. 18, No. 4, 1973, pp. 331-346.

[WI75] Wight, A.S.,
'The EMAS Archiving Program',
Computer Journal, Vol. 18, No. 2, 1975, pp. 131-134.

[WI76] Wilkes, M.V.,
'Software engineering & structured programming',
IEEE Transactions on Software Engineering, Vol. 1, No. 4, 1976, pp. 274-276.

[WI73] Wilkes, M.V,
'The Dynamics of Paging',
Computer Journal, Vol. 16, No. 1, 1973, pp. 4-9.

[WI76] Wirth, N.,
'MODULA, aguage for modular multi-programming',
ETHZ, technical report 18, March 1976.

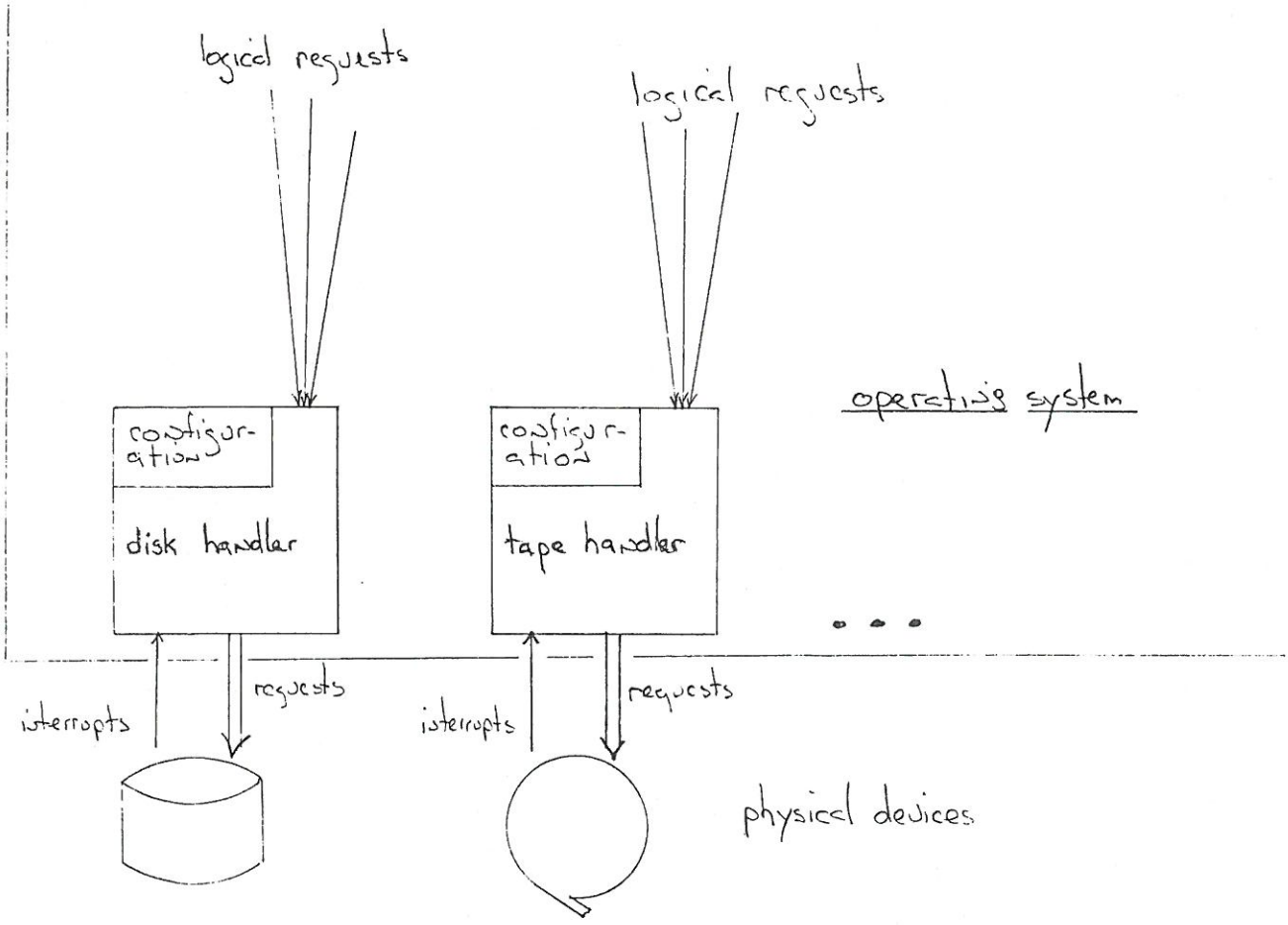


figure 1 a. current device control modularization.

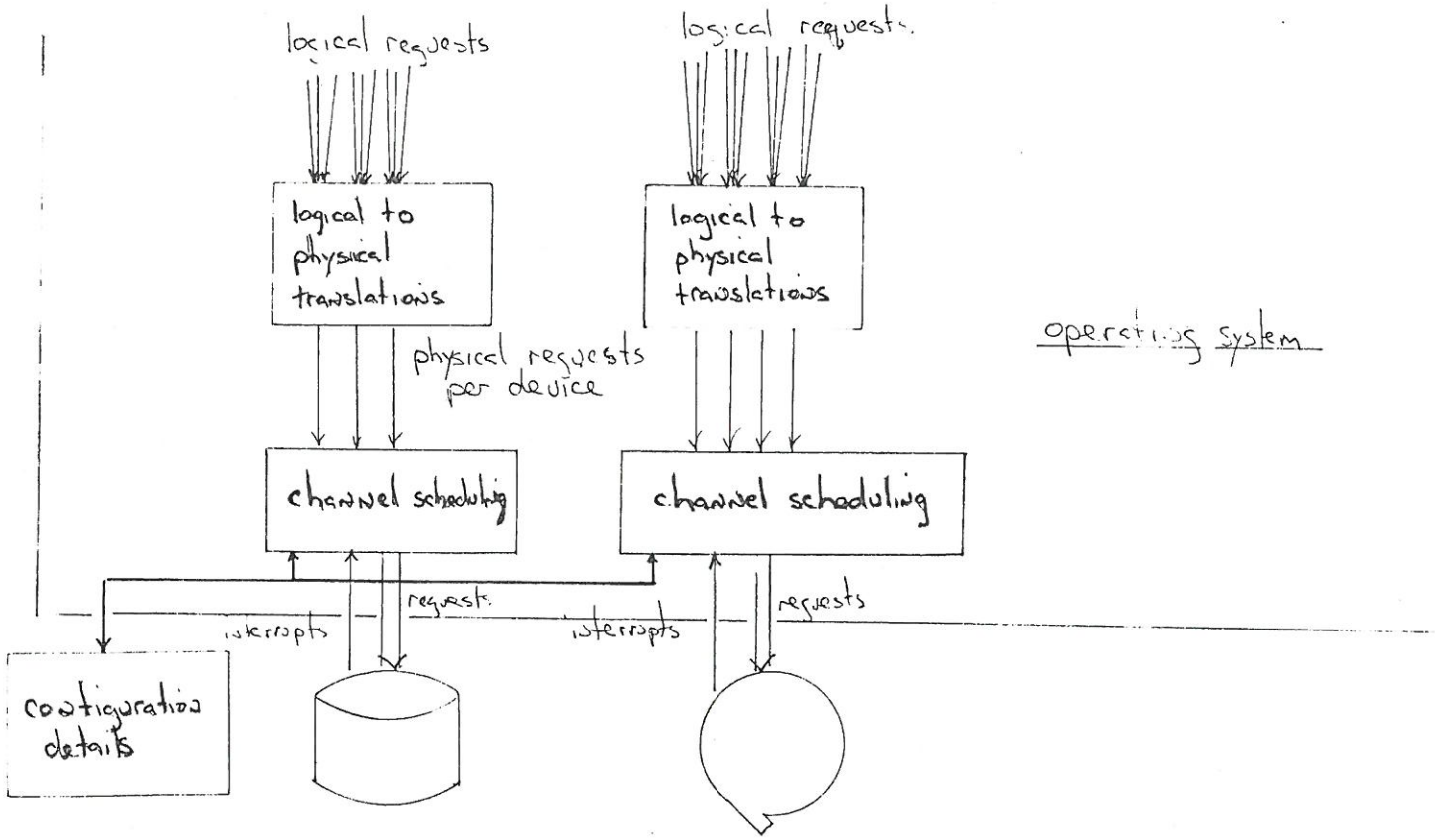


figure 1 b. re-implemented device control modularization.

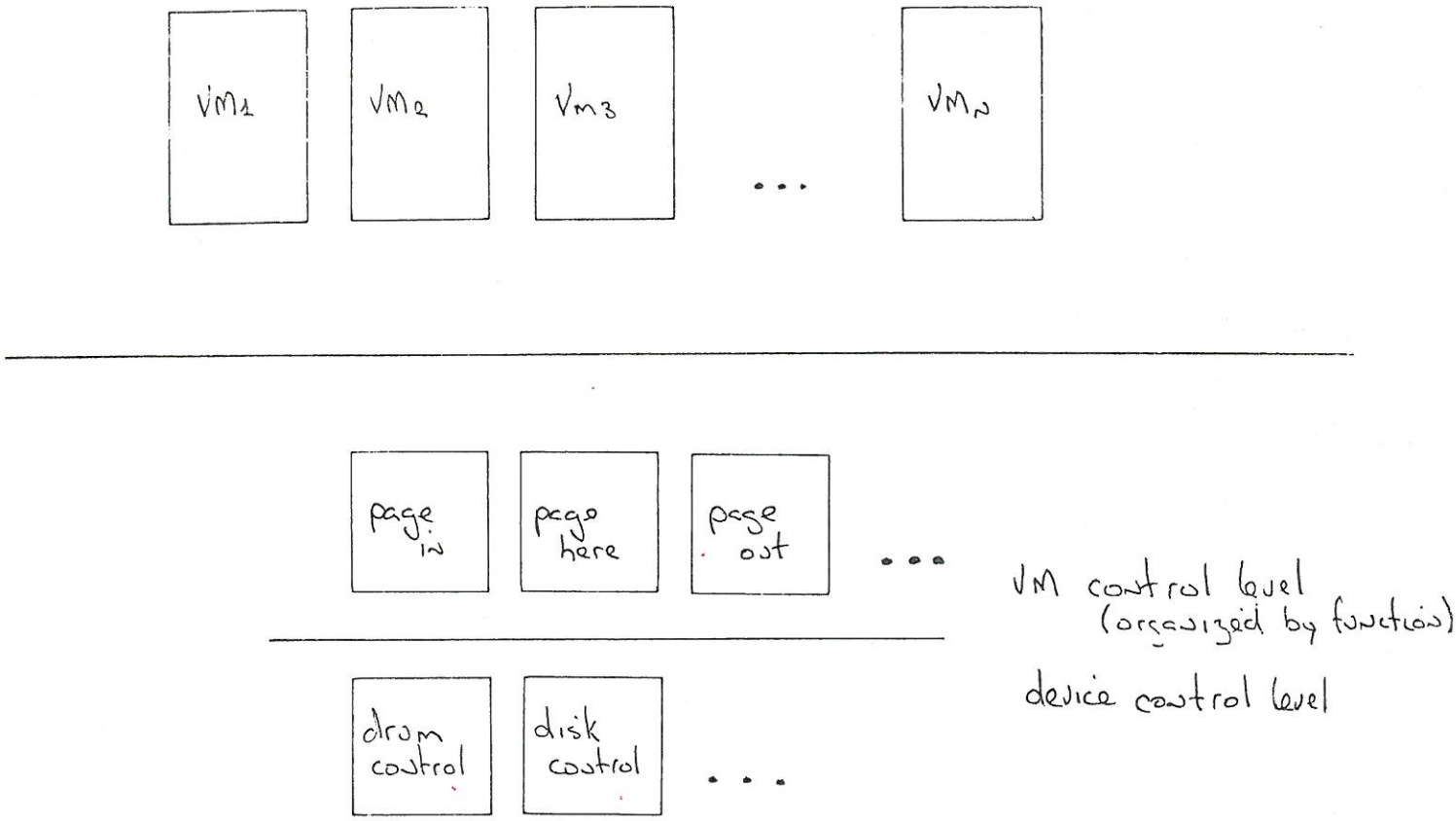


Figure 2 a. (current virtual memory control)

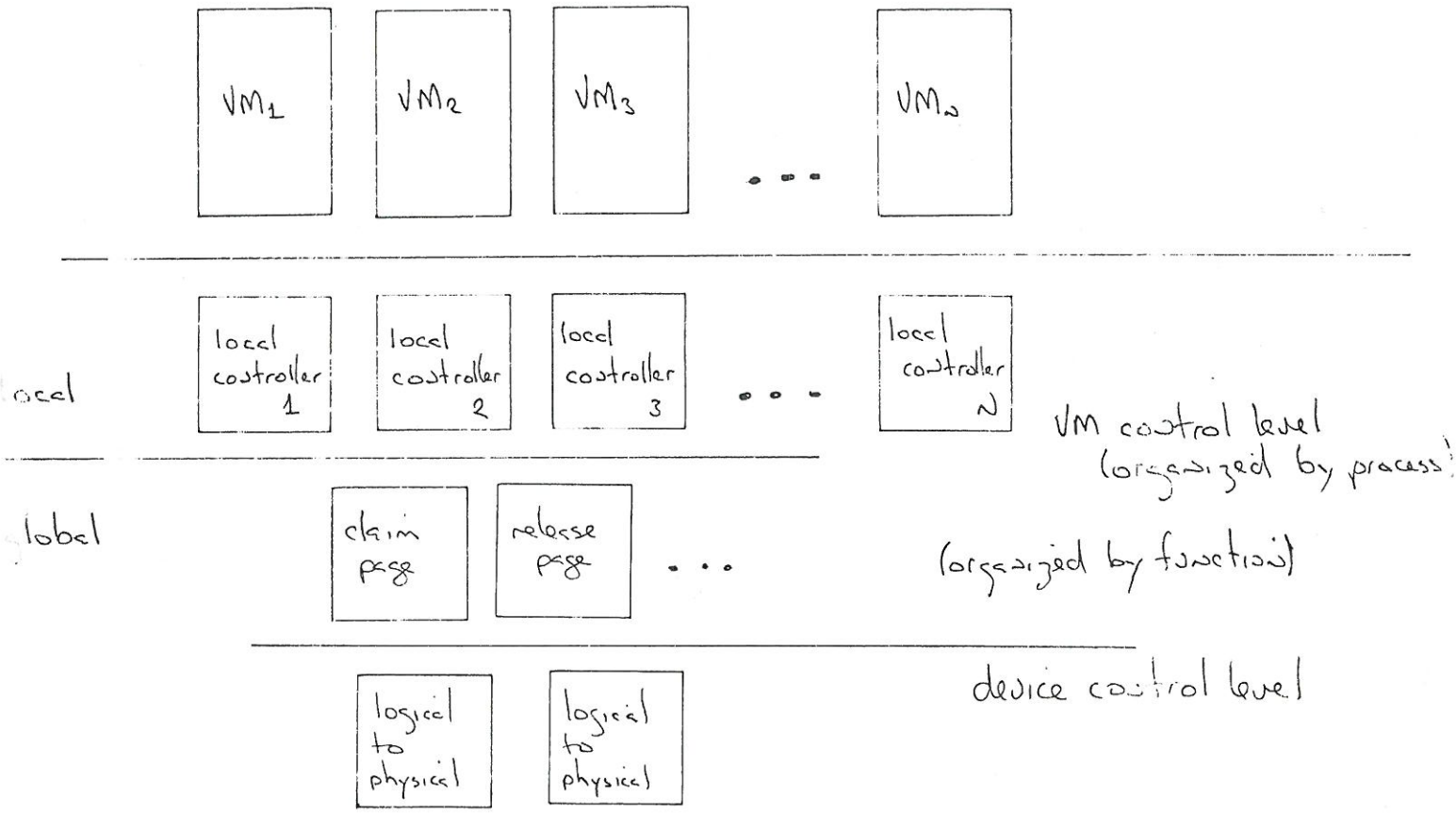
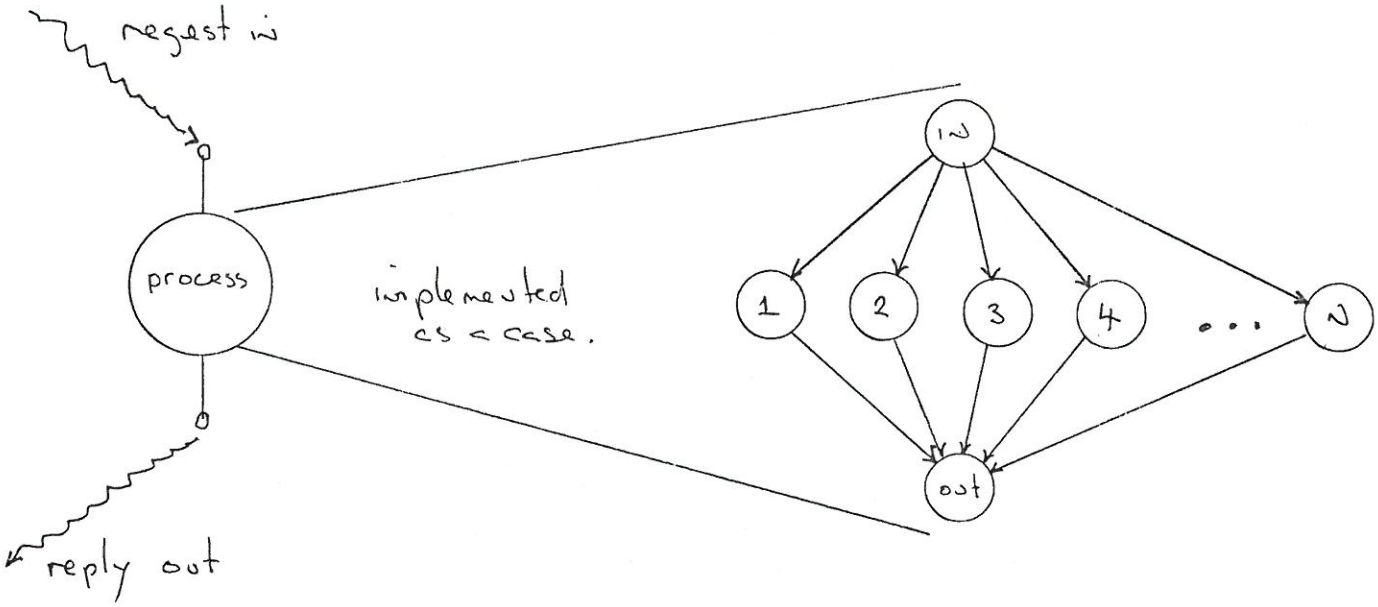


Figure 2 b. (re-implemented virtual memory control)



If we wish to retain context between two arms of the case. We either need to queue all requests while waiting for a reply from the request issued by the first arm of the pair or save the context.

ie.

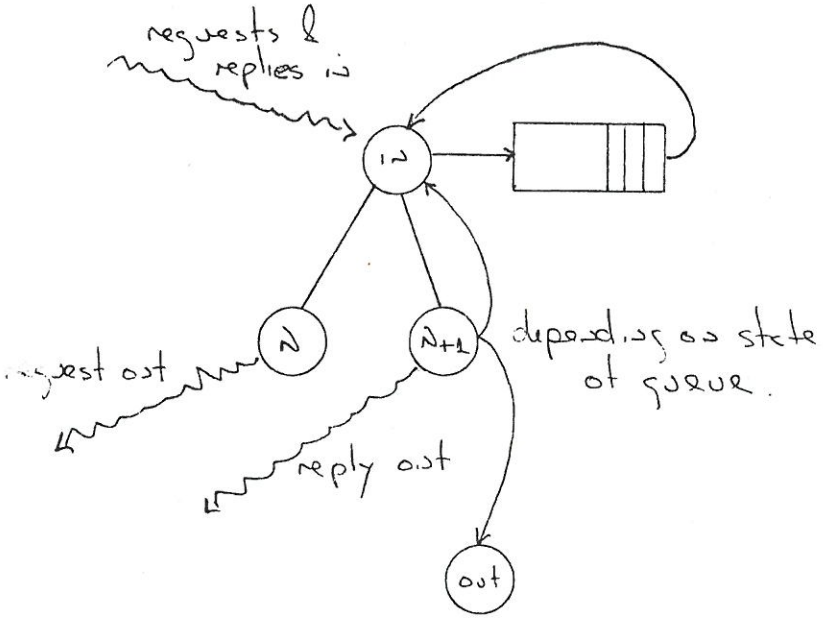
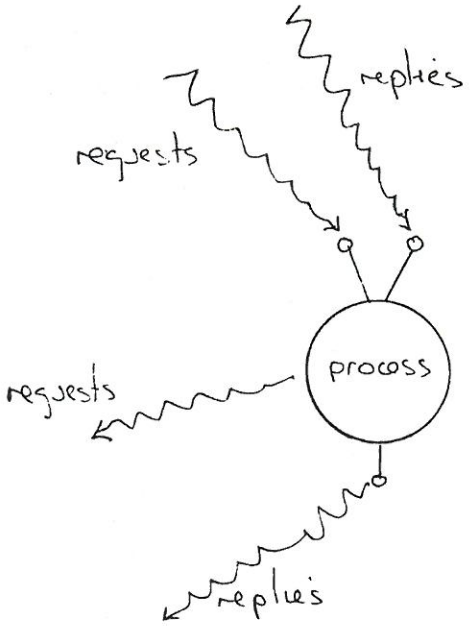


figure 3. (processed with one address for requests & replies).



Now one arm may inhibit requests pending a reply.

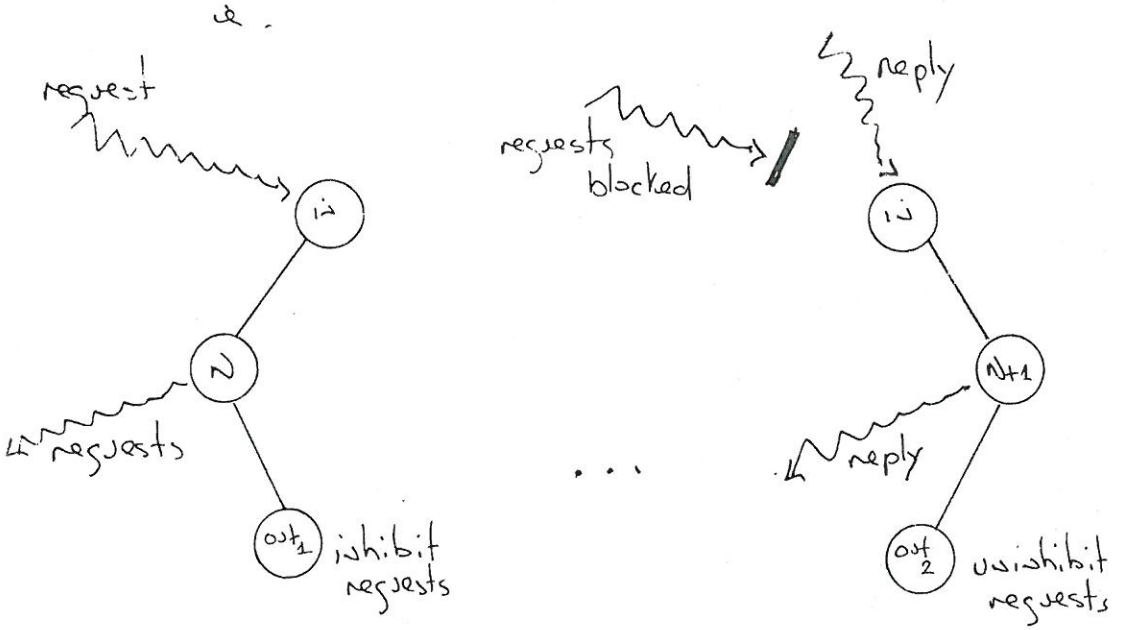


figure 4 (processes with 2 addresses: one for requests & one for replies).

Discussion following Shelness' Talk

Spier Your point about using an existing language for your implementation is an important one. For some reason familiarity with programming languages breeds contempt. If a language is bad, we complain, but if we have used it successfully for long enough without any bother we begin to doubt whether it is really good enough and to want something else. It is ludicrous to talk about *"modern languages"* as if they were automatically better than the old ones.

Shelness I think there is a role for modern languages. But it is best to try them out by rewriting existing systems in them, rather than producing a new language and a new system both at once. This causes nightmares when one is debugging — one doesn't know whether the bug is in the program or the compiler.

Derrett There seems to be a belief in the universities that whenever one writes a new operating system it is necessary to start by designing a new language — or at least producing a new dialect of an existing one. It is time we abandoned this belief.

Flynn What did it cost to reimplement EMAS, compared with the original implementation?

Shelness The reimplementation project has taken 2 years. There was a core of 4 full time and 2 part time people, together with some others on the periphery; so the total is about 12 man years. However the 4 key people were all involved in the previous EMAS implementation, so they were very experienced. It is very difficult to say what the original EMAS project cost, but it was about 100 man years.

Flynn What gain did you get from the reimplementation?

Shelness We felt that there was a pressing need for an alternative to the manufacturer's operating system for the *ICL 2900* range of machines. We wanted to provide the same facilities on the new machines that we had on their predecessors. The desire to do the operating system again and to get it right was secondary. We have extended the lifetime of EMAS, since the old hardware was no longer available.

The new system is much cleaner than the old one. It should be easier to read and maintain.

Flynn Sometimes it is worth rewriting software, and sometimes it is better to throw it away. We need a more realistic way of estimating software costs, effective lifetimes and return on investment.

Shelness I think this is the last time we move EMAS onto a new machine. In the future, systems like EMAS will not be economic — smaller systems will be better.

Flynn Maybe we need another recognised software profession — *the mortician* — who throws away programs which have outlived their usefulness.

Bennett How does the new system compare in performance with the old one?

Shelness The new CPU is about twice as fast as the old one, but we get about the same performance at present. Response time is slightly worse. The elbow of the performance curve is at about 40 users on the old system and at about 32 on the new one.

Derrett Why is the new system not faster than the old one?

Shelness We are still developing it, and making measurements. We hope to improve the performance still. For example under heavy load the old EMAS spent about 40% of its time in the operating system and about 60% running user programs. We have discovered that the new EMAS spends proportionally much more time in the supervisor than the old one — we are investigating this.

Lauesen How big is the new EMAS compared with the old one?

Shelness About the same size. We really need more store on the new machine, especially because the I/O drivers are more complex. For example the code to drive the operator console fills 20K!

Beitz Have you changed the user interface, compared with the old EMAS?

Shelness Very little. The main change is in new partitionings of operating system modules. We understand much better now how to do communication in a virtual memory system.

Derrett What about utility programs — could they be moved across to the new EMAS?

Shelness Yes — most of them. For example we had a complete backup and archiving system right from the beginning.

Lee How was the project managed?

Shelness The project had a manager, but it was mainly democratic. There was no chief programmer team or anything like that.

Shelness A big problem with the original EMAS was discrepancies between the model which the designers had and the model which the implementers had. Programmers tend to inflate their own role — and to produce something too complicated. The team who wrote the new EMAS were more experienced, and have not done this so much.

SESSION D

Tools and Languages

What are the software engineer's tools? What is the difference (if any) between "tool" and "language"? What is the place of standards in a world of tools; can the standard be considered to be a tool? What is the tool's importance? What are the economics of tool making and useage - by what criteria do we justify investment (or lack thereof) in tools? How great is the problem of changing tools, given the resultant possible inability to modify or compile an existing software system?

Considerations for Future Programming Language Standards Activities

John A.N. Lee

Considerations for Future Programming Language Standards Activities

John A. N. Lee
Virginia Polytechnic Institute and State University

This paper reviews the current state of programming language standards activities with respect to the anomalies which exist between the various published and proposed standards for Fortran, Cobol, PL/I, and Basic. Proposals are made for the inclusion of formalisms within future standards and the extension of the standards to include additional items such as error conditions and documentation.

Key Words and Phrases: programming languages, standards, formalisms, formal descriptions, Fortran, Cobol, PL/I, Basic, Vienna Definition Language (VDL)

CR Categories: 2.4, 4.2, 4.29, 4.6, 5.23

Author's address: Language Research Laboratory, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.

Foreword. Dr. Lee has recently been appointed the Vice-Chairman, Development Activities, of the ACM Standards Committee. This paper is most significant in that context, because part of his new charge is to introduce these types of considerations into a program leading to the development of ACM standards. Those projects which would be developed under ACM auspices are those which—from the professional society viewpoint—are not receiving proper attention from international and national standards developing organizations.

Patrick G. Skelly, Chairman
ACM Standards Committee

Introduction

During the past three years, four programming languages have been subjected to public review in preparation to their becoming National Standards. These were: Cobol (X3.23-1974[1]), PL/I (X3.53-1976), Fortran (X3.9-1966, currently being revised as X3J3/76), and Basic (in process of development, BSR X3.360-1977). Each in its own way has contributed to a growing controversy as to the usefulness of standards in general and the uselessness of standards for "dead" languages. The updated Cobol standard was criticized for not maintaining the conformance of programs written in the earlier 1964 standard, whereas the new Fortran proposal has been held up by the lack of "structured" programming elements. PL/I is the first standard which was presented in the form of a totally formal description, though Cobol contained a graphical syntactic specification. This has stirred a controversy related to the intended audience of standards since the lack of a verbal description in the PL/I standard seriously reduced the audience who could respond to

[1] X3 numbers refer to the reference code of the American National Standards on Computers and Information Processing, American National Standards Institute, 1430 Broadway, New York, NY, 10018.

the technical content of the proposed standard during the public review period. Finally Basic added to the overall debate by including in the proposed standard, specifications on error reporting and recovery, as well as certain documentation requirements.

For the first time also, the Fortran and Basic proposal both include specifications for the conformance requirements of programs and implementations. Distilling the conformance requirements from the Fortran and Basic proposals, *a conforming program* is one which is syntactically correct and which does not violate any of the semantic specifications of the standard. Thus a conforming program does not contain any features which are extensions of the standard language and contains only those elements whose constructs are meaningful within the semantics proscribed in the standard. *A conforming implementation* is simply then a processor which accepts and correctly processes a conforming program. By this definition a conforming implementation can contain additional features which are extralingual and still conform to the standard. Such a requirement is perfectly valid when one considers that it is not the purpose of a standard to inhibit language development. In fact, a "good" standard should be considered to be one in which provision is made to extend the language in a conformable manner. By this means, languages and standards can be evolved naturally and revisions can be developed on the basis of tested features.

Reviewing these standards, several questions need to be answered and to be considered to determine the program of work of future programming language standards activities:

1. To whom is a standard directed?
2. a) Should formal descriptions of both syntax and semantics be included in all programming language standards and standards which relate to programming languages (such as the numeric representation standard)?
   b) Should there be a standard formalism developed for use in all programming language standards?
   c) Should a programming language standard include both formal *and* verbal, explanatory descriptions?
3. Does a programming language standard only impact the language processor and programs or are other features of an implementation included, such as documentation?
4. Should any consideration be given to the concept of standardizing language features and then composing standards of these features (suitably clothed in syntax) and special elements?

The answers must be tempered with the basic tenets of standards development. There are three common criticisms of standards—too soon, too late, and who cares? Primarily it must be realized that the purpose of a programming language standards development activity is to produce a standard, *not* to develop a programming language. The materials with which a standardizer works are the *existing* instances of language constructs, the three primary major tasks being: (1) to select the language features to be included, (2) to choose between various alternative "equivalent" features, and (3) to ensure the consistency of the totality of language features.

The solution to the first of these appears in several different forms: the PL/I and Fortran standards choose to define the whole language and to permit future subsetting standards. The Cobol standard presents the language in a complete set of modules which can be combined in a limited number of forms to compose many varieties of an implementation. Initially the proposed Basic standard covers only the minimal elements of the language and future enhancements standards will build on this core. The choice between several forms of a language feature is difficult and invariably prone to partisan pressures for inclusion in the final product. The solution which permits the inclusion of several alternate forms undermines the efficacy of the resulting standard.

The Audience of a Standard

This question has been the subject of considerable debate with respect to the proposed PL/I standard which was reported out of committee X3J1 without any supporting explanatory verbal documentation. Unfortunately much of the discussion which this produced did not differentiate clearly enough between the effects of the standard and the standard itself. For example, at the NCC-1976, Lois Frampton, chairman of X3J1, stated (paraphrased) that ". . . the purpose of the PL/I standard is to ensure that implementations conform to a common set of specifications (. . . and . . .) that users will be protected by the implementers." From this one is forced to conclude that implementations would not permit the user to develop nonconforming programs. However, the definition of conformance as applied to programs and implementations is not as restrictive as to enforce this expectation. In fact, the proposed PL/I American National Standard does not include any statement with regard to conformance. Obviously the ultimate beneficiary of a standard is not the language processor implementer or the programmer user. Instead the beneficiary is the customer for whom the program is written. However, we must again distinguish between the standardized language and the standard. That document which is called a "standard" provides a specification for the language processor implementer to follow and is a guarantee to the programmer that what he writes is universally meaningful.

While a standard does not in itself constitute a programming manual, it must provide the basic syntactic and semantic specifications for those manuals. In

the event of a failure of a specification in the manual, the standard must provide a detailed description of the operation of each language element and an interpretation of the resulting state of the abstract machine. Thus a standard is a part of the essential library of a programmer. However, the primary audience for a standard are the implementer and the procurement agent who must ensure that an implementation conforms to the desired standard.

Formalisms in Programming Language Standards

Experience with the verbal style of programming language standards has revealed that no matter how careful the developers are with their formal style of language, including such terms as to differentiate between "must," "shall," and "is," ambiguities of meaning still arise and require interpretation by the originators. Unfortunately, it is a fundamental principle of law that intention (no matter how well remembered by the authors) cannot be used as the basis for the interpretation of the law. That is, a law must stand on its own feet. Similarly in the domain of standards, the intent of a phrase included in a standard is irrelevant to the interpretation of that phrase at a later time. By coincidence, it may happen that an interpretation conforms to an intent; but that is not a requirement. In the cases of both Fortran and Cobol, their earliest versions [1966 and 1964 respectively] required extensive interpretation in the years between the production of the standard and the subsequent revision [ANSI, 1969 and CODASYL, 1968]. On the other hand, little practical experience has been garnered with respect to the need to interpret the formal specification. However, it is expected that since many of the implementation decisions have to be studied in developing a formal specification, the ambiguities are to be removed. Experience with developing formal specifications from verbal descriptions has also revealed that illogical constructs are readily identified and can be corrected.

There have been four efforts at developing a formal description of a language related entity within an American National Standard: (1) the syntactic specifications within standard Cobol (X3.23-1974), (2) the syntactic specifications within the standard for the representation of numeric values (X3.42-1975), (3) the syntactic and semantic description of PL/I (X3.53-1975), and (4) the syntactic specification of the minimal Basic proposed standard (BSR X3.60-1977). Each of these has been met by the industry with varying degrees of concern, the major response being related to the ability of the regular standard user to understand the formalisms.

Syntax *and* Semantics

There are two levels (at least) of formal specification that must be considered. First there is the syntactic specifications for the language. Immediately we find ourselves in a dilemma. Although the Backus-Naur Form (BNF) has been the common form of specification of syntax in the literature for at least ten years, (1) there exist several different symbolic forms of BNF, (2) BNF and regular expressions are commonly intermixed [see both X3.42-1975 and X3.53-1976], and (3) BNF is actually only applicable to context-free languages. Unfortunately, very few of our programming languages are totally context free and thus a decision has to be made as to the next step to be considered for developing a formal (accurate) description of the language. The PL/I solution to this problem is to specify syntactically a superlanguage, certain elements of which are semantically unacceptable. That is, instances of the language can be developed with respect to the syntactic specifications which are invalid according to the semantic specifications. This approach may be satisfactory on the basis that the specification of the language is to be regarded as a whole and that it is invalid to regard the syntactic specifications as being capable of standing on their own.

There have been several efforts to develop a syntactic specification system which will take into account the context-sensitive requirements, [Ledgard 1974, Lee and Dorocak 1973]. However, these have not been totally successful due to their complexity and lack of readability. A recent survey of such efforts [Marcotty et al. 1976] showed (in the opinion of this author) the complexity which is obtained when syntax and semantics are combined into a single system. It is not expected that these academic efforts will be accepted in the foreseeable future.

The PL/I style of semantic specification which is a variation on the Vienna Definition Language (VDL) [Lucas 1969] is actually only semiformal. That is, there are a number of cases where the complexity of the definition system itself was so overwhelming that the standard reverts to relying on verbal descriptions and "*common sense*" in order to specify the particular feature.

The PL/I descriptive techniques suffer from the fact that the language used, while being mathematical in form, is peculiar to this one standard. VDL depends on knowledge of a conceptual machine which operates over a set of functions (usually visualized as trees) quite differently from any known computer. Furthermore, the set of available instructions in the conceptual machine is miniscule, relying heavily on parameter passing and recursiveness to accomplish the descriptions of the language features. Few programmers will be able to deal with these abstractions. There are two solutions to this problem: (1) create a new means of specification based on a more realistic abstract machine (the "standard" computer?), or (2) provide a verbal description in parallel with the formal description for use by the programmer.

Standardized Formalisms

An examination of the existing and proposed programming language standards shows a considerable difference in organization besides simply the differences in formalisms used.

There are certain advantages to be attained if the set of standard languages are provided with a common base of definition. First, there is then a common means of comparison between the languages, and second, there may be developed at a later date a means for formally describing programs based on the formal description of the language in which the program is written. On the basis of a formal description of a program there exists then the possibility of validating the program through a standard verification procedure.

Computer Science is so young that there is not yet any single means of specification which has emerged as the predominantly superior system. This is partially because most systems [Lucas 1969, Lee 1972, Marcotty, et al. 1976, Strachey 1973, McCarthy 1970, and others] have been applied to few languages and there is little overlap in the objective of each specification. Thus it is difficult to judge the comparative effectiveness of the specifications systems. By sheer numbers of specification applications, BNF and VDL win the race for supremacy. However the appropriateness of VDL must be carefully questioned.

Verbal and Formal Descriptions

In the recent case of the proposed PL/I standard, the X3J1 committee felt that the task of producing both a formal description of the language and a verbal, explanatory document was too onerous. The committee (through its chairman as part of the panel discussion at NCC-1976) pointed out that attempts were made during the early portion of their effort to accomplish the task of developing an explanatory standard which was to be supported by the formal documentation. Conversely, no experience has been garnered in the task of developing the explanatory document based on the formal document of PL/I.

Each time a standard is processed through the American National Standards Committee X3, a different attitude is taken toward the inclusion of explanatory material and footnotes as part of the standard. In some instances there are explicit statements that the footnotes and appendices are not part of the standard. The removal of material from the body of the standard to appendices is then one means of providing explanatory material in the *same* document as the formal part of the standard.

The existence of an explanatory document based on a standard which is itself presented in a formal manner, raises the question as to the status of the explanatory material, as indicated above. One possible solution to this dilemma would be to publish the explanatory document not as a standard, but instead as a Technical Report of the standards committee.

CBEMA (Computer and Business Manufacturer's Association), as the secretariat of American National Standards Committee X3, has already undertaken the production of such reports in connection with the vocabulary (originally issued as a standard, X3.12-1970) and documentation procedures. Alternatively, this supplementary document could be developed and produced commercially by a publishing house, and carry some notation that the contents conform to the standard. This does not necessarily imply that the standards development committee itself must place its stamp of approval on the publication, any more than the committee is expected to validate the implementations which are derived from the standard. Since the publication of the original Cobol standard, there has been produced a number of textbooks which purport to explain the Cobol standard [for example, Murach 1975]. Whether these are accurate manifestations of the standard is not known at this time. However, it might be reasonable to assume that publications of this kind should be subject to the same restrictions of the use of the phrase "conforms to the ANSI standard" as are the implementations themselves.

The advantage of developing a technical report as an addendum to the formal standard is that this secondary document could well be the basis for the development of manuals to be used by the vendors. In the very least this explanatory document should include information relating to all the important features of the language which ought to be included in a vendor's manual. While such a hope is implicit here, the next section considers those extralingual elements of an implementation which should be included in the standardization process.

The inclusion of both a formal and a verbal description of a language in a standard raises the immediate question of which is the definitive portion? If a comparison is made with standards outside the domain of programming languages, and in particular in the field of media (such as magnetic tapes X3.14-1973 and others, disks X3.46-1974, etc.) then it will be seen that such a combination of formalisms and verbal description does already exist. In these cases the formalism is defined to be the standardizing information, irrespective of any ambiguities in the verbal decription. So in the case of programming languages, the verbal description is merely the explanatory material for use by the "standard" programmer and which is subject to interpretation by the formalism when necessary.

Programming Languages as Complete Systems

To this date, the several language standards developed or accepted by the American National Standards Institute have been directed at two elements of the

language system: the language processor and the programs in that language. However, these are not the only elements which are included in the language system which is delivered to a user. At the very least the system contains:

1. the language processor (interpreter, compiler, etc.),
2. the set of conforming programs (by implication),
3. the users manual containing:
   a) the description of the language from the points of view of both the syntactic forms which are acceptable and the semantics of those forms,
   b) the set of directions relating to the operation of the language processor,
   c) the set of error messages which are emitted by the processor and their causes,
   d) the implementation features which are machine dependent, such as the range of numeric representations and the maximum lengths of character strings, and
   e) a listing of nonstandard or extrastandard features,
4. documentation relating to the installation of the language processor and the maintenance of the system,
5. a statement of conformance with the appropriate standard, and
6. a set of test programs [see Hoyt 1976, for example] which either:
   a) validate the conformance of the processor with the standard, or
   b) provide a set of diagnostics related to the operation of the language processor.

To this date, the X3 committee has concerned itself with documentation only from the viewpoint of documenting programs and data elements totally separately from language standards. It is possible and feasible that each standards development committee contain in its program of work the standardization of these essential features of a language processor system. However, there also exists the possibility that a separate general purpose standard be produced which specifies that the additional items listed above must also be provided in order for a system to conform to the standard.

Particularly in the case of user manuals, it is essential that some direction be given to the vendor as to what details must be included. For example, in studies of Basic [Lee et al. 1974, and Isaacs 1973] it was found that the *majority* of user manuals did not include information on the hierarchy of arithmetic operators or the binding time of identifiers in an input statement. Both of these pieces of information are essential to the proper execution of programs and it was only by experiment that the reviewers were able to ascertain the manner of execution of the processors or their resulting programs.

The question of standardizing errors within the standard has been considered only seriously by one standards development committee: X3J2, Basic. Three elements of error specification need study: (1) the feasibility of identifying errors by the processor or by the host system (in the case of a program), (2) the recovery strategies to be followed after an error, and (3) the error messages.

It must be assumed (though as far as can be determined, has never been explicitly stated) that the current language standards define that anything that is not covered by the terms of the standard is "implementation-defined." That is, items which do not conform to the standard can be regarded by the implementer either to be errors or to be language extensions. While it is not the intent of a standard to restrict language development, the standard should include provisions for specifying that certain anomalies must be treated as errors.

These considerations raise two other issues: (a) the problems related to subsetting or supersetting of languages and (b) the questions of distinguishing between those elements which are "errors," those whose semantics are "undecidable," those which will produce "unpredictable" results, those which are "implementation-defined," and those which are "undefined." The question of errors as related to various levels of language implementation is most easily resolved when the original language specification is the superset (as in the case of PL/I). In this case all subsets can be assumed to be proper, and thus those elements which are specified (usually by the implementer) not to be in the subset are to be reported as errors. However, where the initial language specifications are in terms of the minimal subset language (such as in the case of Basic), with all other instances of the language being enhanced versions (supersets), the problem is much more difficult to answer. In this case, it is not known at the time of standardization of the minimal set what language enhancements might be developed at a later stage. Thus it is important to specify errors in such a manner that language extensions are possible. That is, a program which conforms at the lowest level must also conform at the enhanced level and still develop the same result.

Standardized Language Features

Ledgard [1971] showed that there exists a set of semantic features which are common to many languages. These include such elements as: assignment, block structure, functions and procedures, transfer of control, parameter passing, data structures, and input/output which he used as a pedogical aid to the understanding of languages in general. Examination of many

of our languages show this commonality to be true at some high level but in-depth examination of apparently common features reveals only glaring inconsistencies which are the hallmark of particular languages. This is particularly unfortunate since it would be inconceivable at this point in time to go back to revising the existing standards so as to develop a consistency of features. Even more unfortunately, the novice user is unaware of these inconsistencies and expects that (for example) the looping controls in two languages operate in a similar manner.

These latent inconsistencies will preclude the distillation of common features from the existing languages, and may force the consideration of any development of standard language features to such a low level as to only cause further confusion. The concept of a "standard" computer has been raised several times over the years, only to be rejected on the basis of the inability to be totally general (or more correctly, so as not to favor any particular machine architecture), and the restrictiveness of the abstract machine in implementing (easily) certain high-level language features. Even over a restricted domain such as the procedural numeric oriented languages the commonality of features would be politically difficult to justify.

If a formal description of a language can be expressed in terms of standardized features without the formalism as used in PL/I, a standard standardizing language might be created which itself can be formally described, but which in itself is understandable by the vast majority of the industry. Moreover this would give the advantage of expressing a much wider variety of languages (including applications languages) in high-level terms. In this latter case, a standard would then be composed merely of the syntactic specifications of the language and a set of transformations into the "standard language." In view of the current trends within the industry to "standardize" programming in terms of a highly restricted set of control structures (the so-called Bohm-Jacopini constructs), it is even more necessary to consider the construction of a standard base language, into which all other language would be transformed for the purpose of standards specification. Languages include not only symbolic elements and arithmetic operators but also a set of intrinsic functions such as SINE, COSINE, etc. To date, no consideration has been given in any programming language standardization effort to this problem. In fact, there exist commercial multilanguage systems containing several different implementations of certain intrinsic functions presumably because even their "in-house" standards did not include such common function specifications.

It might be expected that a more successful standardization effort can be elided. In this case, it is not the language feature itself which is to be standardized, but rather the realization of the mathematical function in terms of a "package" which is supplied by the vendor. No doubt it would be imprudent to define which approximation is to be utilized in implementing the algorithm which represents the function; however, it would be feasible to consider the permissible variance of the chosen approximation from the mathematically "correct" result. The procedures for testing the conformance of a given package to the standard would be the subject of extensive research prior to the development of a standard.

In the main, the problem of accuracy of implementations is most important close to discontinuities, often points where even the most sophisticated algorithm has problems due to the infiniteness of the host machine. Certain new procedures [Fosdick, 1976] which are being developed for testing programs must be applied to this problem when they are themselves finally shown to be correct, or at least, computable.

Summary and Recommendations

While we must recognize that the state of the art of Computer Science has advanced considerably over the past ten years since the original Fortran standard was promulgated, the step towards totally formal standards documents in the field of programming languages is not acceptable. Moreover, if any consistent effort is to be made to introduce formalisms into future standards documents it must be done in a consistent manner. That is, there needs to be developed as soon as possible a standard for standards which will provide the basis for specifying both the syntax and semantics of programming languages in such a manner as to be both unambiguously explicit and acceptable to the computing community. It is not acceptable that there should be considered to be an elite group within the industry to whom standards are addressed. It should be expected that all those concerned with conformance to standards, from the language processor implementer to the journeyman programmer, should be provided with a means for accomplishing their appointed tasks. To this end it is essential that programming language standards documents contain *both* the formal description of the linguistic elements of the language and a verbal, explanatory portion. The simple existence of a formal description system does not ensure that the specification of any language is any more accurate than the equivalent verbal description, for all its faults. To date, only one nontrivial language descriptor [London, 1972] has been validated by a formal means. As attractive as formal descriptions are to the standardizers, the validity of a standard is only as good as the proven validity of its description system. To this end, it is imperative that the standards development groups be provided with a validation system at the earliest possible time, or if that is not possible a means by which programs written in the language can be validated.

There ought to be a more compelling reason for specifying a language in terms of a formalism than simply to provide a more obtuse specification. At the very least, there must be some beneficial side effects. Besides the ability to validate the specification and possibly the programs written in the specified language, the formal specification should also be the basis for the development of processor audit routines in order to validate any implementation. To date the benefits of a formally specified language have not been clearly delineated; this task is of paramount importance to our future standards activities.

The adequacy of programming language standards in areas other than the software aspects also needs close attention. Initially it must be recognized that a language processor system, as delivered by a vendor, does not simply consist of a machine readable entity. Rather, the system is composed of a number of distinct items including necessary documentation. These additional items also need to be subject to standardization.

This review has purposely (for the sake of length) not included a closer look at the process of standardization itself. The means by which standards in the field programming languages are developed, and in particular the methods for recording the deliberations of the committees, leaves much to be desired, and should be the subject of a separate study.

During the reviewing process to which this paper was subjected, two reviewers disagreed with several of the contentions made by the author with respect to the domain of programming language standards. For example, one reviewer disagreed that standards were capable of specifying error conditions and vehemently stated (judging from the thickness of the pencil marks) that the statement of recovery procedures was *not* to be included. Other disagreements included objections to the notion that a programming language *system* include the necessary documentation which the vendor supplies. The differentiation between the use of the terms "undecidable," "unpredictable," and "undefined" brought cries of anguish from one reviewer who stated that all such elements should be classified as "implementation-defined." This same reviewer also objected to the inclusion of documentation standards and thus would never find out how the implementer had chosen to define the undefined! I agree that we do not want to enter into a process which will overdefine languages to the point where they are stifled; however, there should exist some minimal standards for *all* elements of a language processor.

One final issue remains: how to make National Standards more acceptable to both vendors and users. As industrious as the standards development committees have been, their labors have been somewhat futile as measured by the industrial application of their standards. While it is realized that standards activities in this country are developed by volunteers and conformance to known standards is equally voluntary, the ability of those standards to be accepted as examples of the best of current practice must be seriously questioned.

References

ANSI (Committee X3J3). Clarification of Fortran standards—initial progress. *Comm. ACM 12,* 5 (May 1969), 289–294.

CODASYL, COBOL Journal of Development. 1968 (and later years).

Fosdick, L.D., and Osterweil, L.J. Data flow analysis in software reliability. *Computing Surveys 8,* 3 (Sept. 1976), 305–330.

Hoyt, P.M. The Navy FORTRAN validation system. Dept. of the Navy, Washington. D.C., 1976.

Isaacs, G.L. Interdialect translatability of the BASIC programming language. ACT Tech. Bull., U. of Iowa, Iowa City, 1973.

Ledgard, H.F. Ten mini-languages: a study of topical issues in programming languages. *Computing Surveys 3,* 3 (Sept. 1971), 115–146.

Ledgard, H.F. Production systems: Or can we do better than BNF? *Comm. ACM 17,* 2 (Feb. 1974), 94–102.

Lee, J.A.N. *Computer Semantics.* Van Nostrand-Reinhold, New York, 1972.

Lee, J.A.N., and Dorocak, J. Conditional syntactic specification. Proc. ACM '73, ACM, New York, pp. 101–105.

Lee, J.A.N., Beckhardt, S.R., and Karshmer, A.I. A candidate standard for fundamental BASIC. NBS-GCR 73-17, Nat. Bur. Stand., Washington, D.C., July 1973.

London, R.L. Correctness of a compiler for a LISP subset. Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N. Mex., SIGPLAN Notices (ACM) 7, 1 (Jan. 1972),

Lucas, P., and Walk, K., On the formal description of PL/I. *Annual Review in Automatic Programming, Vol. 6, Pt. 3.* Pergamon Press, Oxford, U.K., 1969.

Marcotty, M., Ledgard, H.F., and Bochmann, G.V. A sampler of formal definitions. *Computing Surveys 8,* 2 (June 1976), 191–276.

McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems,* P. Braffort and D. Hirschberg, Eds., North-Holland Pub. Co., Amsterdam, 1970.

Murach, M. *Standard COBOL.* Science Research Associates, Chicago, 2nd ed., 1975.

Strachey, C. The varieties of programming language. Tech. Memo. PRG-10, Programming Res. Group, Oxford U. Comptng. Lab., Oxford, U.K., March 1973.

Discussion following Lee's Talk

Lee If we consider users, researchers and standardizers as three groups in the computing world then there is very little contact between researchers and standardizers. This is partly because it is difficult to get funding in this area.

Spier What is the goal of standardization? What guides you when you are developing a standard?

Lee Standardization results from the needs of users to move their programs and data-sets from one machine to another.

Spier The purpose of standards is to protect the investment we all have in software. But what I want to move is the semantics of my problem, not the syntax — I want it to perform in exactly the same way on the new machine as it did on the old one. So standards ought to standardize semantics. We cannot do this until we know how to specify semantics.

Lee There are fundamental problems of standardization which are not yet solved. For example moving data — the output from a *COBOL* program cannot be read by another *COBOL* program.

Stoy Do we need standards for computer languages — syntax and semantics — or for computer concepts — what programmers think about?

Lee Both. I would like to see lists of fundamental concepts.

Lee We don't have a language for describing problems yet.

Gram Yes we do — natural language.

Lee Natural language is ambiguous.

Gram So are problems.

Portability

Nielsen The advantage of standards can be seen when one looks at FORTRAN. FORTRAN compilers by now are standardized across most available machines, and that makes FORTRAN programs very portable.

The fact that compilers for other languages are usually so divergent and machine-dependent makes it difficult to move programs.

Lee The inability to be transported lies in the program, not the compiler. If it fails to transport, it is because I did not write it restricting myself to using the universally standard subset of the language that is common to all compilers.

Nielsen I don't believe such a common subset exists, say for *Algol 60* or *PASCAL*. There are *Algol* implementations that require quotes around the keywords and other that don't, etc.

Shelness But these are trivial problems. When you code for portability, you must confine yourself to a restrictive environment and take these problems into consideration.

Grosch There were 7000 machines in the US government when I started in The Bureau of Standards. And we in The Bureau wanted to get a small *COBOL* program to work on several of these. Not all of them, just some. If we had found just *two* machines that could run the same *COBOL* program identically, we could have felt that we had an enourmous advantage. I'm not sure we ever did, although there are rumours that it was done.

Beitz The problem with *COBOL* is that the committee allowed each manufacturer to specify his own external representation. Had there been a uniform external representation, that would have solved 95% of existing *COBOL* portability problems. As it happens, each manufacturer has his own, down to different alphabet codes.

Derrett I think that language portability is a red herring. I have had experiences with moving similar text-processing programs in *FORTRAN* and in *BCPL*. We got the *FORTRAN* program running within 10 minutes but it took us a month to change its output routines to use a different device. The program was *"portable"* but it was incomprehensible. The *BCPL* program took a week to get running, but then we could modify it easily because the program itself was comprehensible. We need ways of expressing semantics in such a way that the meaning of a program remains the same on various machines, and at the same time is comprehensible to users. The syntactic differences between machines are not so important.

Grosch You are not talking about portability — one must assume a certain commonality — the same devices for example.

Conformance

Lee One of the big issues in standardization is *conformance*. There are two types of conformance:

1. Conformance of a program to a standard, i.e. whether the program exceeds the bounds of the language.
2. Conformance of an implementation to a standard, i.e. whether standard conforming programs are correctly executed.

If we look at standards in this way, then development is allowed, but the ground rules are specified.

Spier Any conforming compiler ought to include a special option which rejects any non-conforming program.

Lee Yes. And a standard ought not just to say what is allowed, but also what is definitely not allowed.

Mosses Maybe we should insist that anyone who provides extra features which are not in the standard language, should also provide a way of transforming programs in the extended language into the standard language, so that they can be transported to other machines.

Lee We have considered building an extensibility feature into every language standard — something which says: *"If you are going to extend the language then you must do it this way"*. As an example we could insist that any extension is defined in standard language terms. However, not all extensions can be described in the standard language.

There is also a question of style — insisting that language extensions are in the spirit of the base language. But this is a question of aesthetics.

Grosch I would like to prohibit extra features in compilers, and save them up for a couple of years. Then we could bring out a new, improved version of the language.

Salesmen vie in adding new features to compilers and the result is an upward spiral of incompatibility.

Bennett We need a sort of *ANSII* seal of approval, which says that a particular compiler conforms to the standard.

Lee We can do this, but the manufacturers don't take advantage of it.

Shelness Yes, standards are in the users' interest but not in the manufacturers' interest.

Shelness Why don't we have language checkers, which check whether programs conform to the standard? Such a checker should be written in the standard language.

Grosch There is a *COBOL* validator — written in standard *COBOL*.

Shelness These validation tools ought to be more widely available.

Grosch Yes, validation tools are available within the US government, but I am not sure whether they are available outside.

Lee The validation routines were developed by Grace Hopper for the Navy, and are now administered by the National Bureau of Standards.

Enforcement of Standards

Flynn I think that we encourage our students not to accept standards by not teaching them standard languages from the start. Universities are very prone to using dialects of languages. Somehow, we believe that our dialect is *OK*, and that standards are the product of an enemy bureaucracy.

Lee We teach our students *BASIC*. They get 3 documents — a textbook, with its own description of *BASIC*, the manufacturer's *BASIC* manual and the *BASIC* standard. They can use all three for learning and reference but we require them to use the standard when they write programs.

Grosch We need 2 classes of education in computing. In the universities the necessity for original thinking and the desire to try new things is dominant and necessary. But industry needs a basic pool of beginning programmers who don't need to be graduates. These people need simple instruction with standard languages.

We shouldn't apply university computer science perspectives when training these people.

Beitz If people don't want to use standards then we should not force them to. It is not our job to force people to write standardized programs if they don't want to.

Lee But someone has to say what the standard is, so that people who want to conform to it can do so.

For example there is an engineers' code of practice for building bridges. An engineer is not forced

to follow the code of practice but if he does then he can be confident that his bridge will not fall down. We need similar codes of practice in our profession.

Nielsen We find that our users innocently use all of the featues of the programming languages available, without being aware that some of these features are nonstandard. They write huge programs and then find they can't move them.

Lee Yes, and manufacturers' corporate standards are not the same as international standards.

Lee Conformance to standards depends on having good standards. If we have good standards they will enforce themselves.

Lee It is important to remember that it is not *"THEM"* that do things but us. We are all responsible for standards.

A Comparison of Description Tools

Chr. Gram

INSTITUTTET FOR DATATEKNIK
Bygning 344 og 343
Danmarks tekniske Højskole
2800 Lyngby Tlf. (02) 88 15 66

DEPARTMENT OF COMPUTER SCIENCE
Building 344 and 343
Technical University of Denmark
DK-2800 Lyngby, Denmark

A COMPARISON OF DESCRIPTION TOOLS

- an amateur's view of formal semantics.

Chr. Gram

May 1978

1. COMMAND LANGUAGES

The international IFIP Working Group 2.7 on "Operating Systems Interfaces" and some national groups are studying the problems of defining high-level, machine-independent Command Languages (CL-s). The existing CL-s differ greatly, not only in the syntactical forms but also in the semantics and in the underlying philosophies and concepts. Hence there is a need for ways of specifying the objects and functions in CL such that users and implementors from many circles get the same clear and concise picture of a CL.

Some CL objects, e.g. simple variables, are like the objects of a programming language (PL), but other types are quite different from what is met in PL-s. A CL must be able to manage resources like cpu-time, working storage, auxiliary storage, permanent files, access rights and protection, etc.

As an example of differences in existing CL-s let us consider the file handling. Nearly all CL-s recognizes - under different disguises - three types of files: Permanent files with a 'long' lifetime, accessible from several jobs through some sort of file catalog; temporary files private to one job; and embedded files, i.e. files embedded in the job stream and accessible only from this job. But yet they are handled very different from CL to CL.

IN IBM's OS/370 JCL the catalog of permanent files is tree-structured, but the hierarchy is not used consistently to obtain protection; in many other CL-s the catalog is a simple, 'flat' list structure. CDC's Kronos recognizes two types of permanent files (called Direct Access Files and Indirect Access Files, reflecting a difference in the way they are stored) with two sets of access procedures in the command language.

Temporary files: The command language SCL for ICL2900 has algol-like blocks and the scope of a temporary file is the block where it occurs; in Burroughs' WFL the scope is the actual task (= job step); and in CDC's Kronos the scope is the whole job. In OS/370 JCL the scope may be a job step or the entire job.

Embedded files: In Kronos an embedded file is allowed only at the end of the job stream, while in other CL-s it must be placed in the job step where it is used (and it is local to this job step).

Attributes and access methods: In OS/370 JCL the user may - and in many cases has to - specify a large number of attributes concerning logical and physical storage properties and access methods, but they are not used at the CL-level. In Kronos a file is considered as a sequence of records (numbered 1,2,...) and a file may be positioned to any 'begin-of-record'-mark. In WFL the permanent files may be rewound, but not the temporary files.

CL manuals: Each CL manual uses its own terminology and concepts, and the underlying operating systems are to some extent based on different philosophies. A very simple-minded illustration of this is, that a permanent file is called catalogued by IBM, global by Burroughs, permanent by ICL, and indirect access permanent or direct access permanent by CDC.

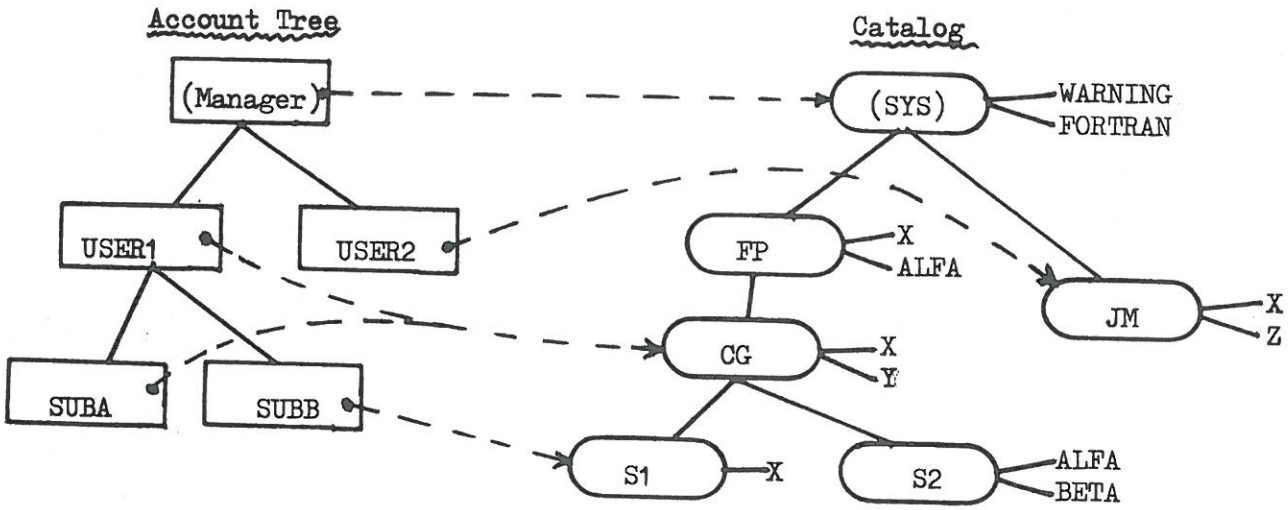
2. DESCRIPTION OF A FILE CATALOG

An important object of any CL is the catalog of permanent variables (files as well as other types of variables). We will try different ways of describing the catalog and the CL-functions working on the catalog. As the model we take the catalog structure in CCL, a simple, high-level, machine-independent CL designed and implemented by Johs. Madsen [Madsen 78].

The following sections sketch how the same catalog structure might be described using natural language and using more formal semantic tools.

2.1 Natural Language

Level 1: Description of objects: The accounts form a tree with the system manager's 'account' as the root, 'owning' all resources and all the other accounts. Every account contains the identification and resource attributes



necessary for the administration and book-keeping. The account also has a scope identifier (dotted line in the figure) pointing to a node in the catalog of permanent variables. The scope identifier defines the access rights to permanent variables of jobs run under this account, as explained below.

The permanent variables (files, integers, booleans, and strings) are organized in another tree, the catalog. The variables - or rather references to the variables - are leaves in this tree and may be attached to nodes at any level of the tree. The rules of access and protection of variables are:

A job may create, delete, read, and write permanent variables belonging to the subtree designated by the scope identifier of the job's account. The job may read permanent variables belonging to all predecessor nodes of the node designated by this scope identifier. All other variables are inaccessible (unless made PUBLIC by their 'creator'). Example: A job run under account USER1 may read and write the variables

CG.X	S1.X	S2.ALFA
CG.Y		S2.BETA

and it may read the variables

FP.X	SYS.WARNING
FP.ALFA	SYS.FORTRAN

Names of permanent variables may be qualified with as many of the predecessor node identifiers as wanted (in order to ensure unique identification). Thus

Y CG.Y FP.CG.Y SYS.FP.CG.Y

all denote the same variable, while X denotes different variables depending on the scope identifier of the job where it is referred to.

Level 2: Short description of functions: A number of operators and functions in CCL deal with the catalog (and the account tree):

CREATE_ACCOUNT_AND_NODE	create a new account and (may be) a node in the catalog.
DELETE_ACCOUNT	delete an account (and may be subtree in the catalog)
CHANGE_ACCOUNT	change some attributes in an account.
EXTERNAL id := expr	create a new permanent variable and assign a value.
EXTERNAL id	make a local variable permanent.
REMOVE id	make a permanent variable local to the job.
PUBLIC node-id EXTERNAL id	make a permanent variable 'more visible' to other jobs by 'lifting' it to some predecessor node in the catalog.
PERMIT id	allow writing for all that may read the variable.
PROTECT id	cancel the effect of PERMIT.
RESERVE id	current job gets exclusive access to the variable until the end of current CCL block.
id := expr	assign value to a variable.
id occurring in expr	get value of a variable.
END	reset the 'exclusive access' status at end of CCL block.

Many of these explicit functions are best defined by introducing some more basic, implicit functions:

SEARCH id	search a variable in the catalog. The node where the search begins is given through the scope identifier of the current job.
CREATE id	insert a new leaf with a variable in the catalog.
DELETE id	remove a variable or a node and its variables from the catalog.
CHANGE_STATUS id	change the protection status of a variable.
. . .	

Level 3: Detailed function description: Obviously the above list of functions does not specify the semantics precisely. Therefore a more detailed description is needed, and as an example we shall give the specification of SEARCH. It is a key function, being used in many of the other functions, and it is one of the more complex functions.

The following notations are used:

A simple name is just one identifier (and should be the name of a variable, i.e., a leaf of the catalog tree).

A qualified name is a list of identifiers, the last of which is a leaf name while the others are names of (one or more) of the predecessor nodes of the leaf. Thus Y is a simple name, and CG.Y , FP.CG.Y , and SYS.FP.CG.Y are the qualified names of the same variable.

The actual node is the catalog node referred to by the scope identifier of the current job account.

The actual scope is the actual node and its predecessors.

Given a name, the search function must distinguish between the name being simple and qualified:

SEARCH simple name: (1) Is it a variable at the actual node?

Yes: Success.
No : Proceed with (2).

(2) Repeat (1) upwards with actual node = father of last used node. If failure within actual scope: Not found.

SEARCH qualified name:

(1) Upwards: Starting at the actual node and working upwards through its predecessors, it is checked whether name = a qualified name of one of the variables.
If this fails for all nodes in the actual scope, the search proceeds downwards.

(2) Downwards: Starting at the immediate successors of the actual node and working down level by level in the subtree of this node, it is checked whether name = a qualified name of one of the variables.
If this fails for all nodes below the actual node, the search proceeds sidewards.

(3) Sidewards: If some variables (not belonging to the actual scope) have been made PUBLIC with a node identifier in the actual scope, the search proceeds upwards as under (1):
For each predecessor of the actual node it is checked whether name = a qualified name of one of the variables made public to this node.
If this fails for all nodes in the actual scope, the total search has failed.

If the other functions were described in a similar way, this would complete the definition of the objects catalog and account tree.

2.2 Axiomatic description

Level 1: Domains of objects: The domains needed for defining the catalog and the account tree may be

```
Catalog   =  TREE( Node )
Node      =  Node_id (Simple_id  Value)*
Id        =  Node_id | Simple_id | Node_id  Simple_id
Value     =  Intg | Bool | String | File_ref
Accounts  =  TREE ( Acc_id  Resources  Node_id )
Resources =  - - -   (not specified here)
```

Level 2: Types of functions: The functions are the same as above, but here they are defined by giving the domain and the range for every function:

```
CREATE_ACCOUNT_AND_NODE  ( Acc_id, Resources, Node_id, Catalog, Accounts )
                         → ( Catalog, Accounts )
DELETE_ACCOUNT  ( Acc_id, Catalog, Accounts ) → (Catalog, Accounts )
- - -
SEARCH          ( Id, Node_id, Catalog ) → Value
CREATE          ( Id, Value, Node_id, Catalog ) → Catalog
DELETE          ( Id, Node_id, Catalog ) → Catalog
- - -
```

Level 3: Set of axioms: The semantics of the functions are defined through a set of axioms that tie the functions together in predicates, e.g.:

```
for all  c ∈ Catalog
         n ∈ the set of Node_id in c
         id∈ Simple_id
         v ∈ Value
let
    DELETE ( id, n, CREATE(id,v,n,c) ) = c
    SEARCH ( id, n, CREATE(id,v,n,c) ) = v
    SEARCH ( n.id , n , c ) = SEARCH ( id , n , c )
    - - -
```

But to find a complete and consistent set of axioms is a major task, and the resulting description is unsatisfactory in the sense that all the functions are defined implicitly through their interrelationship with the other functions, and not through what they do 'on their own'.

2.3 Denotational Description

The following description of the CCL catalog is developped by D. Bjørner (private communication) and slightly adapted by me. The notation used is very close to the high level language Meta-IV [Bjørner 76, 77].

Level 1: Domains of objects: The semantic domains may be defined functionally:

```
Catalog   ::  Subtree  WS
Subtree   =   Simple_id  -m->  Catalog
WS        =   Simple_id  -m->  QVal
QVal      ::  Qual_id  Value
Env       =   Simple_id  -m->  Value
Value     =   Intg | Bool | Quot* | File_ref
- - -
```

Level 2: Types of functions: The functions are defined by specifying their domains and ranges, which may be functions:

```
GET_ENTRY  :  Qual_id → (Catalog → Catalog)
SEARCH     :  Qual_id  Id → ((UP|DOWN) → (Catalog → Value))
CREATE     :  Qual_id  Id  Value → (Qual_id → (Catalog → Catalog))
DELETE     :  Qual_id  Id → (Catalog → Catalog)
- - -
```

Level 3: Function definitions: The semantics of the functions are described through functional, static expressions and statements. As an example the definition of SEARCH might be (GET_ENTRY is an auxiliary function):

```
GET_ENTRY(node)(catalog) =
   if node = <>
      then catalog
      else (let mk-Catalog(subtree,ws) = catalog;
            if h node ∈ dom subtree
               then GET_ENTRY(t node)(subtree(h node))
               else error )
```

```
SEARCH(id,node)(dir)(catalog) =
   (let mk-Catalog(subtree,ws) = GET_ENTRY(node)(catalog);
    if is-Simple_id(id)
       then if id ∈ dom ws then ws(id)
                           else if l id > 1
                                   then SEARCH(f node,id)(UP)(catalog)
                                   else error
       else (trap exit with:
                if dir=DOWN then exit
                            else if node = <>
                                    then error
                                    else SEARCH(f node,id)(UP)(catalog);
             if l id = 1
                then if h id ∈ dom ws
                        then ws(h id)
                        else exit
                else if h id ∉ dom subtree
                        then exit
                        else SEARCH(node ⌢ <h id>, t id)(DOWN)(catalog)))
```

A similar denotational semantic description of the catalog in IBM's OS/370 is found in [Madsen 77] .

2.4 Computational Function Definition

The functions may be defined constructively by writing them down in a high level algorithmic language, say Algol68. But - as experienced by the English BCS Working Party on Job Control Language - this leads too quickly into implementation details. It seems difficult, if not impossible, to write a complete algorithmic function definition without de facto making some decisions and choices concerning the implementation, which should be of no concern at all at this level of abstraction.

3. CONCLUSION

While developping ideas and concepts a very careful balance must be kept between formal and informal descriptions, especially when exchanging ideas between groups with different background.

The advantages of an axiomatic or an denotational approach is that it allows you formally to prove and deduct certain properties of the objects described. On the other hand, the formalism may conceal some simple, fundamental properties that are easily expressed in natural language.

At present I personally favour a description consisting of

- the semantic domains, i.e., the objectsunder consideration described in some set-theoretical way.
- the function types on these objects, i.e., definitions of domains and ranges of the basic functions.
- the semantics of the functions, described mainly in natural language, but using figures, tables, and formulae whenever convenient.

REFERENCES

[Bjørner 76]: D. Bjørner: "META-IV: A Formal Meta Language for Abstract Software Specification", Dept. of Comp. Sc., Techn. Univ. of Denmark, Res. Report ID670 (Nov. 1976).

[Bjørner 77]: D. Bjørner: "Programming Languages: Formal Development of Compilers & Interpreters", Proc. European ACM Int. Comp. Symp., North-Holland (Amsterdam 1977), 1-26.

[Madsen 77] : Johs. Madsen: "An Experiment in Formal Definition of Operating System Facilities", Inform. Proc. Letters 6 no. 6 (Dec. 1977) 187-189.

[Madsen 78] :Johs. Madsen: "CCL - A High-level Command Language; Design and Implementation", Dept. of Comp. Sc., Techn. Univ. of Denmark, Res. Report ID 754 (Jan. 1978) (Submitted for publication in Software).

Discussion following Gram's Talk

Stoy It seems to me that at present formalising any computer language — whether Algol 60 or JCL is a *tour de force*, requiring enormous numbers of axioms and so on. Maybe this is because the languages are too complex and they force us to use natural language and handwaving.

Maybe when our languages are simple enough, the formalisation will capture nicely and concisely what is there.

Gram But think of formalising a dirty business administration system, the problems are much worse.

Stoy At least if the tools are simple it will be clear that the dirt lies in the business administration system and not in the underlying computer system. A good formalism should focus our attention on the real dirt.

A Language for System Description

Ib Holm Sørensen

A LANGUAGE FOR SYSTEM DESCRIPTION

by

Ib Holm Sørensen

Abstract

In this paper I present a language which can be used to describe computer systems implemented in a mixture of hardware, firmware and software. Only a brief introduction is given, and the reader requiring further information is referred to Sørensen [1].

The reason for developing the notation presented in this paper was the need for a modelling tool, which could be used to design, analyse and describe a Disk Channel Controller for the experimental Computer System of Aarhus University [2].

The Disk Controller had to provide various degrees of service sophistication (i. e. it was not a fixed-purpose device) as well as allow for future functional evolution (being an experimental device). The main logic of the device was to be implemented in a combination of firmware and software to provide this flexibility. Specialized tasks, which could relieve software and firmware routines of the monotonous and time consuming jobs of monitoring status flags and transferring single words between disk and main memory could be implemented in hardware.

In the actual implementation some system components would be implemented in software, communicating through variables (store cells); some in firmware, communicating through registers; and others in hardware, communicating through wires.

To document the design I needed a notation which could be applied at all levels of implementation - software, firmware and hardware. Once the system was designed, the descriptions would be used as implementation schemes. Therefore the notation used had to be comprehensible to both hardware and software engineers.

This paper describes the notation which I developed. The notation has been a useful tool for describing the complete Disk Channel Controller.

A system such as the disk controller may be regarded as a set of "processes" which communicate with each other via "lines" (physical lines, storage cells, registers etc.) but which can otherwise proceed independently of one another. Each process is a sequential computation.

The description of the system makes use of two notations:

- A language, used for a detailed description of each process's internal operations.

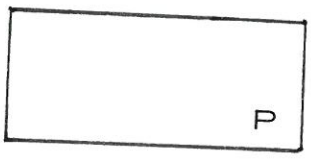
- A pictorial representation, used to describe inter-process communication.

For the description of mechanisms local to a process, any programming language will do. I use an extended version of the high level language BCPL [3]. The extensions to BCPL are very limited and are introduced for the purpose of describing shared components and inter-process communication. For further details the reader is referred to Sørensen [1].

To illustrate a system composed of independent cooperating processes a pictorial notation seems to be the most appropriate. The notation I use lies somewhere between hardware wiring diagrams and flow-charts. It pictures processes and describes interaction between them.

The notation is new, and will undoubtedly require further study and refinement. This short paper is intended as an introductory overview.

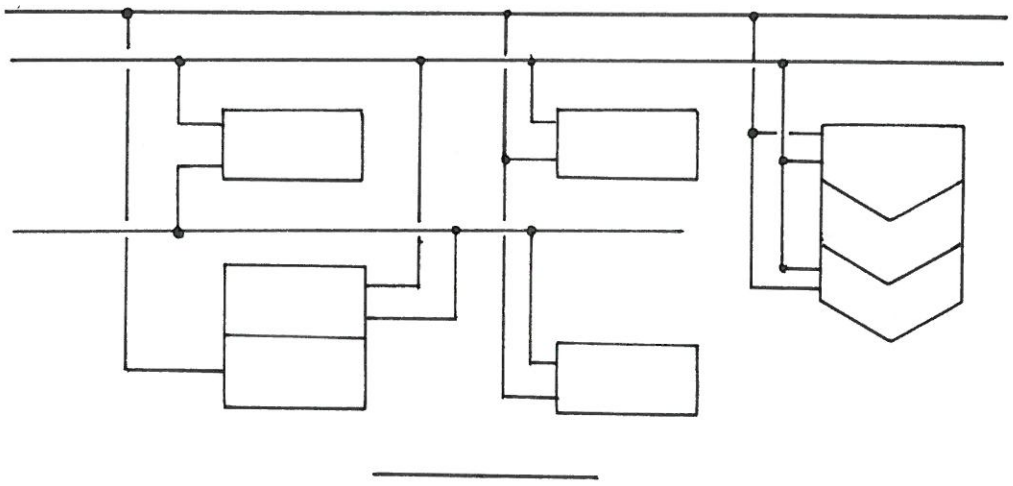
A sequential process whose local activity is independent of all other processes is represented as a simple box,



A collection of sequential processes whose activities are interdependent can be represented by a compound box. A compound box is of either of two shapes,

The pictured processes are connected through shared components such as storage cells, registers or communication wires - in general, any resources which can be accessed by more than one process. Shared components are represented as lines.

Using these elements we can draw a picture showing all the interconnections between processes,



An important aspect of the behaviour of a system is whether and when a process interacts with other processes. Processes interact in two ways[4]

- A process is influenced by another process by reading the value of a common line.

- A process influences another process by assigning a value to a common line.

The basic element which can be described in our notation is a "simple process" which is represented by a simple box.

The activity of a simple process is as follows,

1. Prologue - The process is activated upon some state of its input lines. Simultaneously with its activation it may sample, and possibly write to, its lines.

2. Body - Its "body" is executed independently of all other processes.

3. Epilogue - It writes to its output lines and terminates its activity (until it is activated again).

Only the prologue and epilogue involve interactions with other processes. The body is completely independent of other processes and may, if desired, be executed in parallel with them.

An "interaction statement" (IS), placed in the box adjacent to a line, specifies the process's activity with regards to that line.

An IS is an expression of the form

<prologue> : <epilogue>

where

: denotes the activities local to the processes (body). Local mechanisms are of no interest when the larger system is considered.

<prologue> and
<epilogue> denote the interactions with other processes - reading from and writing to lines - respectivily preceding and succeding the local activities.

Six kinds of basic interactions can be described by means of the interaction statement:

- Conditional Influence - IS form <C-list>:←<E>

  Given a certain value of the line, the process changes the line's value. The process may use the value in its local activity.

- Conditional Inspection - IS form <C-list>:

  Given a certain value of the line, the process's behaviour is affected.

- Influence - IS form :←<E>

  The process changes the value of the line.

- Inspection - IS form :

  The process's behaviour is affected by the value of the line.

- Singular Conditional Inspection - IS form <C-list>←<E>:

  Given a certain value of the line the process changes that value. Both test and value change happen as a single indivisible act. The line's original value is available locally to the process.

- Singular Conditional Influence - IS form <C-list>←<E1>:←<E2>

  Given a certain value of the line the process changes that value. Both the test and value change happen as a single indivisible act. The line's original value is available locally to the process. The process changes the line's values once more when it terminates.

Some of the processes which we want to describe are not simple processes - they require more complex interactions with their environments. We call such processes "compound processes" and represent them as compound boxes. Any such process can also be described as a collection of simple processes.

Composed boxes are provided for sets of sequential processes which are unable to execute in parallel and where the mechanism which administers the flow of control between these processes is self-contained, providing a predictable flow of control.

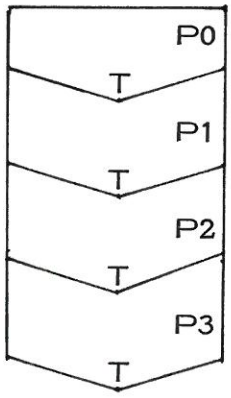
The use of compound boxes in a pictorial representation is often advantageous, as it minimizes and clarifies the pictorial schema of the system.

Four types of compound boxes can be used to describe compound processes.

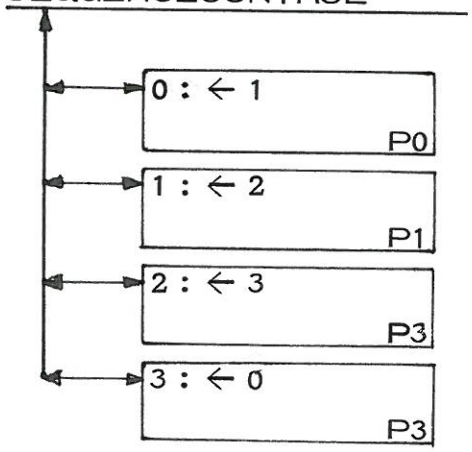
- Sequence - The processes perform their activity in sequence.

  Figure 1(a) illustrates a sequence construct. Figure 1(b) defines its behaviour in terms of a more explicit diagram. Figure 1(c) shows the same behaviour in terms of a program example.

a) Sequence



b)



c) P0 : Wait for Conditions$_0$
Actions$_0$

P1 : Wait for Conditions$_1$
Actions$_1$

P2 : Wait for Conditions$_2$
Actions$_2$

P3 : Wait for Conditions$_3$
Actions$_3$

Goto P0

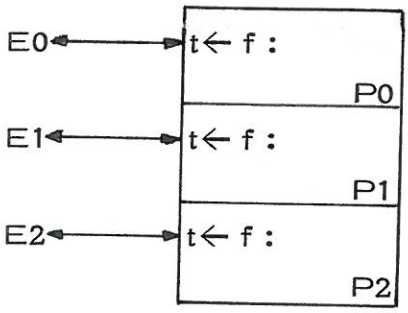
Fig. 1 Sequence Control

- Priority - The processes are controlled by a priority mechanism.

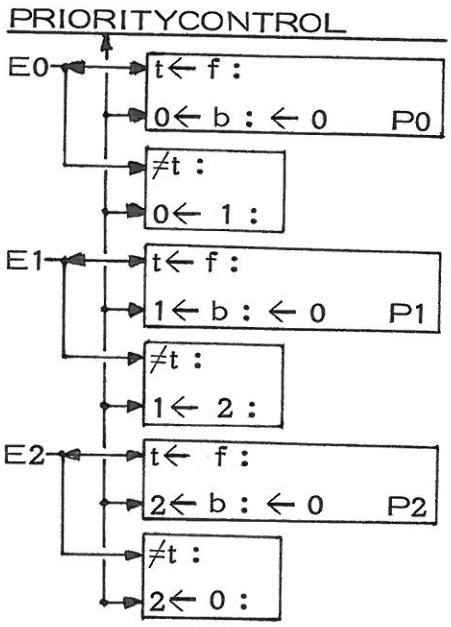
  If two are capable of simultaneous activity, the higher priority one will be allowed to proceed.

  Figure 2(a) illustrates a priority construct. Figure 2(b) defines its behaviour in terms of a more explicit diagram.
  Figure 2(c) shows the same behaviour in terms of a program example.

a) Priority



b)



c)

P0 : if Conditions$_0$ {Actions$_0$
goto P0 }

P1 : if Conditions$_1$ {Actions$_1$
goto P1 }

P2 : if Conditions$_2$ {Actions$_2$
goto P0 }

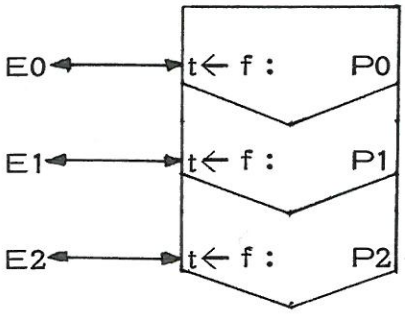
goto P0

Fig. 2 Priority Control

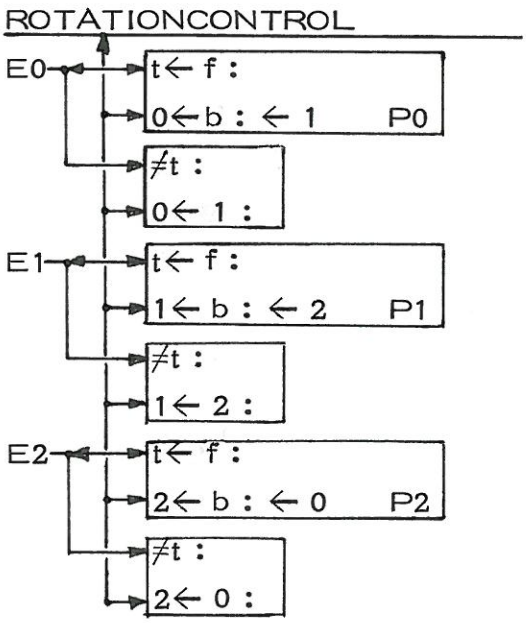
- Rotation – Processes are given a chance to execute, on a strict round robin basis.

  Figure 3(a) illustrates a rotation construct. Figure 3(b) defines its behaviour in terms of a more explicit diagram. Figure 3(c) shows the same behaviour in terms of a program example.

a) Rotation



b)



c) P0 : if $Conditions_0$ do $Actions_0$

P1 : if $Conditions_1$ do $Actions_1$

P2 : if $Conditions_2$ do $Actions_2$

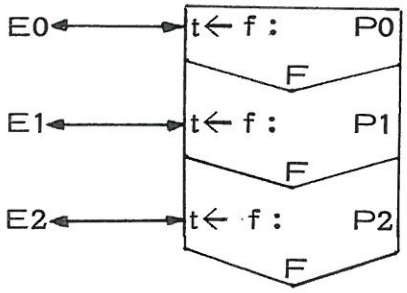
goto P0

Fig. 3 Rotation Control

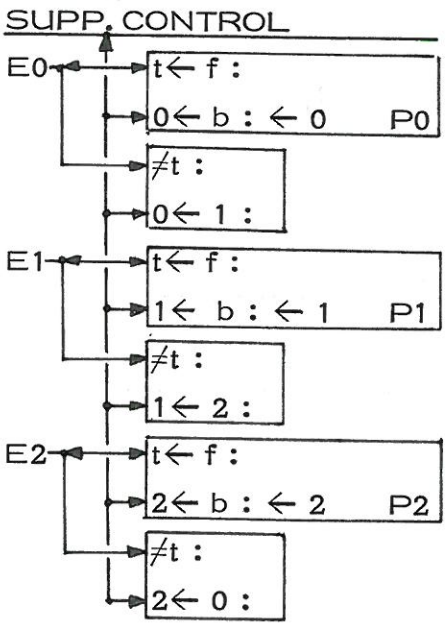
- Suppression - As long as the executing process's <prologue> conditions hold, it repeatedly activates, i. e. run to completion. The process is in what can be thought of as a while loop.

  Figure 4(a) illustrates a suppression construct. Figure 4(b) defines its behaviour in terms of a more explicit diagram. Figure 4(c) shows the same behaviour in terms of a program example.

a) Suppression



b)



c) P0 : While $Conditions_0$ do $Actions_0$

P1 : While $Conditions_1$ do $Actions_1$

P2 : While $Conditions_2$ do $Actions_2$

goto P0

Fig. 4 Suppression Control

Example

Consider an asynchronous pipeline system. Such a system can be regarded as a set of independent sequential processes, each performing certain operations at a specific stage of the pipeline. All these processes should be able to proceed independently and should only require synchronisation (i.e. access to sharable resources) upon activation and termination. Upon activation, a process at some pipeline stage n gets input produced by the process at stage n-1; upon termination, it sends its output to the process at stage n+1.

Figure 5 represents a pipeline which consists of

a) 3 independent processes P0, P1 and P2.

b) 3 data lines L0, L1 and L2. L<n> contains output produced by P<n-1>. P<n> takes input from L<n> .

c) 3 status lines S0, S1 and S2. S<n> specifies the current state of data line L<n>, being
da = data available
sa = space available.

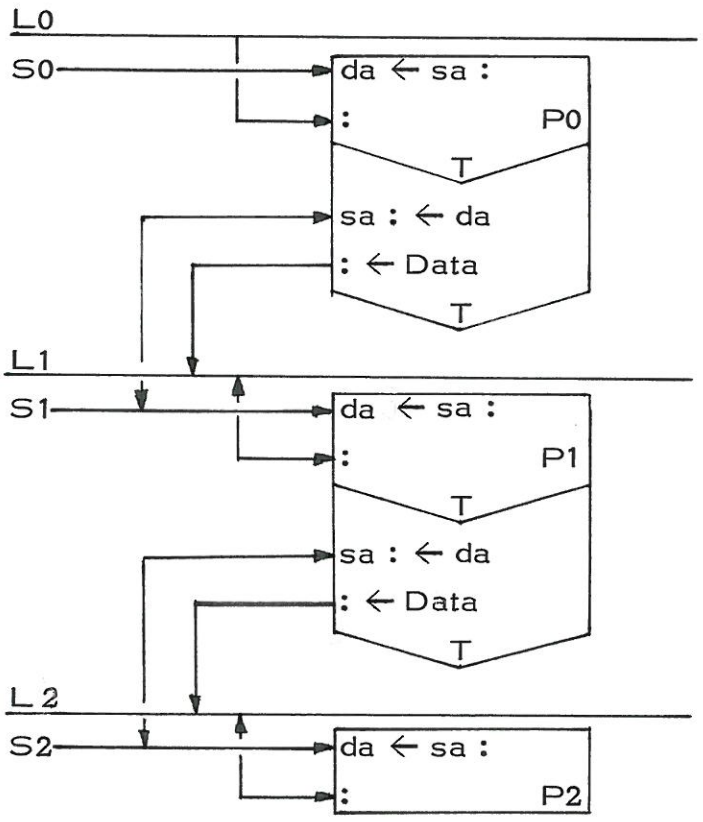
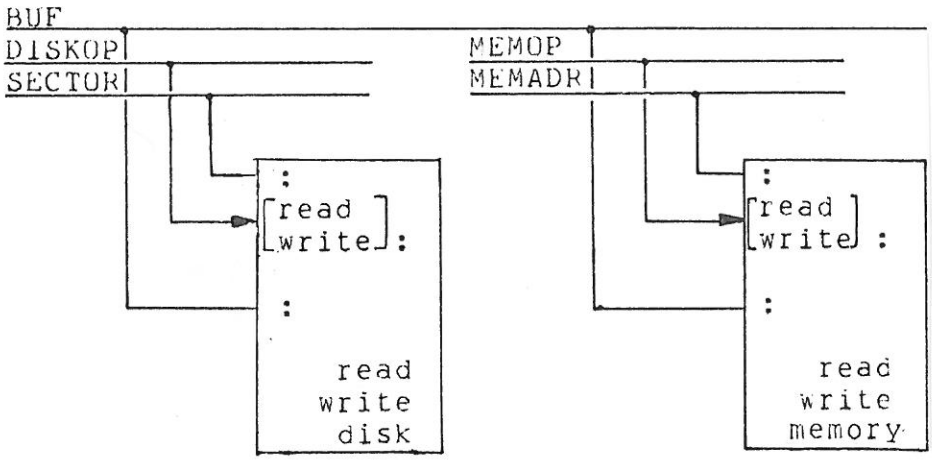


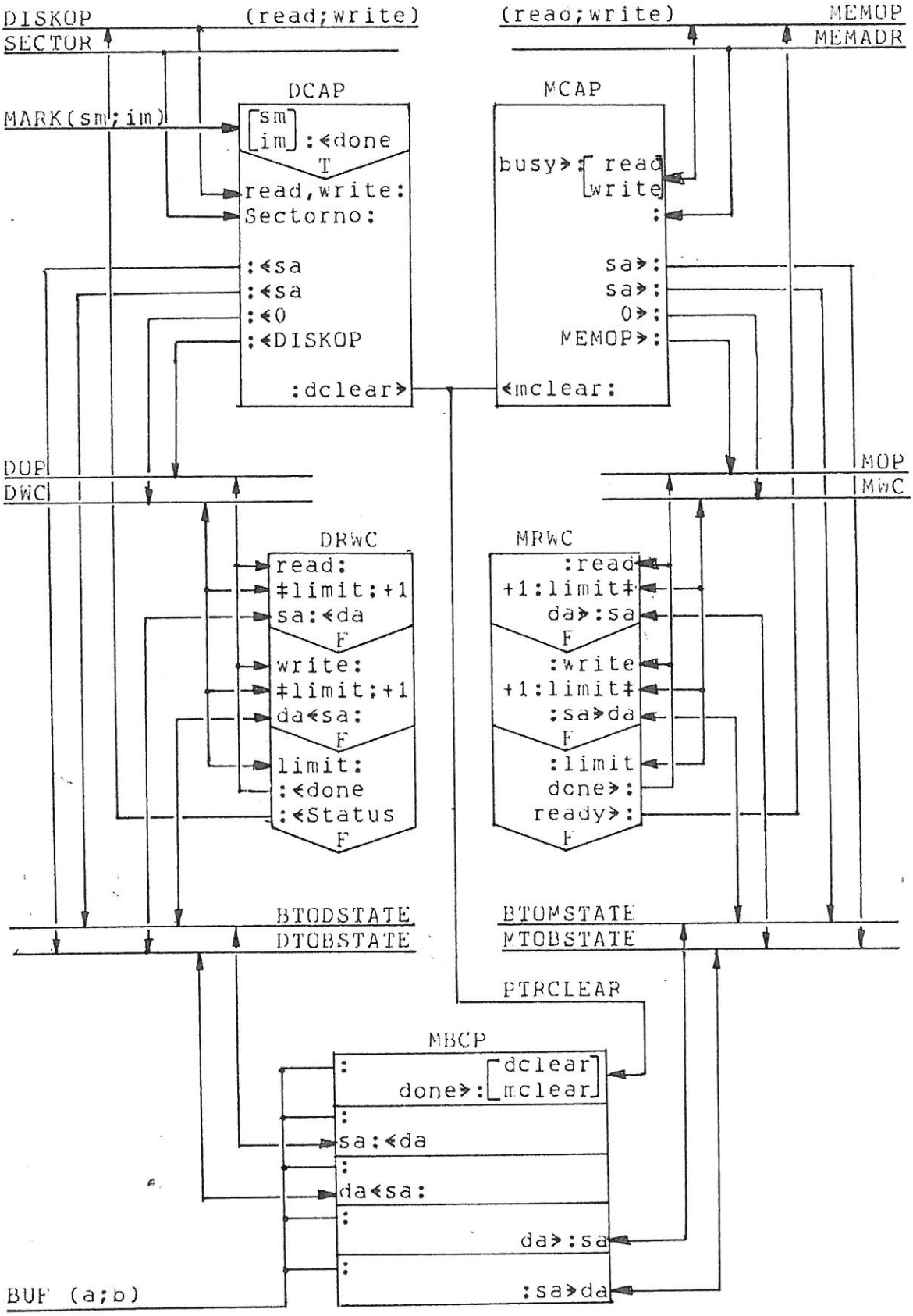
Fig. 5 A Pipeline System.

Each link in the pipeline consists of a compound block of two minor processes. The first process starts when input is available. The second process outputs the internally held data onto a communication line when the line is free. The two processes are interdependent: the second process needs data produced by the first; the first process cannot produce more data before the second has accepted the present load.

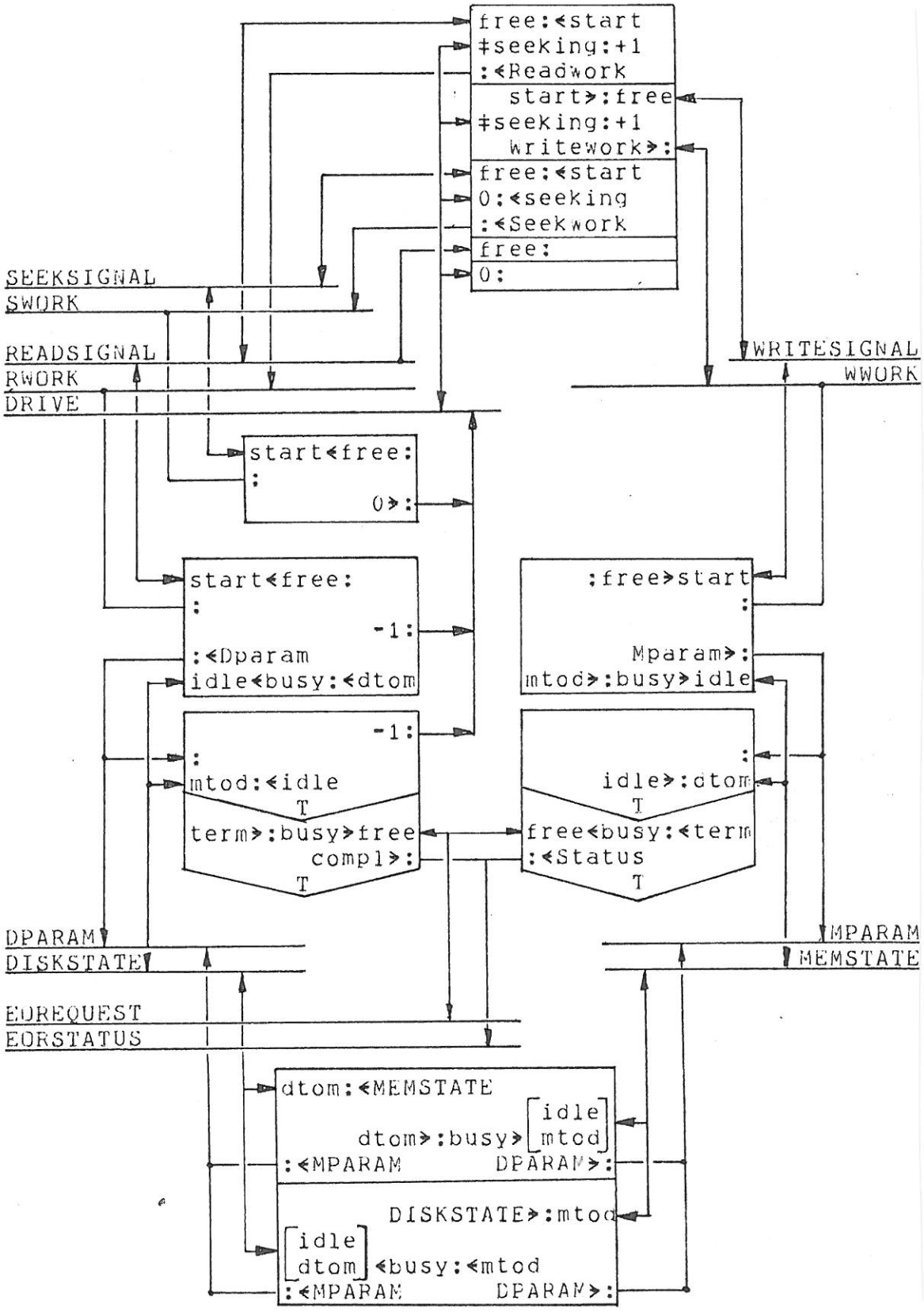
Following are three diagrams, taken from [1]. The first two describe the same disk channel system, at different levels of detail. They are a fine example to illustrate the recursive applicability of the notation. The third describe another system component. The interesting thing here is that this component is at firmware level, while the channel is at hardware level; the notation is used indistinguishably.



A Channel System - the operations performed by this system component are illustrated as 2 processes.



The Channel System – the interdependency between the two processes are here shown in detail.



A model of the set of processes - in the Disk Channel Controller - which are realized by firmware functions.

References

[1] I.H. Sørensen:
"SYSTEM MODELLING"
DAIMI PB-87, Computer Science Department,
Aarhus University, Denmark, March 1978

[2] P. Kornerup, B. Shriver:
"An Overview of the MATHILDA System"
SIGMICRO, January 1975

[3] M. Richards:
"BCPL : A tool for Compiler Writers and System Programming"
Spring Joint Computer Conference, 1969

[4] M.J. Spier:
"Lecture Notes on Operating Systems"
Computer Science Department,
Aarhus University, Denmark, December 1977

SESSION E

Software Engineering Activities

What are the various professional activities, besides the obvious ones: "designer", "analyst", "programmer", and "manager"? What is the relative importance of these activities? To what degree are these activities "respectable"? What technical problems, methods, tools etc. are known to be associated with these activities?

On the Performance Measurement of Production Software

J. Michael Bennett

ON THE PERFORMANCE MEASUREMENT OF PRODUCTION SOFTWARE

by

J. Michael Bennett
University of Western Ontario

Abstract

It is argued that regular performance measurement should be incorporated into the normal development cycle of production software. Reasons why this is not done are suggested. Advantages of doing so are discussed, along with an examination of the feasibility of implementing easily-invoked, yet effective, performance measurement tools.

ON THE PERFORMANCE MEASUREMENT OF PRODUCTION SOFTWARE

Introduction

Over the last decade, we have come to realize that the Software Engineer, as he is constructing his product, should have several fundamental software tools available to him. Most of the tools described in the literature apply to the initial design of the product, its coding and debugging, and final certification. We can now claim, even if we do not follow our own advice, that we have a good understanding of the properties of these tools: we appreciate the usefulness of top-down modular programming, of powerful and friendly editors, of interactive debuggers, of meaningfully verifying the correctness of the final code, etc. But here the matter normally ends. Little thought is usually given to the question of routinely measuring the performance of the finished product. We have little available technology to easily aid the programmer in doing so, even if he so desires. This is an unfortunate state of affairs. I argue that performance measurement is the final stage of the development process and that we should be providing the Software Engineer with measurement tools.

Consider the typical implementation cycle of a piece of software meant to be used by some user community (a compiler, payroll program, numerical software package, etc.). Suppose it has gone through the normal development phase of design, compilation, debugging, certification and release. Suppose too that after some time, in which bug reports and efficiency complaints have been received, the Software Engineer is asked by his management to advise on the feasibility of improving the performance of the product - even to the extent of totally rewriting it.

Consider his formulation of a response to this challenge. What are his facts? He knows, for individual modules, that considerable effort was spent at design-time, trying to choose efficient algorithms. But what he doesn't know is how these system components act as an integrated unit, for it is very difficult to predict global efficiency. If a certain module is little used, an efficiently-coded algorithm is a waste, but others could be used so frequently as to make their algorithm selection very critical. Yet how can he tell?

He is an engineer. As such, he cannot give motherhood comments, or vague mumblings; he must formulate his advise on the basis of measurements. The performance of software can be measured and suitable metrics attached. Then one can discuss meaningfully the sense of recoding specific modules.

To do this, he must have available some technique for automatically instrumenting his code. By this we mean inserting performance-monitoring mechanisms at critical locations in the code, having the instrumentation data collected and then reduced at the end of the run. It must be simple to invoke or suppress these mechanisms. Then actual numbers can be produced on which the arguments for improvements can be based. Later, the same techniques can be re-applied to measure the degree of actual performance improvement.

The need for measurement tools and techniques has never been properly satisfied. Why? Can it be that the problem is unsolvable? If so, we should be able to justify it rationally. If not, why are the tools not available for regular use? To analyze this, let us look at ways in which instrumentation might be done.

What to Measure

First we must consider what it is we want to measure. These are some fairly obvious things:

- One can do straight instruction counts, which can give some indication of what the program is doing.
- Statement execution frequences.
- Location counter samplings - at preset times sample the CPU location counter and use statistical sampling to judge where the program has spent most of its time.
- Procedure entry and exit timestamping and frequency counting.
- Time spent in operating system itself, or in any global routines which are used by the program we are measuring.
- The User might want to look at some special thing which he knows about.

What can performance measurement tell us?

Next we want to consider what kind of things performance measurement can tell the programmer.

1) It permits him to identify areas where large amounts of processor time are being spent, thus suggesting which modules could benefit from more efficient implementations.

2) It tells him about the flow of control in the program - that is what modules are called in what sequence.

3) It permits him to check and measure the real costs of services provided by the Operating System or Run-time Environment. In particular, he can assess the true costs of using various input-output schemes.

4) He can assess the effects of hardware changes on the performance of his product. A system administrator can check the effects on all production programs currently running the system.

5) He can assess the effectiveness of manufacturer-supplied software and complain accordingly.

6) He may also gain new insights into the program. For example if the program runs on a paged machine he may be able to reorganize the procedures in the program so that procedures which call one another are on the same pages, rather than in some random order decided by the loader.

7) The existence of a performance measuring package makes it possible to change the program design cycle, so that the individual programmer is not initially concerned with local optimization, that is coding an individual procedure optimally. It is often the case that we understand what goes on within a single module, but when we put modules together then what was a considerable optimization within a single module may be irrelevant for the performance of the whole program, since it only spends 1 % of its

time in that module. Thus when we are writing a large program we should not worry induly about local performance optimization - we should first get the program running and verified and then afterwards use performance measurement tools to find the critical routines which it is worth our time recoding.

8) Performance measurement may also give us insights into what to do if we ever want to recode the system later.

Principles for performance measurement tools

Let me state some general principles about instrumentation that I want observed.

First (with the execption of taking a little longer, or requiring a little more store) it must not affect the execution of the program in any way.

Secondly, the user himself should be able to specify at what level he wants the program interfered with. Measurement is always an interference, it is likely to cause the program to be larger and take longer, and if one is not vareful it can produce enormous volumes of output. The user should be able to specify how much information he wants to see. I claim that

I claim that meaningful and reasonable information for our purposes can be obtained from just monitoring procedure and operating system calls, measuring their frequency and time.

Thirdly, and I think that this is one of the most important points, the tool should be very easy to use. If we are to get programmers to do performance measurement then the tools themselves must be almost trivial to use.

Fourthly, the tool itself should be general. If we have a standard operating system environment, then any program which runs under that environment should be measurable. For example the measuring tool should not be restricted to programs written in a particular language.

Fifthly, one should have some sort of protection mechanism, so that the tool cannot crash the user's program, and so that the user is prevented from measuring things which are none of his business.

Finally a performance measuring tool should be usable with various different systems.

Techniques for implementing performance measurement tools

Performance measuring is reasonably straightforward to do. There are various possible tools, depending upon what level of detail we want to look at.

Using Microcode

Suppose that one can exercise reasonable control over a micro-programmable computer. Then one can imagine straightforward implementations which would, for example, change the microcode addresses for procedure call instructions to ones which would first store the location, time and frequency of that activation and then continue in the normal manner, with similar action at the return. Some communication would be necessary to link the physical addresses with symbolic names, but this should not be difficult. Selective invocation can be done with a special break-point facility.

This would work all right, although it assumes that only standard procedure calls are executed. Given a reasonable amount of temporary core, one could then make a space-time record of the execution of procedures. Post-processing routines would summarize the data, with appropriate printouts of frequency counts and average timings etc. Monitor calls could be similarly instrumented.

Using a special procedure entry instruction

A second approach assumes the availability of a special type of hardware linkage mechanism. Suppose that the actual procedure call instruction is designed so that it activates the procedure by examining a particular flagword at the procedure head first. After doing the proper linkage setups, the instrumenting-flag bit in that word is examined. If set, a monitor instrumentation routine is automatically invoked. It would time-stamp and frequency-count the entry. If so instructed by the user, it could execute a specially-supplied user software package to do any sort of strange thing. Control would then return normally to the procedure-body. On execution of the procedure-return instruction, the clean-up would be done in the same fashion.

This is an exceptionally clean type of implementation. The procedure instrumentation-flag can be set of reset at any time. For example, it could be done at "monitoring-time" by a special interactive program. It could be done as a matter of course by the Operating System - which would permanently monitor all programs claimed to be in production, and so on.

The handling of calls to the monitor could be done in precisely the same manner, assuming the monitor services to be callable only from a user service procedure that serves as a functional bottleneck. If it is permissible to call the monitor directly from any user code, the calls will be much messier to intercept and measure.

Other alternatives

If one does not have either of the two previous system alternatives, it is much more difficult to implement painless instrumentation. However, it can be done with more conventional techniques.

For example, if one is fortunate enough to have a software system that uses conventional macros for all language calling sequences, then the same instrumenting ideas that we have just discussed could be used. One would have to assume that other linkages would not be instrumentable. The basic Multics linkage was designed in a similar way, although only a frequency counter was added to the procedure-call mechanism. This was not nearly so felxible, but still of value. The same applies of course to all language processors, if we happen to be able to play with the code they generate.

Another possibility is to actually modify the code at execution-time (in a manner similar to DEC's DDT). Some information about the compilation must be provided but it normally can be made available anyways. One could then design the tool to replace the initial instruction with a jump to the performance measuring routine, collect the details necessary, execute the replaced instruction and return. This approach is fraught with difficulties. One must have a superbly debugged tool to permit the replacement of instructions. Sharing code becomes very difficult. Also some details must be assumed about the generated code and the interfaces to the Operating System (we treat these as procedure calls). However, given the case that we can rarely affect the design of our computers, this is the most likely candidate for implementation.

Which brings me to the question which is the subject of this paper: Why are we not performing such measurements routinely, and why are we spending so much effort and resources on the preventive optimization of assumed (and only rarely verified!) inefficiencies?

I do not have the answer, but I venture to offer the following observations:

- programmers often suffer from tunnel vision, tending to see their little component as a crucial contributor to the system's efficiency. They tend to invent optimizations accordingly.
- few people have the converse talent, that of seeing the larger system as its components are being designed and implemented. Therefore, nobody designs global optimizations. [As a corollary, nobody prevents tunnel-visionaries from doing their little local optimizations.]
- given the above, nobody bothers to incorporate into the system all those necessary hooks that would allow it to be instrumented (hooks of the kind I outlined earlier).
- by the time measurements are to be done, we have some brave souls trying to get meaningful information out of the system by running heavy-handed scripts and doing other such tricks. These methods may be suitable for convincing the customer of the system's wonderful throughput performance, but as tools for pinpointing performance problems, they are pitiful.

We simply do not provide easily-useable instrumentation tools which make it simple to collect meaningful software performance statistics. Surely the time has come to design and implement such tools and then insist on their application in the development of quality software.

Discussion following Bennett's Talk

Beitz Computers should be built with performance measuring tools built in. It is often difficult to build them on top of the architectures that manufacturers give us.

Shelness The Amdahl 470 machine has built-in performance-measuring hardware.

Shelness I would like to talk a bit about the measurement tools we used when we rewrote EMAS.

We have built a remote terminal emulator which allows us to simulate a number of users. We have had this tool from early on in the development of the operating system, and it has allowed us on a day-by-day basis to get quantitative evidence of how the system is performing. We run a speed trial every week, comparing the system with the previous week's version and with the old EMAS on the *System-4*.

We measure 3 things in particular:

reaction time — the time until the system generates the first character of a reply to a user interaction.

response time — the time until the system generates the last character of a reply to a user interaction.

interaction time — the number of interactions per minute.

We have found that average reaction time is a poor measure of performance for a fast system, since the variation is extremely high, and a couple of bad reaction times can have an enormous effect on the mean. So now we measure the percentage of reactions in less than 1 second, less than 2 seconds and less than 5 seconds. That gives us a much better feel for how the system is performing.

Displaying Results

Lauesen The major problem with performance measurement is how to present the results so that the ordinary programmer can understand them. A simple profile in terms of the program counter is not good enough.

Shelness The profile should be given together with a source program listing. The Belfast Pascal Compiler is a good example of this. Every compiler should do it. The advantage of this kind of profile is that it is painless — it comes out automatically without the programmer having to do any work.

I believe that profiling is an essential tool and should be used as early as possible in the design process. The results are often surprising — even for a very experienced programmer it is very difficult to intuitively identify the loops where the program is spending its time. Profiling should be at least at the module level, but preferably at the statement level. A profile is also easy to comprehend — much simpler than trace information.

I think that Europeans are much better than Americans about producing diagnostics and performance information. American systems tend to be very poor about this.

Profiles are also particularly useful for teaching programming. It is so much easier to debug a

program if a profile is supplied. Profiling can also be used to check that every exception path in a program has been tested.

Beitz At UCSB the students use a Pascal system which shows on the user's CRT exactly what the program is doing all the time it is running. This is a very nice way of seing what is going on.

Grosch I'm not too keen on that sort of thing. If it is used too early in a student's career it is likely to persuade him that the whole business of programming is like a game of *Startrek* — lights flashing and scores coming up. Universities are training engineers and should be instilling engineering descipline and I don't think that little flashing lights are the way to do it.

Manthey I think that fancy displays are great as long as they provide the information that one needs.

Halphen Monitors which are too fancy can produce large amounts of data which are useless. The important thing is to abstract the relevant information from among the rubbish.

Making Predictions

Lee *"PM"* should not just stand for *"performance measurement"*; it should stand for *"prediction measurement"* as well. Over the last couple of years we have been considering whether we can predict program performance in advance. We have been trying *Monte Carlo* techniques using, for example, flow-charts and we have been amazed by how good our predictions can be. All we need to know is what is the most favoured side of each branch. It doesn't make much difference whether it is favoured 51% of the time or 99% of the time.

We have adopted the nanocode of a QM1 machine so that we can do measurements, and the accuracy of our predictions lies at about the 97'th percentile for most programs.

Lauesen Performance prediction is very important because it gives us the chance to change the design. Afterwards it is only possible to make local modifications, and the really big savings are always made at the design stage because that is where one can change the entire program structure.

Lauesen I would like to report our experience in using our engineering skills in estimating program lengths in advance, as part of our time-schedule planning.

We sat down with our rough plans of the system and made estimates of the sizes of the components. Surprisingly we were able to make very good estimates of the sizes of the modules — usually within about 10% of the final size. These modules were about 100—200 lines long.

Beitz I have an example of the opposite: Harald Sachmann at SDC found a difference of 50:1 in programmers' estimates of program size, program writing time, execution time, and the like. That was in 1968.

Lauesen It depends a lot on how big your chunks are. We found that we had forgotten a lot of modules, so our estimate of the total program size was wrong, but we could estimate the sizes of the individual modules rather well.

Shelness It depends a lot on the programmers — Lauesen probably had an experienced team. I give my students programming problems, and I find a factor of 3 difference even within the solutions which I regard as good. Some of the bad ones are enormous — and they don't even work!

Discussion E.1

Performance Measuring Professionals

Flynn Do you think that the applications programmer is really the right person to do performance measurement? Optimising a system is a very specialised task, and maybe it needs a special kind of person.

Spier We need an identification of what the various computing sub-professions are — the only one which we identify is *"programmer"*. The other people should be given the credit they deserve. What sort of person would be good at doing performance analysis? In my experience nobody wants to do it.

Lee Some of our students have had the reverse problem. They have gone to firms and wanted to do performance measuring as a job, but have been opposed by the personnel department.

Marks I believe that the people who wrote a program are the only ones who can optimise it. And if the person doing the measurements is separate from the programming team then he may not have the political power to make them change their program. In particular once they decide that they are finished, and go off to do something else then he is helpless.

Manthey There is a phenomenon called *"cognitive dissonance"*. All programmers think their own code is great and are not very motivated towards improving it.

Shelness I don't think that the problem lies with the programmer — he is usually aware of the shortcomings of his program. The problem may well be with his manager.

Halphen We must remember that programmers are human beings, with limited knowledge. However they are interested in learning. We have a standard mechanism which we use for measuring all of our application programs. The manager looks at the results together with the programmer, and we can easily see if he has done something inefficient and stupid. Once a programmer has made such a mistake he learns, and doesn't do it again.

SESSION F

Software Management and Economics

What is the relation between technical and managerial or economic issues in software engineering? How do they affect one another? What have we learned from past and present economic trends, and what can we predict for the next 10 years? How do scientific and technical advances affect the software market, and conversely how does customer expectation affect scientific and technical developments?

Developing Large Systems

Graham Pratten

Presentation by Graham Pratten

Developing Large Systems

Pratten Background

Let me begin by saying that I am associated with the CADES Project at ICL. In 1970 ICL was about to embark on the development of a large operating system, VME/B, for its New Range of computers, now known as the 2900 Series. The VME/B development team, led by Brian Warboys, were only too well aware of the problems they had encountered in the development of earlier operating systems. They, therefore, decided to set up the CADES Project. Its objective was to provide better support for the development of large software systems and to help control the large projects responsible for their development. The CADES Project has introduced a development methodology, or approach to system development, which has been used for the development of large systems within ICL, and has produced a Computer Aided Development System (CADES) to support this methodology.

In this presentation, I shall not describe the CADES Project or the methodology and CAD System produced by it. They are adequately described in other papers. Instead, I shall describe some of the problems that are encountered by any large project trying to develop a large system. I shall then consider how we may find a solution to these problems.

Large Systems, Large Projects

Now there may be some cynical people in the audience who will ask *"Why do we need large systems and large projects? Are we just building large systems in order to keep large projects occupied and creating large projects in order to solve the unemployment problem?"* I thought I would start by trying to answer these questions.

Do we need large systems? Recently people have begun to talk about computers on a chip and dispersed systems and claim that all our problems will be solved within a few years. Are these new developments really going to make large systems obsolete? I do not think so. The problems of size and complexity will apply to the chips of tomorrow as much as they do to the large software systems of today. Perhaps the problems of shared storage and shared processors will disappear. However, we shall still need to support the wide range of applications that is already available and will need to extend it to meet future requirements. We shall still have the problem of communication between an ever increasing population of computer users and the problem of large information bases shared by these users. Hence I believe we shall need large shared systems. These may be dispersed over networks of computers but the problem of developing the total system will be as difficult as it is now.

Can the large systems actually provide more capability than small systems? Or do they just reflect the ignorance, incompetence, poor communications and incoherent organisation of the projects which develop them? The answer here is sometimes yes, and sometimes no. Large systems can provide more capability than small ones - unfortunately they do not always do so.

Can we develop large systems with small projects? Do large complex systems really require large projects to develop them? Why can't they be developed by small but highly effective projects? I believe this will only be possible when we have developed a much better understanding of the system design process. However, the requirements placed upon us will then increase to keep pace with our increased understanding, so we shall still need large projects.

So I shall assume that the problems of developing large systems and controlling large projects will remain with us.

Let us look at some of the problems encountered by large projects. I will consider these under three headings:-

1) organisation
2) requirement
3) system itself.

Organisation

Complexity The project may involve several hundred people and thus is a complex system in its own right. It will need to be partitioned into many subprojects, sharing a complex information base, and requiring a complex network of interactions and information flows between them.

Communication This leads us to the problem of communication within the project; the problem of making sure that the people who need to know about particular problems or aspects of the system really do know about them.

Next we have the problem of *dilution of competence*. In a large project we frequently find the more competent and experienced staff managing and co-ordinating their less competent and experienced colleagues rather than producing effective output themselves. One is tempted to ask - would it be better for the best people on the project to develop the system on their own? The answer will certainly be yes if the project is badly organised.

Competition In a large project, different views of the objectives of the project may compete with each other in an unconstructive manner. However, this will prove even more disastrous if it happens within a small project. It may destroy the project.

The opposite side of the coin is *Compromise*. We frequently find cases where two good ideas are proposed. The pros and cons of the two ideas are argued over until finally someone says in desperation *"lets put them together"* and we end up with one really bad idea. Compromise could be considered to be the curse of the computer industry.

Inbreeding is a common complaint of large projects. The project communicates only with itself and develops its own strange view of the world, refusing to communicate with the outside world or even read the literature. Although this problem is less often encountered within a small team, it can prove even more destructive when it does occur there. At least there will be some healthy interchange of ideas within a large project. If a small project suffers from inbreeding, it becomes complacent and stagnant.

Requirement

Length of Timescales - The large project exists over a very long timescale. It starts with objectives which have to be met in stages over a period of years. It is very easy for the project to be diverted onto different and inconsistent objectives, simply because of the length of time involved.

Evolution is a major problem for such a project. The project can see the state of system development it would like to achieve in two or three years time but has difficulty in seeing how to get there, evolving from the existing state. The evolution must be slow enough to avoid disruption of the existing customer base but fast enough to satisfy the more advanced and ambitious customers.

Range of Requirements The large project in an industrial environment will be expected to satisfy a very wide range of requirements with one system. These requirements will frequently conflict with each other.

Support The project will not be producing the system purely for its own use. Instead it will be providing a system which can be supported on a wide range of installations. The cost of developing a system which can be supported in this way is much greater than the cost of a simple one off system.

System Itself

Coherence The key problem here is the need to maintain the coherence of the system. A small project team can develop and maintain a common understanding of the structure of the system they are developing. They can interpret any development of their system in terms of this view. In a large project different parts of the project may have different views of the objectives and structure of the system. The inevitable result is an incoherent system.

A related problem is the problem of *commonality*. In a small project is is possible to recognise when two people are trying to solve the same design or implementation problem. They can then use a common solution to their problem. In a large project the situation often arises where different parts of the project solve the same problem in different ways. That leads to waste of effort in the project and incoherence in the system. Even more seriously, if different parts of the project are responsible for developing different parts of the user interface and they fail to recognise common features in the interface, the system will end up with an incoherent and unusable interface.

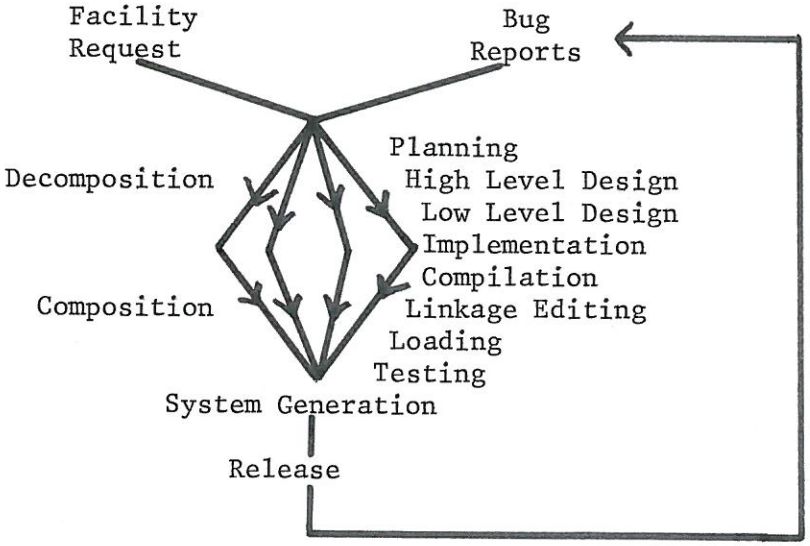
Structural Decay Lehman has demonstrated that a system decays as it is developed and maintained over a period of years. Enhancements and corrections to the system gradually become more and more costly until a point is reached where it is cheaper to completely redevelop the system than to enhance or correct it again.

This arises because the *connectivity* or totality of interactions within the system becomes too complex for the project to cope with.

Solution

Now I want to consider how the problems of large projects may be solved. Are good specifications, good programmers, and chief programmer team structures sufficient? The large project has to bridge the huge gap in time between initial requirements and final goal. It needs to bridge the wide communication gap which exists between people within the project. I believe that in order to fully tackle the problems facing it, the large project requires a *development methodology* which will determine how the project may operate effectively and a *computer aided development system* to enforce and support this methodology.

A development methodology must encompass an *understanding of the total development cycle* including planning, design, implementation, linkage editing, loading, system generation, release and support.



This repeats the point I made earlier that a large project is a complex system in its own right. We need to understand its structuring into subprojects, the activities performed by each subproject and the complex interactions between these activities.

The methodology must be reinforced by a suitable *project organisation*. This organisation must define the roles of each person within the total development route. It may encompass ideas such as chief programmer teams, system architects, central design teams, quality assurance teams, etc. Having observed a number of successful and unsuccessful projects within ICL, I believe that the way these projects were organised was fundamental to their success or failure. At its simplest, this meant having the right man in the right place.

The third feature of a development methodology which I want to mention is the *design methodology*, or the approach to design which must be adopted by the development project. I shall cover this particular aspect in more detail later.

The fourth aspect is the provision of a *feedback mechanism* which allows the user's experience of the strengths and weaknesses of the system to be conveyed to the project and to influence the future development of the system.

The fifth aspect is the provision of a *computer aided development system* to support the development methodology. In a sense this is the easiest part. If we have a good understanding of the development methodology, or approach to development, which we wish to encourage, then we can produce a CAD System to support it. The CADES System we have developed within ICL comprises a database capable of containing representations of the design, implementation, linkage editing and hardware mapping of a system under development and capable of containing a representation of the project responsible for its development including the subprojects, activities, requirements, dependencies, etc. within the project. The CADES System also comprises a set of compilation, linkage editing, report generation, and analysis tools which operate on the information in the database. I shall not describe this further during this presentation. It is adequately described in other papers written by D.J. Pearson, R.A. Snowdon, myself and other colleagues in the CADES Project.

Design

Most of the aspects discussed above (i.e. understanding of total development route, project organisation, feedback mechanism, etc.) help us to constrain the problems inherent in the development of a large system rather than solving these problems. They allow us to develop the system required at reasonable cost. They do not change the nature of the system produced or make it better or worse than it would be without them. One could say that the computer industry has only succeeded in constraining its problems over the last ten years rather than solving them.

A full solution to the system development problem lies in a true understanding of the system design process. Progress over the last ten years has been very disappointing. Activity has been compartmentalised into three areas - program design, database design and system analysis.

The greatest concentration of talent (eg. Dijkstra, Hoare, Jackson, etc.) has been applied to the problem of *program design*. Considerable progress has been made on this aspect of the total design problem.

Less, but still significant progress has been made in the area of *database design*.

The third area, *systems analysis*, has been almost totally ignored by the academic community. It has been left to software houses and manufacturers. Software houses have recognised that there are many potential customers who require help in this area and that the rewards for providing this help will be high. However, the solutions they have provided have been almost totally inadequate because they have been based on inadequate understanding of the system design process. Briefest examination of these solutions reveal glaring errors and omissions. I suspect that the benefits they have bestowed on their users have been psychological rather than technical.

I believe that an adequate understanding of the system design process will emerge eventually from a drawing together of ideas on program design and ideas on database design. A unified view of the whole design process will be created which treats database design and program design as aspects of one continuous design process rather than as different and unrelated disciplines.

Design Methodology

I shall now consider the features which must be encompassed by a system design methodology.

The gap between initial requirements and the actual implementation of a system is so large that we require a *formal representation of the design* which can be shared and maintained by the designers of the system. I do not want to advocate rigid levels of design. I would prefer to see a formal representation of the design which is established at the beginning of the design process and gradually enhanced in small steps as the design progresses until eventually it becomes the implementation of the system.

The design methodology must in some sense encapsulate our best *understanding of the nature of software* and/or hardware. By this I mean that the methodology must identify the types of entity, attribute and relationship we need to specify in order to provide an adequate design of the system. These may include entities (such as jobs, tasks, modules, processes, files, message streams, events, etc.), attributes (such as name, size, order, description, etc.), and relationships (such as made-up-of, uses, owns, types, arithmetic functions, etc.).

We need a *method of representing the design* of a system in terms of our understanding of the nature of software and hardware. At present we have no coherent method of representing the design of a system, no method capable of representing the full richness of the design. We have to represent different aspects of the design using different and inconsistent mechanisms eg. programming languages, database description languages, job control languages, CAD databases, documentation, etc.

We require an *understanding of the transformations* which may be performed on the representation of the design as the design develops. I will cover this point in more detail later.

We need to place *constraints* on these transformations in order to control and guide the development of the design. I will cover this point in more detail later.

Finally we require *control procedures* to ensure adequate authorisation of design, budgetting of human and machine resources, reporting mechanisms, etc.

Transformations

Design methodologies provided by software houses have tried to identify the transformations which may be performed on the design of a system as it develops. For instance, a number of these methodologies have considered a system as comprised of activities (or processes) and data, and have described how these activities and data should be partitioned or transformed into component activities and data as the design of the system develops. However, their definition of this partitioning process has proved unnecessarily restrictive.

I believe the key to our understanding of the system design process is an understanding of the transformations which can be performed on the design. I believe we have failed to recognise the richness of these transformations and have failed to identify and analyse them.

Recently we have been talking to some hardware designers and have observed a phenomena which we have christened the *wire syndrome*. Our discussions began as the designers began their design. They began with a diagram comprising nine boxes with a network of lines connecting them. We assumed that this diagram provided an abstract representation of their design as a set of processes with message streams passing between them. However, we soon discovered that their initial view of the design already had physical connotations. They were already assuming that each line would eventually become a wire or bundle of wires and each box would become a hardware unit. Right at the earliest stage of their design process, they were trying to reach physical "reality" as soon as possible. Even when they did manage to produce a useful abstraction of their design which would explain some aspect of the design, they were tempted to include it in the physical realisation of the design.

We have observed a similar phenomena amongst software designers which might be referred to as the *procedure call syndrome*. Software designers assume, consciously or subconsciously, that the lines which appear in their initial design diagrams will eventually become procedure calls in the implementation of their system and that the boxes will become procedures. If they are not to unnecessarily constrain the design process, they should consider each line in their diagram as a message stream which will eventually be transformed into a procedure call, or a set of records in a file, or a sequence of values held in one data item, or a procedure which receives transforms and sends messages or etc.

Why do hardware and software designers shy away from abstraction? Why do they reach for "physical reality" as soon as possible and even before it is possible? I believe it is because they have no satisfactory method of representing abstractions of their designs *and* no method of representing how these abstractions are transformed into other abstractions or into the implementation of the system. Existing methodologies do not recognise the richness and variety of these transformations. They place arbitrary restrictions on them.

Constraints

Given that we understand how to represent the design of a system and understand the transformations which may be performed on the design as it develops, we will require the ability to place constraints on these transformations to achieve various objectives.

We need to constraint the *connectivity* within the system in order to keep the complexity of the system within human comprehension. This means limiting the scope and number of interactions permitted to each module of the system.

In order to keep the development of the design of the system within human comprehension we need to constrain the *rate of expansion* of the design. This means that the number of new design concepts introduced by each transformation of the design must be severely limited.

Commonality In order to constrain the cost, complexity, coherence, and size of the system and its user interface, we need to encourage the use of common designs, implementations and interface features. I believe this will be vitally important in the future. When we discover a method of representing our designs satisfactorily we should be able to make our systems much more coherent and compact by using common designs within the system and its interface.

Conclusion

In the last part of my talk I have been stressing the importance of a design methodology. I have said that we require a method of representing our designs and need to understand the transformations which are performed on these designs as they develop. I believe that when we achieve this, if we ever do, it will dramatically change the nature of the systems we produce. The improvements in coherence, commonality and compactness will produce different types of system. We shall produce total information systems rather than operating systems, editors, compilers, application systems, etc.

The main problems we need to tackle are the problem of representing designs and understanding the transformations performed on them.

Discussion following Pratten's Talk

Organick Concerning your list of problems of of large projects, can you put weights on the importance of each?

Pratten I personally would say the greatest problems are the size problem, and the range of requirements of the product; also the organisation and communication within the project.

Spier I have lots of additional points for your list: documentation; manufacturing setup; test packages; test procedures; controls; education; training of educators; customization.

Pratten A lot of those come under communications.

Spier Let me give you an example. I was involved lately in making a new machine and so we had a problem. Last time *DEC* made a new machine *(PDP11)* a huge number were sold. Now each customer got some credits to have the manufacturer teach him something. Now suppose the new machine is a hit. Then you have to set up a mechanism to handle the education of 50-100,000 people over the next few years. Now I don't know how big Aarhus University is, but can you imagine the physical facilities this company must produce just to handle the educational needs?

Shelness That has nothing to do with the fact that the system you're producing is large or small. That problem of documentation and education is independent of the problems that Pratten is talking about.

Spier They're not independent at all because it's amazing how something, which could be done by 3 people if you only had 1 user, gives huge problems when you have 10,000.

Lauesen What's the difference if the system has 1 user or 10,000? The educational problem is the same.

Spier If a system has to be taught to these people, it has to be teachable. The people having to make those courses have to get involved in the design process because by the day you announce you want people coming in to be taught, so now there's more people there that have to document and more to read the documentation and prepare the course notes. Now everyone is beefing again, affecting the design process. Now *"communications"* is a good abstraction, but I'm saying it's incomplete in the sense that it does not reveal the horrible details unless one enumerates them. And it's amazing how those things have more effect on the degree of badness of the stuff you're producing than any technical considerations.

I was involved in making Europe's first time-sharing system back in 1964. To look back now, it was incredibly sophisticated. It was made in 1 1/2 years by two people, including documentation. It was made by a commercial company but we never had more than 8 customers. So if they wanted to know something they could contact one of us. I'd hate to think how many years of people it would have taken if we had to cater for 10,000 customers.

Gram I would suggest that if we knew a way of using specification languages, many of these things would fall into place quickly.

Pratten I entirely agree.

Spier You talk about levels of design, and transformations between levels. But those relationships go off in all kinds of dimensions, which are completely unrelated to one another.

Pratten I agree there's a great problem in knowing what those transformations are. Is that what you're saying?

Spier Yes, your transformation ideas are too simple. There may be many dimensions. You may want to do some *"something"*. And then someone says *"yes, but I believe in the future, we may want to negociate some contract for some customer who wants something which is in contradiction to this."* And someone else has some problem with something else. And let's say in the future there's going to be some new piece of technology coming along, so you mustn't do anything which would preclude us from using it. You see, there are all kinds of decisions which sometimes are completely irrational and they are very indirect. Sometimes you forget what is going on, because of certain repercussions. All of a sudden, you want to modify something which looks very nifty here. But it's completely unclear, as you make the change, whether you should remember that thing over there. You don't, and one of the original designers comes and says *"But I told you not to do it because of ..."*.

Pratten You've really described one of the problems of the design process: that of modifying the design. What we need is some way of formally representing the design. Then we can make modifications such as you describe and relate them to the design.

Spier But it's not clear how to represent it, because we do not know what the levels are. I'm asking *"does ANYONE know how to represent that stuff?"*. I've tried it with no luck.

Pratten That's a very important problem area, just finding a way to do that. What worries me is that I don't see anyone actually trying to find a way, except a few software houses.

Spier I don't think that's even true; they just want you to believe it.

Pratten That's right and they're making a lot of money out of it, but they're not actually solving the problem.

Shelness One of the problems we always face when we try to produce something like a concepts manual, is that the documentation people, who are brought in to make this concepts manual, think that basically there is only one set of concepts for one view of the system. But, certainly in the case of an operating system, there are multiple views. One of the things that gives students such great troubles in learning about operating systems is that until they understand that there are multiple orthogonal views of the system, all of which are equally valid, they have real difficulty. It's very difficult to explain to students, for example, where precisely memory management fits into an operating system. The memory hierarchy is, in some sense, one dimension, and you can describe a system solely in terms of its memory hierarchy and movements within that hierarchy, which is completely independent of the process structure of the system. Process structure of a system is another dimension and you could describe that in some sense in terms of interactions of messages which pass between the processes. The third dimension of the system is the levels of abstraction which are available for the users: they have a number of abstract machines which are created out of a number of processes that live in the storage hierarchy. The really difficult problem, as with any orthogonal system, is how to put these dimensions together. You don't have one defined description; you have a whole bunch of orthogonally defined descriptions. The only book which reasonably solves this problem is a recent one by Lister. It shows how the various parts fit together. On the other hand, most of the literature presents only one point of view, they don't give you this sense of coherence.

Gram As we design very complex systems, we can no longer afford to do specification in minute detail — it is too complicated. So we need off-the-shelf building blocks which will be the accepted main components of the system. For example nowadays one can no longer afford to build an aeroplane from scratch. You start with the available technology for engines, wings, electronics, controls, etc. You specify the plane's design at the level of such major components. We need libraries of useful modules.

Shelness More generally, how do we build things that are too big for us to be able to describe what they do?

Stoy Mathematics has done that. When you deal with the theory of differential equations you are taking the natural numbers for granted; you don't bother to re-invent the numbers first. Mathematicians have been able to cope with that hierarchy by having well-defined interfaces: the statements of the theorems, which are then used without any reference to the mechanics of their proofs.

Derrett . . . and are all too often used mechanically without reference to their definition. You produce a useful but restricted theory, as is often the case in applied mathematics. It is only supposed to work if this and that and the other are true. And then physicists and engineers use it for everything. This goes back to Mike Spier's point about defensive programming.

Project Organisation

Gram In the verification process of your design schedule, do you set up separate groups to do the measuring and validation?

Pratten In the *VME/B* project, we have a separate group providing that function.

Beitz Do the people designing these modules actually test them before the independents come along?

Pratten Certainly.

Beitz Doesn't it seem sort of paranoid then, checking it twice. Wouldn't it be easier to test the entire checking in one sub-organisation?

Pratten The problem there is that if you do that, the people just hand over new routines to the checking group, without any checking themselves, because of time-scale pressures. We're much more in the view that the actual group producing the module retains responsibility for it right the way through, although it may ask the central group to give it some help in the actual validation. The responsibility resides with the project, not the central group. Otherwise the central group would just get a load of rubbish, mainly because of time-scale pressures.

Shelness Do the people in ICL know what the organisational structure is? — chief programmer team or the like.

Pratten Well, in the CADES group itself, they are very aware what the structure is.

Shelness Is it important that they know?

Pratten Yes I think it is! Because they know where that decision is going to be taken at the end. It stops the conflicts you can get in a project. There's always the right person to make the right decision. I think the role of the chief programmer team, or whatever, is very important; it provides some cohesion to the system.

Spier You mention that ICL's sucessful projects depend on key people at the right places. How do you detect those key people?

Pratten Well, I have observed people while they are on projects, and some just stand out.

Lauesen Did you find these people as the project was finishing or could you identify them at its beginning?

Pratten To take ICL's example. Eight years ago we had what we thought were a dozen key people. We would have expected them to be so now. Out of that dozen, about one third have proved to be good. There's an evolution going on here, obviously — people evolve into jobs.

Derrett Your key people, do they have management abilities? Are they programmers or managers?

Pratten Actually they are more managers and don't program. One doesn't take flow charts in to the project leader. However there are various kinds of management skills.

Spier I've seen places where they installed chief programmer teams. Previously they had some little nameless guy in the basement who did all the real work. They always made him feel very guilty about how deficient he was, to keep him there. Then they needed a Chief Programmer Team because Brooks said so. So they hired some guys from Stanford. But they still had this little guy sitting there doing all the work. So the project was a success and they decided that the CPT approach was good.

Grosch [*to Pratten*] I think that what you're talking about, Graham, is a humanised kind of management that applies in what you call large (I call medium) sized projects — 50 people or less. If you're going to go into IBM or the government, you've got to manage 1000 or more people on these projects. And for that, you really do have to have more hierarchical structure. You said yourself that people grow into jobs. The fact of the matter is that you might as well have them grow into an easily-managed hierarchical structure, as into a specialised thing that just happens to fit your particular prejudices, once you get above 50 people. Up to 50 people — organise them to fit the people by all means. Above that, it's probably not going to be possible, just because of the nature of IBM, Fujitsu or the DOD. You've probably got to have layers of management, reports, etc. Well, you just can't help yourself. That's antithetic to your kind of skill.

Beitz You still have an option. The hierarchy can be horizontal or vertical.

Grosch No you haven't. One guy can only manage 6-8 people and still know what's going on. If you're going to have 1000 people on a programming team, you've got to have 3-4 layers of management.

Pratten But you run into the problem of technical managers *versus* administrative managers.

Grosch Don't get me wrong. I'm in favour of the managers knowing what they are doing. I don't believe in professional managers, in the sense that you hire a guy whose greatest recommendation is that he moved the outdoor lighting department from Schenectady to North Carolina! That doesn't work, and *GE* and *RCA* proved it. But there are people that do understand the technology (at least yesterday's technology), and possess management skills.

Spier But they get obsolete very quickly. I admire Chairman Mao for sending the bureaucrats into the fields once a year to dig potatoes, or whatever it is they eat in China. We ought to do this with managers.

Shelness If you look at architects' offices for example, you will find that there are various structures alongside each other. The management heirarchy and the intellectual heirarchy are not necessarily the same.

Grosch I think that the thing that really matters is the guy at the top. A big organisation needs a special kind of man to lead it — a man in a million. If you can find such a guy then things will work, otherwise they won't. These people combine technical and managerial skills.

Function and Appearance

Beitz My background came out of architecture. I was trained as an architect and the key thing in architecture was a thing called *design*. And in design, we had something which I think is missing in the education of Computer Scientists. There were two basic concepts: We had an appreciation of history — yet nobody in computing goes and looks at precedents. I think that we should be *made* to look at precedents — other people's Operating Systems from the past, or redevelopment of Operating Systems. The second key thing in design is that someone has to be *the meanest SOB in the valley*. He has to unify the whole thing and control it. That's what makes for good design. Unless you have the Frank Lloyd Wright, etc. it doesn't look like a unified design, it looks like what it is — a dog's breakfast. I think that although we've adopted the language of design (we say *"sotfware architecture"*) yet we're not borrowing from their discipline. For what is *"engineering"*? As far as I'm concerned, you look at precedents, you do experiments, you develop a set of metrics to measure things, then you do some analysis and say "Now we know that this bridge, given that it fits these things that we've measured in our experimental environment, will stand up." We learn how to use those metrics to predict the performance of the system. We haven't got an analogue in Computer Science.

Spier Of course not. The stuff is abstract — you can't visualise it.

Shelness Yes you can — for example the publishing of OS6 and the commentary.

Derrett Architects can look at the history of their trade. You can come to Århus and see the results of Arne Jacobsen and his colleagues. In computer science we have only the rather short papers of the Dijkstra's to describe their systems. A lot of the published articles either contain approximations to the truth or downright lies, because they are written before the systems actually work. They describe how the authors *think* it will go. What we need are things like Stoy has mentioned — that people actually publish their designs so other people can look at them.

Shelness Having once written a compiler, I enjoy reading them now. (I read the *PASCAL-P* compiler and find it *"not so groovy"*.) When you read a lot of other compilers, you begin to see that you can write them so they are clear and correct and understandable. I think we're beginning to do that with Operating Systems. I enjoyed reading *OS6*.

Lauesen But there is a difference between the looks of buildings and their function. Very often they look lovely but function in a much different way. I have observed the same parallel in system design. Some of the very beautiful systems (the *RC4000* operating system for instance) are horrible to live with.

Halphen I want to throw another argument into the analogy between classical architects and system engineers. Don't you think that architecture is more static? Pratten was talking about *"requirements"*. We know what the requirements of a bridge or a house are. But do we know what the requirements of *OUR* systems are?

Pratten We do have some written down.

Grosch There is an idea about a house in general that we all understand. It must have certain components: a roof, be cool in the summer, hot in the winter. But we don't know that about Operating Systems. They're a bunch of hobbies you guys have created. One guy was too lazy to walk across the room to mount a tape drive, so you've got something in there that mounts tape drives. Some other guy was too lazy to go to the line printer, so he's got a console typewriter. And you have to have a whole bunch of dumb software to run it. The guy at the top who says yes and gives you the money for it, is doing so on the basis of whether it's saleable and not on the basis of whether it's necessary or handsome.

Beitz Architecture has a history that goes back a little way — it's mature. I don't think we've started to reach maturity in Computer Science.

Pratten Well, I don't think we're in the architect stage yet. We're in the 14th century, building a house. We have a vague idea of what the house should look like, so we put stones on top of each other, hoping that it will work out.

Goal Driven Software Engineering

Robert J. Flynn

GOAL DRIVEN SOFTWARE ENGINEERING

Dr. Robert J. Flynn
Polytechnic Institute of
New York
333 Jay Street
Brooklyn, New York 11201
Home Address:
82-79 164th Place
Jamaica, New York 11432

GOAL DRIVEN SOFTWARE ENGINEERING

Abstract: Software engineering is viewed as a goal driven exercise. In that context the question of computer security and privacy is addressed with consideration given to record security. In addition, the question of lifecycle system costs is addressed in relationship to computer architecture.

The term "software engineering" is reasonably provacative. Not computer science, not the existensial joys of programming, not software reliability, not even programming. Clearly the operative word is "engineering." Engineering, classically, is the use of science in order to achieve a goal - be it to improve the quality of life, to maximize profits or constrain costs. Engineering is a derived, synthesis operation. What distinguishes software engineering from computer science is the clear notion of a goal, a purpose, a meaning.

Mathematical theorems are either true or false. They need not mean anything in that they may have no apparent relevance outside themselves. Theorems need not be justified, only proved. Programs can be proved correct but as such they are now theorems and theorems are only true or false. Programs obtain their meaning if they are useful, brought in under budget and robust enough to survive a system respectification in a year without having to be rewritten totally.

Hannah Arendt, in a recent posthumous essay on thinking, argues to the same point in discussing the distinction that Kant made between reason and intellect. "The need of reason is inspired not by the quest for truth but by the quest for meaning. And truth and meaning are not the same thing."

In order to discuss any topic in software engineering there must be at the beginning an understanding as to what constitutes the desired goals. It is not uncommon to obscure this point. Is the goal of a given university to educate or produce ideas? Perhaps both? A failure of all parties to agree on a common goal typically results in unnecessary conflict in the policy making and implementation phase.

Clearly one can solve an ordinary differential equation numerically with a Taylor series. That is true. But if the goal is to develope code which can be used on a computer with ease and flexibility on a wide variety of problems then the Taylor series technique becomes virtually useless in relationship to perhaps, a variable step size, variable order predictor corrector technique.

If the above is in order, it creates a context in which to inbed certain specific software engineering discussions.

1. What is the goal of computer security?

Clearly there are certain primitive goals that all agree on. The basic goal is clearly the survival of the system against accidental or intentional destruction. Beyond that, one can clearly get universal agreement on the need to insure rational, secure access control to the various users' domains of program space and information space. If one were to accept, for the sake of argument, that one could insure the satisfaction of these goals against all but a truly pernicious attack from a malelovent source, we are still left with certain difficult areas.

The establishment of a large computer data base is now often viewed as a self evident threat to the privacy of people. The State of Texas recently ruled that the establishment of a computer based file on child abuse and abusers was an inherent threat to the right to privacy of the individuals involved. In contrast to that ruling, the State of Massachusetts recently found that an index of criminal cases maintained privately by the courts must be revealed to all under the general principle of freedom of information - the right to know.

The tacit assumption often made is that access to computer based data cannot be properly controlled. There is a growing fear of content searches. Computer based information systems are, increasingly, the subject of litigation.

Clearly the right to privacy and the right to know are delicately balanced and often opposed forces. Somewhere at the fulcrum of the balance rests an unspecified technological problem.

Is there a professional standard of data maintenance appropriate for the maintenance of large data bases? Is there a critical mass of information about an individual which by its very nature is a threat to the individual? When it is maintained on a computer? Clearly yes. These questions become one of public policy more than computer policy. However, it really is a technological problem.

The basic question to be asked is "protect what from whom?" The most typical unauthorized data access to an individual's record is not from a totally external snooper. The individual may in fact be authorized to see some information. How does one maintain data in a hospital, for example, which allows a newspaper reporter to see how much money was paid by a government health insurance program but nothing else; which allows Mr. A's physician to read/write Mr. A's diagnosis but no one else's; which allows all billing information to be retrievable by appropriate officials but not by others. How does one do this efficiently?

To date most approaches to computer security have been both binary and file oriented. Classically the above problems result in fragmented files - medical information in one file, billing information in another file. One then protects each file by a password. Passwords are the weakest protection available to anyone. All the analysis in the world won't change the basic reality that passwords are kept and exchanged by people and are binary in nature in that one password unlocks the whole file. One has the additional problem of where to

store the list of passwords in the system.

Encryption is a significant enhancement in computer security rendering the data impervious (leaving aside the question of malicious decryption to anyone who gains access to data set without the key). Again, however, the key is both binary and file oriented.

Carson, Flury and Welsh, as well as Carson, Summers and Welsh suggest a selective encryption technique and terminal in which records are buffered in an intelligent terminal on both input and output. Selected fields are encrypted by supplemental hardware, allowing records to be maintained in a ringlike structure. This approach would encrypt, for example, the medical diagnosis but not the physician's name, the individual billing items but not the balance due.

Clearly if too much data is encrypted or protected, the net result is that nothing is protected. Nothing is protected in that too many people must necessarily have access to the key or password. If access is not granted to some fields, then any access implies total access to all records and fields. The intent here is to allow only a physician to see a diagnosis and only a bill clerk to see an itemized bill on any patient.

Flynn and Campasano take this notion one step further in suggesting a data dependent key using a selective encryption terminal for the encrypted data. Here the observation is that Mr. A's physician is entitled to see and alter Mr. A's diagnosis, but not his bill and not the diagnosis of any other patients other than his own. Here the attempt is to generate a key or keys for fields on a record by record basis, not a file basis or on a field basis. The right to privacy is an individual right. A tax examiner using a computer to examine Mr. A's tax records should not be able to at the same time access the tax records of his neighbor or anyone else.

An automatically generated key or password is needed if one has a genuinely record by record key. If not, the record keeping for the keys would be impossible. However, individually assigned keys is the only way to break up the ringlike structure.

What is proposed is the following:

The goal of the computer security, beyond the survival of the system and the security of program space, is to maintain information in privacy with restricted accessability. Information about people and programs increasingly are subjected to conflicting rights -- rights to privacy and rights to know. These conflicting rights imply a standard of data maintenance that is perhaps novel. If too much information is encrypted or protected, too many people need the key or password and the result is less security. Perhaps what is called for is selective encryption of certain fields within a record. However, a ringlike structure results if the same key is used for a given field of all records.

The ring structure implies that a person authorized to examine a given field in a given record uses the same key for that field in all records, and therefore, has access rights dependent on field authorization not record authorization. One technique might be to encrypt various fields of a given record with various functions of identifying plain text information. This would, inhibit context searches designed to identify individuals with certain properties if the given fields were encrypted.

One could go further and implement these functions in a hardware terminal and then make the problem of security one of the physical security of the terminal. The key for Smith would be different than the one for Jones and one could obtain the keys only if one had access to the specified terminal.

One could thus extract from identifying plain text data on Mr. A numbers, a function of which would provide a partial key to some of Mr. A's data. Access to a given terminal which computes the correct key for looking at Mr. A's medical records need not be sufficient to look at Mr. A's diagnosis. A supplemental key might be extracted from a correct logon by either Mr. A's physician (who would have a special key of his own) or someone who knew Mr. A's physician's code. To make context searches totally impossible, one could go further and encrypt the plain text source of the keys, although this is not totally necessary.

Thus data security can be both physical security and record security.

Beyond this point there is a larger question that needs to be addressed. How in general does one allow the software engineer to express the relationship between the examiner, the fields and the individual examined? Is the correct solution a hardware/software solution? There must be simple techniques to leave data retrievable but at the same time eliminate context searches.

The goal of computer security is to protect records more than fields. Conflicting legislation and regulations suggests that unless prudent software engineering solutions are forthcoming, the end user will be perhaps constrained in the maintenance of computer data basis in a totally uneconomic manner.

2. What is the goal of software engineering?

It would appear obvious that the goal of software engineering is the minimization of the total system lifecycle costs while maintaining a level of support which is adequate. Techniques for estimating and comparing total systems lifecycle costs are, however, by no means so clear and obvious as to make major purchase decisions self-evident.

If the goal is cost then perhaps a seminal concern to the software practitioner should be software systems accounting and cost estimation. There is a natural tendency to measure what is measurable and caliper what is caliperable more than to ponder what is really imponderable.

There is a reasonably straightforward process involved in benchmarking competing systems with a standard job mix. One can make various estimates of the impact of an altered cycle time. Certainly, one can time out certain processes, if time critical, to see if certain performance levels are to be met by a considered system. One can price a line of code of a certain type on a given system with given programmers and documentation levels. No decision is ever made without considering the various financing arrangements that are possible. How does one handle lifecycle estimates?

A recent distributed system selection allowed a different, detached view of the process. One seldom has an opportunity to look at a total package, without the encumbrances of an existing software investment and without an existing machine or inhouse group of programmers. Often if the existing gestalt is very complex, the gestalt tends to dictate the mode of analysis

unless one has infinite time and an extraordinary amount of corporate flexibility.

An insurance administrator with a reasonable data base (approximately 20 million characters with growth not to probably exceed 60 million) had in place an existing apparatus consisting of purchased computer services for maintenance of the data, billing and accounting. Manual procedures were used for general claims processing. What was unusual was only the simplicity of the analysis. It is not suggested that this analysis is appropriate for other systems.

A primitive decision made was that lifecycle total systems costs were the primary consideration, almost the only consideration. A product lifecycle was assigned. Both the hardware and software costs were depreciated, for the sake of analysis, over the same time frame. The investment, replacing a clear outside charge, could be returned in a fixed time (less than two years) against a projected minimum life of five years.

Software development if treated as an expense item, typically has no predetermined lifecycle and hence is expected to live on, long after its true relevance has significantly weakened. Amortizing software with hardware naturally suggests a different perspective.

In addition, a time value of reports and transactions - a time value of knowledge was assigned to the principle transactions. Certain transactions had to be completed in a certain fixed clock time without reference to the length of computation time they took. If this minimum could be accomplished then the lifecycle costs would be the only thing looked at.

This is a simplistic view of the decision making process in an application which is inherently simpler than most decisions. However, it resulted in a nonstandard solution. The decision was made to purchase a distributed processing mi~~cro~~computer system which had evolved from a large key to disk system. This implied a system with an inplace operator interface that was extremely well thought out. It resulted in the acceptance of a non-standard COBOL like language based applications environment which eliminates environment and data specifications, data being specified via CRT formats which link logically to the data that passes thru and a restricted verb set. It resulted in accepting a machine which would benchmark out as being slower than a comparable unbundled minibased system.

The decision to choose the path taken was based only on lifecycle costs estimates. It was felt that the software development costs to be incurred would be significantly less than those on a competing system with lower hardware costs and faster CPU in which one would have to create a viable user front end from scratch and from the immediacy of the development that would take place in an interactive environment. To date, this projection has been upheld. In addition, the simplicity of the front end implies that no computer personnel need be hired for the life of the system.

There is nothing earth shaking in the above except that software development and maintenance costs were bundled into the total costs estimates and led to the acceptance of the nonstandard solution -- a slower machine with a nonstandard software package.

All of this leads to the point in question. Lifecycle costs are the goal. However, most purchase decisions are made on various components of cost - benchmarking performance,

standardization and portability of software used. This is due in part to the quantifiable aspects of these components. Does this process in someway inhibit a truly radical rethinking of the whole process? Can one ponder the imponderable (or at least nonquantifiable)?

What are the lifecycle costs of allowing access to unallocated variables? What are the lifecycle cost improvements in a system which restricts the reference space for each module so as to enhance the reliability and encourage the modularity of the software developed on it? What are the lifecycle costs of a system which allows reference to array elements outside the defined array extent? What are the lifecycle costs of a system which allows reference to an integer variable as a floating point variable? What are the costs associated, with allowing a program to mix modes in arithmetic operations (divide by a string variable)? What are the costs in program complexity in having a system which partitions its name space into hierachial memory spaces (registers and memory)? Does the performance enhancement justify the load/store overhead and the increased complexity when viewed from the perspective of lifecycle costs as opposed to benchmark tests? What software reliability and cost impact results from having a programmer aware of double-word boundaries? Is a system reducing the costs of the user if it allows, via a floating to integer conversion, a program to access an array element via a floating point index? How much real cost is incurred by fitting variable length data into fixed length forms? What happens if all costs are recast as lifecycle costs which include software development and maintenance costs with hardware costs?

There are many such imponderable questions. Myers has an extensive list of such problems and proposes a software reliability directed machine. This includes decimal arithmetic, token storage descriptors including type, length and allocated status and other enhancements. Spier argues to a domain architecture with a restricted module refence space. There have been numerous language based machines. M. Flynn observes that the ratio of M-instructions, instructions such as load and store which move data items within a storage hierarchy to F instructions, instructions that perform computational functions in that they operate on data is perhaps excessive (2.9 for S/360) and introduces a directly executable language concept. Barton argues against fixed word length. Perhaps in a distributed processing world, a processor need not be all things to all people, always - except always cost effective. There are, of course, many others.

Many of the points raised are not quantifiable. Improved timing is easy to achieve and sells machines. Somewhere, however, the imponderable question lurks in the background. What are the long term costs associated with going fast? The total cost to the end user is somehow missed. Should one have a token architecture or whatever? Presumably it will execute significantly slower. But is the caliper time or money?

The end user, it might be conjectured, is more interested in having a system reduce program maintenance costs than in reducing time. The end user it might be conjectured, is more concerned with standardization of program interfaces to accounts receivable packages than in the standardization of yet another language. (Consider, for example, the ease of access that has evolved in using standarized numerical analysis subroutines, such as Shampine and Gordon's DE package, that are available through centralized code distribution centers.)

standardization and portability of software used. This is due in part to the quantifiable aspects of these components. Does this process in someway inhibit a truly radical rethinking of the whole process? Can one ponder the imponderable (or at least nonquantifiable)?

What are the lifecycle costs of allowing access to unallocated variables? What are the lifecycle cost improvements in a system which restricts the reference space for each module so as to enhance the reliability and encourage the modularity of the software developed on it? What are the lifecycle costs of a system which allows reference to array elements outside the defined array extent? What are the lifecycle costs of a system which allows reference to an integer variable as a floating point variable? What are the costs associated, with allowing a program to mix modes in arithmetic operations (divide by a string variable)? What are the costs in program complexity in having a system which partitions its name space into hierachial memory spaces (registers and memory)? Does the performance enhancement justify the load/store overhead and the increased complexity when viewed from the perspective of lifecycle costs as opposed to benchmark tests? What software reliability and cost impact results from having a programmer aware of double-word boundaries? Is a system reducing the costs of the user if it allows, via a floating to integer conversion, a program to access an array element via a floating point index? How much real cost is incurred by fitting variable length data into fixed length forms? What happens if all costs are recast as lifecycle costs which include software development and maintenance costs with hardware costs?

There are many such imponderable questions. Myers has an extensive list of such problems and proposes a software reliability directed machine. This includes decimal arithmetic, token storage descriptors including type, length and allocated status and other enhancements. Spier argues to a domain architecture with a restricted module refence space. There have been numerous language based machines. M. Flynn observes that the ratio of M-instructions, instructions such as load and store which move data items within a storage hierarchy to F instructions, instructions that perform computational functions in that they operate on data is perhaps excessive (2.9 for S/360) and introduces a directly executable language concept. Barton argues against fixed word length. Perhaps in a distributed processing world, a processor need not be all things to all people, always - except always cost effective. There are, of course, many others.

Many of the points raised are not quantifiable. Improved timing is easy to achieve and sells machines. Somewhere, however, the imponderable question lurks in the background. What are the long term costs associated with going fast? The total cost to the end user is somehow missed. Should one have a token architecture or whatever? Presumably it will execute significantly slower. But is the caliper time or money?

The end user, it might be conjectured, is more interested in having a system reduce program maintenance costs than in reducing time. The end user it might be conjectured, is more concerned with standardization of program interfaces to accounts receivable packages than in the standardization of yet another language. (Consider, for example, the ease of access that has evolved in using standarized numerical analysis subroutines, such as Shampine and Gordon's DE package, that are available through centralized code distribution centers.)

REFERENCES:

1. Arendt, H., "Thinking," The New Yorker, November 21, 1977.
2. Carson, Flury, Welch, "The Selective Encryption Terminal: A New Approach To Privacy Protection," M 75-76, The MITRE Corp., METREK Division, September, 1976.
3. Carson, Summers, Welch, "A Microprocessor Selective Encryption Terminal For Privacy Protection," Proceedings of National Computer Conference, AFIPS Press, Montvale, N.J., 1977.
4. Flynn, R, Campasano, "Data Dependent Keys For A Selective Encryption Terminal," Proceedings of National Computer Conference, AFIPS Press, Montvale, N.J., 1978.
5. Myers, "The Design of Computer Architectures To Enhance Software Reliability," PhD Thesis, Polytechnic Institute of New York, 1977.
6. Spier, M. J., "A Model Implementation for Protective Domains,: International Journal of Computer and Information Sciences, 2(3), 1973.
7. Spier, M. J. "A Pragmatic Proposal for the Improvement of Program Modularity and Reliability," International Journal of Computer and Information Sciences, 4(2), 1975.

8. Flynn, M., "Computer Organization and Architecture", Lecture Notes in Computer Science, V60, Springer-Verlag, Berlin, 1978.

9. Barton, R. S., "Ideas for Computer Systems Organization: A Personal Survey," in J. T. Tou, Ed., Software Engineering, Volume 1, New York: Academic, 1970.

Lifecycle costs are relatively imponderable, except in terms of what they have been in the past with the given structure. They are totally weighed down with applications program maintenance costs and a variety of other imponderables. Is the difficulty of the analysis of the future keeping radical rethinking of the future from taking place? If so, at what cost?

Security and Privacy

Flynn I personally have seen the payroll of a not-to-be-identified university three times. Once because a young man in the computer center discovered a way of seeing the payroll; once because students accidentally broke the system and made the thing come out; and lastly because some fool had taken the entire payroll listing and left it on a desk in the computer room.

If you don't want this to happen, you go out and get yourself some security. But there is a more modest goal: privacy. Privacy is a very subtle matter. There is a quote which always comes up on every chapter on privacy. It is by the American judge Louis Brandeis, who said in 1927 that *"The most fundamental of all rights is the right to be left alone"*. I checked, and it was a dissenting opinion! That is, it has no weight in law! So privacy is not an absolute right at all, and it appears in the domain of computer systems, where many have a legitimate need to know. I cannot easily see a student's grades. If I want to, I have to make an appointment, go to the registrar's office, and look at the folder there. I can't simply take a bunch of folders home, to read over the weekend. I think in this case I have the right to know that information, so that I can help the student plan his courses in the coming semester. Privacy and the need to know compete.

If we don't now look at the structure and design of databases in the reality of the competing goods in society, to see how they compromise between somebody's right to know and somebody else's privacy, then we will implement a something and we won't know what it does. As databases become larger we will, no doubt, have more and more secure systems, secure, that is, from outside invasion. Will the systems, however, protect my privacy? Will the systems be designed to protect me against unwarrented snooping by a programmer at my employer's computer center?

Spier When you talked about your encryption scheme you said that for example *Mr. A's* physician will have to give his own (the physician's) code together with some information on *Mr. A* in order to see *Mr. A's* record. — Isn't this just giving two passwords instead of one?

Flynn No, because it is also necessary to use the physician's terminal. The ultimate security is still physical. Certain data can only be accessed through a specific terminal. I can lock up that terminal, or I can post an armed Marine guard to watch it. That physical security is a lot more secure than anything provided by software. As for two passwords, the privacy of A's records and A's physician's need to know is a relationship that exists between two people. Why not have a sign-on procedure that expresses that relationship. I don't believe that physicians have a right to know anyone's medical history (except their own patients').

Derrett The data sitting on the computer is also protected — even if someone steals the discs he can't read them if they are encrypted. Also it doesn't do him any good tapping the transmission line — it is the terminal which does the encryption.

Grosch Concerning physical security, what about signature machines? I think there are two machines. One British, out of NPL or the Post Office, and there's at least one at SRI in the States.

Spier If you can verify the signature, somebody may still fake that image before it is looked at by the machine.

Shelness That's not what happens. It looks like a hard problem, but you want to realise that if you verify while the guy's actually signing, it is a trivial problem. Looking at someone's signature can be extremely difficult, actually. But these machines recognise the dynamics of the act of drawing the signature.

Grosch They recognise forces and directions rather than positions.

Shelness Here's an interesting way of protecting information: There are cases where you have to know statistics but don't have to know specifics, indeed want to protect specifics. We recently did this with a 1951 British census which we put on tape. The technique is to insert random noise into the record. The random noise makes absolutely no difference to the statistical conclusion you'll come to. But you can no longer trust the individual record because the chances are very high that the record lies about specifics.

Beitz I had to do a profile on a large population sample. They gave me the data in a mixed-up state. They put all the values for one item in one bucket, and all the values for another item in another bucket, and so on. There was no way I could correlate this. I had a lot of collective data, but couldn't even find out a single individual's whole name.

The Difficulty in Properly Quantifying Values

Flynn If there is one thing wrong with engineering, when that is used as a model, it is that engineers have the tendency to quantify the quantifiable. There is more to life than that. Personally, I love statistics. I love them especially when I can play number games with them. What terrifies me about it is when you go overboard quantifying the quantifiable, and you forget your goals. There are questions that need be asked about life-cycle cost, about things that are genuinely imponderable, or at least not easily quantifiable. For example, what is the life-cycle cost of allowing the system to use wrap-around addressing resulting in the referencing of unallocated areas? Or simply of referencing an unallocated or uninitialized variable? What is the cost of that? It is very difficult to answer this kind of question. You can benchmark the system, and you buy it on the premise that it runs four times as fast as another system. Or you can buy it on the premise that its associated software and programming tools allow you to write your own programs quicker. But what impact do hidden properties of the system have on the actual cost? Now if you insist on a numerical answer to this, then I'm in trouble because I cannot quantify it. But you can be sure that I shall assign some value to it, if only in the sense of being aware of taking a risk.

There are whole series of questions that you can ask yourself. For example if you want to reference an element in an array and you use a floating point variable to supply the index value, then you must correct the value into an integer. And I wonder what the life-cycle costs of that are. You can do benchmarking and performance measurement and still have no answer. The real question that remains answer-less is whether the system as a whole, or its underlying framework, are supportive of the system's goal. And I distinguish between cost and the normal measures of efficiency. You can optimise in terms of speed or memory useage, but in doing so you may have increased the actual cost. And ultimately *cost* (or the budget) is the only allowable measure. We don't really know how to do it.

Look at computer purchase agreements. Look at them from the purchaser's point of view. Look how hardware, software, performance, delivery date etc. are bundled together. This may give you a new perspective on things. And I'm worried that our modern numerology, the habit of quantifying the quantifiable, perhaps is keeping us from asking these kinds of questions. Efficiency is not a virtue if achieved without consideration of the context in which it has to exist. We must consider **total life-cycle costs**. How would you design a system, both hardware and software, if your goal was to minimize the cost of developing and maintaining correct applications programs?

Spier We do tend to lose sight of the issue of cost. But it is a real issue, and somebody made good money by recognising it. Somebody capitalized on the conjunction of two facts. First fact: There are hundreds of installations running old equipment such as GE400's, RCA501's, Honeywell 200's and you name it. These places have an enormous investment in software and in all of its related aspects: forms, procedures, trained keypunchers and programmers and operators and clerks and managers, and God knows what else. That equipment is no longer cost-effective, or even maintainable, and to replace it with other modern equipment could be a disasterously expensive undertaking. Second fact: Modern technology provides us with inexpensive, relatively high-performance, microcodeable hardware. So now there is a company that manufactures new inexpensive versions of those machines, plug and software compatible. The machines are not only inexpensive, compared to the replaced relics' original cost, but also much faster. This company may never become a second IBM, but they sure have a captive — and probably thankful — market.

Now this case history should make us stop and think. Do we always need radically new machines? Under what circumstances do the new Machine's virtues pale in comparison to the established software investment? At what point should satisfactorially operating software be inviolable?

Shelness What seems to be seen, from the evolution dynamics of large programs and the life-cycle cost, is that it does not matter how nice the architecture is. It does not matter what language you write it in. It does not matter how well-structured it is. You get into exactly the same problems. Take a *DEC* system written in assembly language. Take a *Burroughs* system written in beautifully structured *Algol*. Whichever way you do it, you cannot predict the way in which the system is going to be used in the future. And what you wind up getting is a clash between the system's structure and its function. And the only solution really is to start all over from scratch. All those nice structuring methods have a usefulness that goes only so far.

Lauesen I don't know how it is in the States, but in Denmark customers don't just choose computers by comparing the price. What seems to be said here is that we ought to develop other comparison techniques besides the performance benchmark to allow us to make wise and informed choices.

Flynn Yes, we need comparisons other than a performance benchmark or purchase price. Diesel trucks cost more than gasoline powered trucks in the U.S. However, diesel trucks are better engineered, last longer, and cost less to operate in the long term. Is there an architecture or a software system about which we can say the same thing? Do we design for the purchase benchmark or for the lifecycle benchmark?

The Goal of Software Engineering

Flynn I've been intrigued by something that has been discussed here, on and off. Namely the nature of mathematics and of engineering. My students asked me what engineering was, and I had a definition that I found in a dictionary: *engineering is the use of science in order to achieve a goal, such as: improve the quality of life, maximise profit, etc.*. I thought about this in relation to the proof of programs. I personally don't mind people proving programs correct. I don't mind people investigating the complexity classes of Turing machines. I enjoy it. It is reasonably elegant, it's a lot of fun. But I also am in the business of buying software. When I do software commercially, I do not want anybody playing with that stuff while he's working for me — let him have his fun on his own time.

If somebody wants to go out and prove 150 programs correct, and if he gains some magnificent insight because he notices a theme emerging from the way he's proving them, and if as a result of this insight he develops a better way of doing something — a way that will save me a lot of money

— *then* I'll lend a sympathetic ear.

But let us assume that he is not having such an insight and does it just for fun. He should be entitled to do that, just as a colleague of mine who works in homology theory goes out and writes papers on homology theory. I don't know what homology theory is, but he is an awfully nice guy, and he has no goal! He has no goal, because mathematics does not need a goal!

I remember one day I was reading in a library something touching on geometry. And I went to a dear friend of mine who does geometry and told him *"Why don't you read this, this looks like fun?"* So he went home, and he got an idea, and he wrote a lovely paper, and he sent it off, and it felt good to him. It didn't have to *mean* anything; it felt good. Turns out someone in Argentina thought it was terrific. I think it would be terrible if mathematicians ever were goal-driven. I think it wonderful that a mathematician can go out and do whatever he wants, and that a body of knowledge emerges, and if it turns out that it actually happens to be relevant, that's wonderful. Calculus is nice. Also, you can't really get by without calculus. It is less so with homology theory.

There is a fundamental difference between the questions of software engineering on one hand, and computer science on the other. It is *OK* to have an abstract view of the computing process within the computer science *milieu*, but that is very, very, very different from actually producing software. Nobody has ever given me, as a mathematician, a theorem to prove, saying *"for $10,000, within 3 weeks, prove it!"* But we accept being offered money by someone who says *"I want this coded and running 3 weeks from now"*. And he wants it tested, and specified, and documented, and that is a very different reality.

After years of experience in making software, I have recently had the novel experience of buying it. It changes your perspective enormously.

I asked myself what the goal of software engineering is. As opposed to computer science, as opposed to mathematics, the goal of software engineering is to maximise the software value you can get for a certain amount of money; to push down the software's life-cycle costs.

You can debate all you want about the relative merits of robustness or elegance but when I buy software, elegance does not matter — I'm willing to have it as long as it does not cost too much money. Concerning the relative merits: having ten pounds of elegance and one pound of robustness may be much more expensive than having ten pounds of robustness and one pound of elegance. Now if you can give me ten pounds of robustness and ten pounds of elegance at no extra charge, then you'll also satisfy the computer scientist in me. But when I buy software, I am predominantly a businessman who worries about cost effectiveness. All this comes directly from the original goal which is: I have this much money budgetted and available to buy a minimum of good-quality function; if I can get more function for the money — great; if I can get the needed function for less — also great; but let nobody here fool himself into believing that by making it elegant (whatever that is!) and more expensive he could still obtain my business. The software has to come in on budget. And the basic goal, vulgar as it is — is still measured in terms of money, spelled *M.O.N.E.Y.*

Gram I am left with a very uneasy feeling. You may be absolutely right, that *M.O.N.E.Y.* is the ultimate goal of everything . . .

Flynn No. Of software engineering.

Gram . . . of software engineering. But it cannot be right that we should support that view and live up to it.

Stoy That money is the ultimate goal of software engineering is a rather simple-minded goal that Americans seem to perceive.

Grosch May I ask why software engineers switch jobs so frequently in the UK?

Beitz For the money.

Meineche-Schmidt You are saying that computer science is distinct from software engineering, and that software engineering is recognised by its goal of saving money while computer science is not recognised that way.

Derrett Money should not be the ultimate goal of software engineering. You may pay me money to make your software. But as an engineer I have other responsibilities besides giving you your software for your money. There are other, and I think more important, things. For example, I also have to think of the people who will use the product, which is not necessarily you; you are only paying for it.

Flynn Obviously. I understand that. It is clear that if I have a system that doesn't function well, that causes problems to its users, then I have not got my money's worth. Let me say that it is staying within budget that is the goal, including life-cycle costs; not money *per se*. Money is just the measure.

Derrett Money is not the only measure.

Beitz All other things you deem important are ultimately translated into money.

Grosch I think that the revulsion expressed in this room against this idea is quite justified. If you look at our total western culture and see where it goes, with our most successful people or top executives dying of heart failure before they are forty years old, then you realise that this is a morally wrong goal. But you'd have to substitute something for it in a culture that is deeply devoted to it. And I suggest that we could, just a little bit, steer away from the money thing by promoting professionalism and ethics and that sort of thing.

In principle I think that if you are a paid software engineer and you are told that you are to produce a secure system, and you judge by your professional estimate that the thing the customer wants could never be secure, you have a certain duty to society to say *"this doesn't work"* and if necessary to blow the whistle on it. In the future you could hopefully get support from your professional society.

But this is a fringe consideration. I'm afraid that Flynn is right in 99% of the cases.

Flynn I agree with you totally. I hope that nobody believed me to have said that money is the goal of everything. It cannot replace professionalism or ethics. But the professional's inner self-respect and professional growth must evolve and exist within some constraining framework, which is economical. Think of this as a constrained optimisation problem.

My personal goal in life is satisfaction, a part of which comes from professionalism, another part from solving complex problems. If I am in charge of a large project and I have an employee who is going way over budget because he is trying to be elegant, with an emphasis of the word *trying*, then I have a problem. That problem is an ethical one for me. I can't allow that person to waste the purchaser's money or to jeopardize the project and the employment of the other team

members. I can't do that any more than I can let a product go out the door which is not robust. Minimizing lifecycle costs is different from maximizing profits, even though they can both be measured in the same units.

Spier We should divorce the recognition of money-as-a-measure from our moral objection to it. Why, for example, should you find it moral and scientific to study the questions of machine useage optimisation, when in most cases all you need is enough money to go to IBM and buy a bigger machine. All these optimising techniques are solutions to non-problems if you do not admit to the validity of financial constraints — such as the inability to buy more and faster equipment.

Grosch Many of these objections are concrete. When you talk of deep social things, the richness and happiness of life, you talk of things that are important but which still translate into money.

We always make value judgements. RCA's bid may be the lowest, but if, in your judgement, RCA will no longer be in the business in three years' time, then you don't buy it. Now I may make another judgement. Namely, if a vendor offers me $10,000 under the table to select his machine, then I'm in an ethical fix; but it is still a money judgement — how much is my honesty worth?

Spier Society is a function of economics.

Shelness Companies, at least most companies, don't run for the gratificaion of their employees. They run in an economic system — be it Western, African or East European, or any system — whose only measure is money.

Derrett But we in the universities shouldn't, and needn't, have our activities motivated by money.

Grosch The reason that Whirlwind was turned into a digital computer was that its makers couldn't get enough money to keep building it as an analog computer. So in a sense digital computing was motivated by money.

SESSION G

Hardware for Software

Which software problems are self-inflicted non-problems, in the sense that if we had better machines then the problem would not exist? What would such machines look like, and what benefits are expected from their availability? What new software problems might arise from the availability of such machines? How can system costs be reduced by shifting more functionality onto the hardware level?

A System Based on Functional Programming

Robert S. Barton

Presentation by Bob Barton

On Modular Machines

Barton The systems ideas that I am about to discuss are motivated by the remarkable new LSI technology. We still hardly know what to do with it. I had been considering what kind of machine would evolve if the people who developed the *LEGO* toy — quite a remarkable application of the building block concept — developed similar approaches in the computer industry. Because in the space of a single *LEGO* block you could have enough logic and storage to do quite a bit. And in that space you could also have a suitable packaging scheme so that the blocks could be plugged together end-to-end.

Well, that's a common kind of thought people have been having for many years. It is now beginning to look practical in an engineering and manufacturing sense, but otherwise we don't have much to go on.

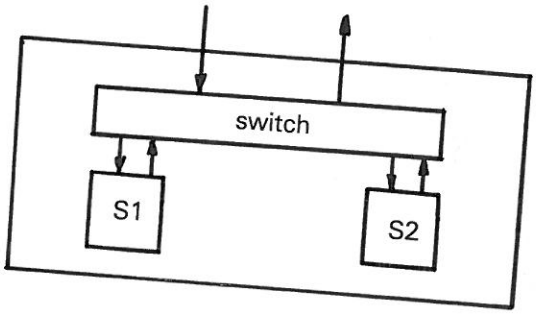
Knowing something of what had been done, and having lived through many many machine design efforts which were hamstrung to some degree by compatibility, I decided this problem was worth thinking about but had to be put into a new setting. We had to get away from language compatibility. Not just machine-level compatibility, but language at any level. So we did that, and we started at different places, almost disjointly, and it is only recently that these ideas have converged to a point of view that looks worth pursuing.

I think that I can tell you about a few of the starting points, then come up with a few remarks on how I now can relate it to some of the other things going on.

The question posed is what the problem would be in building a machine out of many identical modules. What I mean by *module* so far is an identifiable piece — maybe one of these *LEGO* blocks. It is almost obvious that even before we know what is in the module, the real problem could be in the switching between them. The first thing you might think of is that it would be nice to have many processors, and have every processor be able to get at every storage module. If you do that then the switching of course grows proportionally to the square of the number of components. This is not very practical if you really want to build big machines out of *LEGO* blocks. It would be nice to have some sort of a recursive definition of a machine, so that after we connected two of them they could behave as one.

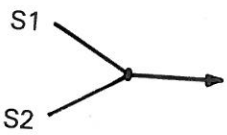
Here are two systems which we shall assume identical in characteristics:

and we only ask that we be able to interconnect them in some way so that we can regard the pair as a system of the same sort. Each one of these systems has at least one input and one output, and so the connected pair as a whole must have an input and an output.



We have to add something that will serve as a switch. We can say a lot about the characteristics. For instance, if we're coming from the outside we have to select the one we are going to.

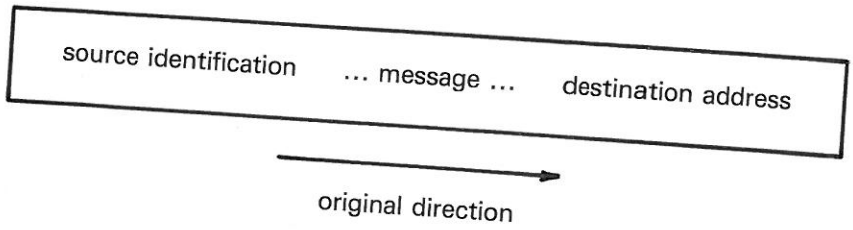
If we're going to the outside from the inside, and the two are autonomous we have to resolve possible conflicting requests for the path.



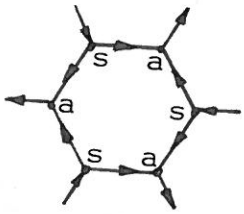
Since it would not be fair to put the two together and forbid communication between them, there should be symmetry within the switch. Now this is only one of many such parts, and so if we really allow the system to get arbitrarily large and don't know how far apart the parts are going to be, the whole thing should work asynchronously. The switching will have to be asynchronous. You can't merely send a message — you have to send out a request, and get back a reply. So just with these considerations alone, we get the beginnings of a specification for a way to interconnect parts.

Let us first consider the tree of arbiters. Any terminal may initiate a message which finds its way through this tree, with arbitration at the nodes. It eventually gets service from the shared device, and a return path along a structurally identical net of selectors is taken back to the terminal.

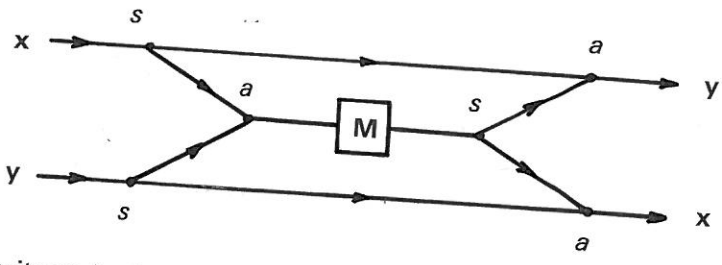
What are the transmission assumptions? We send a message which has as its head the *destination* address. We make this into a binary string so that whenever the message reaches a selector a bit is peeled off, a switching decision is made, and the message passes through. On the other end of the message we need a source identification. With considerations of symmetry and simplicity in mind, we want the source identification to specify a path back to the sender, so whenever the message goes through an arbiter it tacks a bit on to the end which will serve to get the return message back to the sender. Thus a message is a self-delimiting stream of bits, the front part of which is a destination address for a selector network and the tail of which will be the address for the return path.



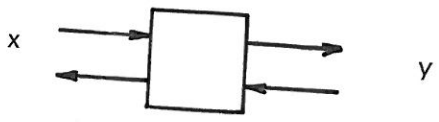
Returning to the original problem, we are interested in building up a system, recursively, from identical *LEGO* parts. How can we use selectors and arbiters to build it? The thing has to have three-way symmetry. Here is a hexagon of alternating selectors and arbiters:



There are different ways in which to have concurrent paths. We can associate opposite pairs of arrows. We can turn two arrows inwards, and connect them to a module; we don't know what it does except that it has an input and an output:

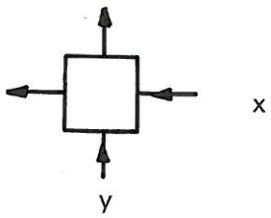


If we now call opposite nodes by names *x* and *y*, then we can re-draw the structure



and maybe that could be our *LEGO* block, provided that we can do any computation that way.

We might rearrange for connections in two "dimensions", represented upwards and sideways, and maybe that will allow us to do something useful.



There are many patterns possible, but the two we found most useful were the selector-arbiter pair for controlling a shared device, and the boxes which we can make into a linear string as shown above.

We have been experimenting with these forms. The devices must be asynchronous so that a system can be as large as we wish.

We haven't said how many *LEGO* blocks we might want to use, but we had in mind streaming algorithms which we used in the old days for punched cards and which have been discussed more recently by Burge. We can imagine special-purpose machines containing many thousands of modules. Each module would be a single simple integrated circuit part — allowing us to reach economic volumes of production.

What's in a module? We naturally have some prejudices — we prefer stack machines to the other kind and definable-field machines to machines that impose word structure on the machine language. Characters as primitive data elements are all right if the characters themselves aren't constricting somehow. I believe that there are useful ideas in data-driven sequencing, though it can be overdone.

I decided a long time ago that one way to find out how ugly a language is is to visualise the machine that executes that language. I'm not precluding a translation to a simpler form; still, what kind of machine do you want after the translation?

We had an idea about a year ago, as to how we would like to build a machine. The whole organisation of it just came in one big thought.

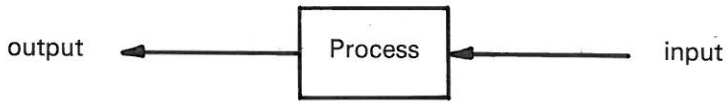
We had a big problem — could anybody program it? The form we imagine using is full of brackets of various types. I know a bit about existing programming languages, including some of the odder ones, and I was concerned whether there was any style of language appropriate to our machine as conceived.

I found out recently that there is, but I found it out in rather a roundabout way. So let us drop what we have looked at up till now and I'll give you the other part of the story.

One thing will have to be done before we can take advantage of large-scale integrated circuits: the people who design applications for machines will have to begin to think about concurrency, and the possibilities for concurrency. Most people, when thinking of concurrency, do it with a view to getting more speed from the hardware. Our first consideration regarding concurrency is how to allocate resources, particularly storage.

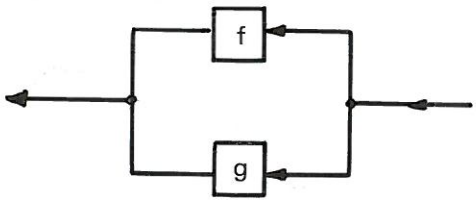
We started off with a simple line of thought: data-processing applications.

Programmers do the same things that they have been doing for years and years and years — the languages used encourage that. Language is what we work with, and it influences our thoughts. What if we went back to graphical descriptions? Let us imagine that an applications designer would work with something we might call a concurrency diagram. We start with a box representing the notion of a process having an input and an output.



Probably it isn't meaningful to talk about an application unless we can at least name it, even if we are otherwise vague about it. We have some confidence that it can be done, mostly because of experience in the real world. We have some idea what it is we want to transform in the process, and what we want to get out.

Now, it should be possible to use the notion of concurrency as the basis for functional decomposition. From the other end, it could be used as a basis for synthesis. For example, this process might break up into two parallel parts *f* and *g*. We mean by this diagram that *f* and *g* are independent:

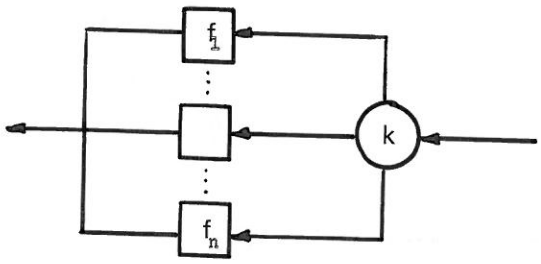


and the same input value goes to both of them. There is no reason to restrict the number of boxes to 2 — it could be any number.

Another decomposition is the series arrangement, with the output of f dependent on the output of g:

Here we have a simple concept (it has to be simple because it has to be understood by non-mathematicians): we do things in stages. In the early days, when people wrote applications for small machines, they had to think of things stage-wise: tape in; do something; tape out; and tape in again for the next process after you put in a new program. So this is a natural way to think. What has it got to do with concurrency? Typically we have a series of cases. When the first case reaches a process *f*, then the predecessor process *g* is ready to handle the next case. This is a pipelining arrangement.

We also have the idea of choice. A process *k* makes a decision as to which of the processes f_i to use,

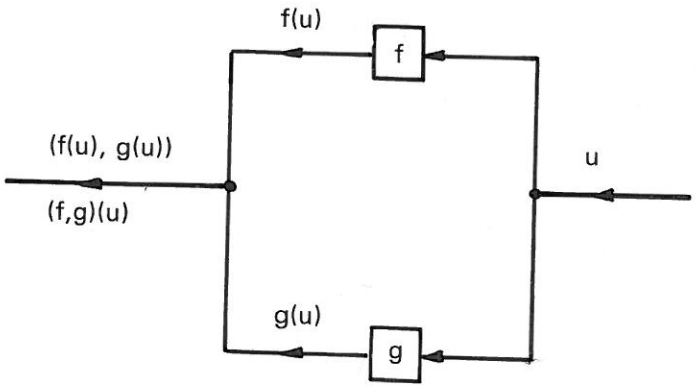


where the processes f_i are mutually exclusive.

Another kind of thing to rear its head and present difficulties, but which is very important, is the notion of a process that is repeated a sufficient number of times.

One possible way of planning an application would be decomposition on the basis of concurrency. It's not too clear what we mean by concurrency in all these cases. I made some experiments to see if people actually could organise their work in this way, but did not get very far; I don't have any captive students.

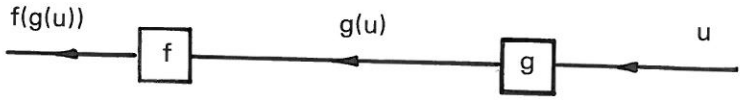
Later I saw that a natural kind of notation exists for these things. Originally I thought we would have to develop a new notation, but why make something new if you don't need it? Let us take the parallel construct:



The ordered pair (f(u), g(u)) which results from the combination may be written as

(f, g) (u)

and we call this **construction**. The series arrangement

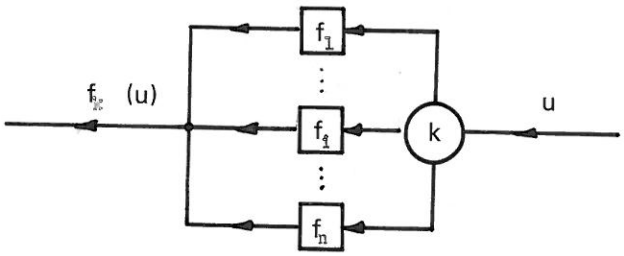


is called **composition**. The output is f(g(u)), and we might write this with a circle standing for composition:

$$f \circ g\ (u)$$

In fact *u* isn't doing anything for us in these forms so there really isn't much point in putting *u* there at all. When I talked to people, who managed programs, about these things, they said one ought to name the inputs and the outputs. But there is a sense in which you don't need to write them down.

We can write the choice



as f_k (u) or just f_k .

Iteration can be viewed as functional exponentiation. If *n* is an integer constant, then f^n is the composition of *n* instances of *f*:

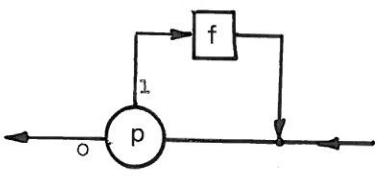
f^n can be defined by

$$f^n = f^{n-1} \circ f$$

where we note that

$$f^1 = f$$
$$f^0 = i \qquad \text{(the identity function)}$$

If the exponentiation depends upon the result of each stage of the iteration, let *p* be a predicate (**1** represents true and **0** false) and write $f^p *$. The * (after Kleene) represents a composition of as many instances of f^p as are required to make $p = 0$.



Now we have 4 ways of breaking down processes into components. Where do we terminate this decomposition? We thought of stopping when we could write the components as programs in some language. Of course you need some additional primitives, which have not been discussed here, but you can break up applications into very small pieces that way.

Recently we started to look at the literature on functional programming. Many of the authors expressed doubts about replacing "practical" languages having go-to's and assignment statements. Backus, however, takes a very positive attitude, and we agree.

In summary, we plan a machine composed of identical parts, the language for which will be a functional notation. We claim that if one cannot program an application in this fashion then, in a deep sense, one does not understand what is to be done.

Discussion following Barton's Talk

Grosch How complex is one of your modules, Bob?

Barton I think that one of our modules is practical for today's level of integration. The order of complexity of today's microprocessor might do it.

Grosch So it is not out of technological reach.

Barton No.

As we stand right now, we know we can build interesting machines. We don't know how widely applicable they are.

Grosch What you have going for you is that we are now at the technological stage where you can get quite a high yield of chips regardless of their internal complexity.

Barton If you have a special application then you could get economical parts this way, and the overall assembly would be uniform.

Grosch You could even go to a higher level of integration for your physical module, using microprogramming to give it a specific function.

Barton Yes. In fact, technology is such an imperative here that I think it may very well bring about change where we thought change unlikely. I also think the Japanese are going to force change. They will force down the price of conventional computing so much, that even IBM will have to change in order to compete.

Spier Your talk brings to mind Bell's Register Transfer Modules (RTM), collectively known as the *PDP16*. Would you care to comment on this?

Barton No, these are not related.

Clark RTM's are traditional in concept. Our [*i.e. Clark and Barton's*] modules have an entirely different structure of processor, with an entirely different form of communication between them. Ours are asynchronous. Bell has asynchronous connection to a bus.

Lauesen How do you connect these modules?

Barton We are primarily interested in uniform connection. One may think of data as having an access function associated with it. We are also interested in programs that can be restarted anywhere, provided the breakage is along functional lines. We get that for free.

Shelness I am reminded of CCD analog devices. Your proposal has a very similar strung-out structure.

Barton Some very simple ideas can become timely simply because of the properties of new components.

Shelness We have been looking at this building block approach for compilers. I can see how to make those, but it does not seem very satisfying. I could sum up my feeling by saying that it would take three to do the work of one.

Barton Ever since I started in this field, people have talked of *"efficiency"* without bothering to define what they meant. I stopped listening after a while.

Barton Backus mentions two extremes. One is a complex language with a sparse subroutine library; the other is a very simple language with a rich program library. He prefers a simple fixed framework and a great variety of interchangeable parts. Contrast this with having a complex framework where the skillful programmer will always see another way of doing something he should long ago have put into a library of functions, whose details can be forgotten.

Beitz Both Pratten's talk of the complexity of large systems and Shelness's remark of trying to put together large processes are the antithesis of using a library of functions. There you need three modules to do the work of one, and here you have synergy where the whole is greater than the sum of the parts.

Grosch [*to Barton*] Your notation is mathematical. What sort of people do you think will be able to use this notation naturally?

Barton Anyone who uses mathematics as a tool. Engineers and so on. They may have to change their notation but the ideas are the same.

Grosch The trouble is that they all know about our kind of computers now.

Barton I know.

Non-mathematical programmers can structure programs in boxes in the way I showed — my decompositions show connections between modules.

Barton In building our machine, we'll pay attention to the requirements for proof.

There is an interesting controversy going on between Perlis and Dijkstra right now and I find myself both agreeing and disagreeing with both of them. How can people prove really big programs? Especially when considered realistically they are not at all static objects, and are changed continually. But why give up the notion of a much higher level of confidence just because absolute correctness cannot be proved?

Computer Architecture for Correct Programming

K. Berkling

COMPUTER ARCHITECTURE FOR CORRECT PROGRAMMING

K. Berkling
Gesellschaft fuer Mathematik und Datenverarbeitung mbH Bonn
Postfach 1240, Schloss Birlinghoven
D-5205 St. Augustin 1

A computer architecture which is centered around the transformation of expressions facilitates the generation of programs from definitions, axioms, and theorems. It is shown that some of the well known methods to construct correct programs are directly available on this novel computer. Due to its novel architecture the need for support by large software systems and entailing systems size is diminished or has disappeared. Using this novel computer employing a lambda reduction language as machine language it is demonstrated how suitably chosen invariants get transformed into an executable program which is correct by construction. The various types of information which get incorporated into the result, as well as those parts of the original expression denoting the invariant which get expelled because they are constant, are discussed. It is demonstrated why the above mentioned novel computer architecture frees the user from being concerned with inessential problems, while on the other hand the use of higher level programming languages introduces additional problems due to the semantics of these languages. They necessitate considerations alien to solving the original problem. Crossreferences to other disciplines are pointed out which employ similar concepts.

Introduction

The issue of program correctness has drawn substantial attention since the so called software crisis became evident. The ways and means to alleviate the situation, consist of a methodology to construct correct programs, to prove the correctness of existing programs, and to make the process of program generation transparent. For obvious reasons, the intentions of a user cannot be used for these purposes as such. These intentions need to be represented in a formal way such that they may become part of mechanized proof or test procedures. It may be quite a formidable task for a user to formulate his intentions in an appropriate manner, a task at least as difficult as writing the program itself. However, there seems to be no way around having a problem specification at least twofold, first in a form suitable for someone to convince himself or another human being about its correctness, and second in the form of a program, that is an efficiently machine processable object.

Transitions between these two forms have been demonstrated in the literature.[3 11 12 14 18] Considerable software support is required on conventional computers to conduct such transitions.

A novel computer architecture is demonstrated which facilitates the transformation of expressions and provides mechanical support for program generation and transformation in areas where conventional computers and programming languages do not. This novel computer architecture leads to computers which are particularly well suited to formulate invariants in terms of their machine language, and to perform the transformations of such invariants into a machine processable form, which is again formulated in terms of the machine language of these novel computers. The formulation of invariants is by no means easy, especially if non-numerical cases are considered. The point being made is not that everything becomes easy and all problems are solved if one uses the method of invariants, the point being made is that part of the methodologies developed to achieve program correctness becomes available on a relatively small system based on the novel computer architecture without the need for large software systems.

A few simple examples are worked out to show that already small problems lead to a large degree of complexity.

The Novel Computer Architecture

A full and complete description of the architecture behind the subject of this paper is given in.[1 9 10] By using the slightly sugared machine language of this computer we will try to make transparent as much as needed from the computer architecture.

The storage system of the reduction language machine comprises several stacks each one byte wide connected like spokes to a hub as a control unit. The simulator and the hardware model employ seven stacks with respect to the implemented set of reduction rules. The control unit manipulates tree structures which are stored as pre-order linear representations of trees in these stacks. A scan procedure is designed such that a "tree-walk" is emulated.

Trees consist of constructors as nodes and atoms as leaves. Atoms may be characterstrings, digitstrings, truth values, and some special characters, for example, "]" which serves as the NIL-atom. Constructors make a new tree from two subtrees. Several types are used. There is a DOT constructor "." to make data structures. The DOT constructor does not assign special roles to its subtrees. A group of "apply" constructors serve to designate their subtrees as function and argument:

apply <function tree> to <argument tree>
apply to <argument tree> <function tree>

Note that in the second line argument and function have interchanged their position.

There is a function making constructor "λ":

function with parameter
<characterstring> <function body>

The control unit recognises a function

application by a constructor pair in the corresponding predecessor-successor positions while scanning a tree, and performs it according to the beta-reduction rule of the lambda calculus [4 5] by literal substitution. Some atoms denote primitive implemented functions, like arithmetic, boolean connectives, constant and identity functions. The reduction language machine scans an expression and performs operations called reductions until this expression is free of any instances of reduction rules, that is free of any local combination of constructors and atoms which is designated to cause a transformation of the involved subtrees. Such expressions are called constant expressions.

Application of a lambda expression to its argument (known as beta-reduction) may be suspended until the function tree or the argument tree has been reduced to a constant by preceding that tree with the keyword reduced in order to achieve certain evaluation sequences like call by value or call by name. Thus, we have 6 different apply constructors and we use the following abbreviations:

← f a	apply f to a
. f a	apply f to reduced a
▵ f a	apply reduced f to a
→ a f	apply to a f
: a f	apply to reduced a f
▿ a f	apply to a reduced f

To improve readability, the DOT constructor may be given to the input also as "[" or "," such that the input [A,B] is converted to •A•B]. Arithmetic operators are "Curry'd" on machine level, but fully parenthesized input is accepted, such that

$$(a+b) \equiv :a.+b$$

and

$$(a+b) \equiv \bullet a \bullet +b$$

where the latter notation serves to denote values which are of interest, but there is no need to compute them. Similarly, • f e denotes a value obtained by applying function f to argument e, but the value is not computed.

An important extension of the beta-reduction rule comprises the appearance of an expression in the parameter part of a function.[8 18] Function application is in this case conditional on a match or no-match of the parameter expression with the argument expression. Variables in the parameter expression match with subexpressions in corresponding positions of the argument expression, which is partitioned into these subexpressions if the match is complete. These subexpressions then get substituted into the function body separately according to the matched variables. In the case of a no-match, the combination of function and argument remains unaltered. In order to provide an alternate path in this case we use a special list of functions:

$$\subset f_1 \subset f_2 \ldots \subset f_n \ ]$$

If this list is applied to an argument, a match is tried for function f_1, if it does not succeed, function f_2 is considered, a.s.o. This special list works as identity function such that only the unaltered argument remains as a result, if no function succeeds.

Predicates reduce to truth values "true" and "false" which are one-character atoms. They work, however, as functions $\lambda u\, \lambda v\, u$ and $\lambda u\, \lambda v\, v$, respectively. Thus a conditional expression can be formed from a predicate expression p, two other expressions a and b, and two application constructors ▵:

▵▵ p a b

According to the semantics of the applicator ▵ defined earlier, this expression will reduce to a or b depending on the result "true" or "false", respectively, obtained by reducing the predicate expression p.

The implementation of recursion in conventional systems is accomplished by allocating an activation record for each call of the recurring function or procedure. In a reduction language machine a copy of the recurring function expression is the activation record. A copy of the function expression gets literally substituted for each occurrence of the function name. The corresponding reduction rule is akin to the LABEL operator in LISP and implements the α-rule in Dana Scott's logic for computable functions (LCF).[15] To associate a function expression f to a name v we construct the expression:

α v f

or sugared:

recursive function f with name v

Each free occurrence of the name v in f is replaced by a copy of α v f:

α v f ==> ← λ v f α v f

Iteration does not constitute an essential difference. In this case a copy of the loop-body is substituted.

All input is preorder linearized treestructure as a sequence of bytes. Except for desugaring and code transliteration there is no compiler, assembler or an interpreter in the conventional sense.

This should suffice for an introduction. Further detail will become clear during the discussion of the examples.

Examples

We will present two examples, which do not in any way present new solutions or new information. Their only purpose is to exercise a methodology to construct programs from a formulation of invariants. The significant result is, that the reduction language machine supports interactively this construction work by providing expression structure and mechanical transformation which are usually available only in larger software systems.

EXAMPLE I. Let the problem be the computation of the binominal coefficient

$$\binom{n}{k} = \frac{n!}{k!\ (n-k)!}$$

Let us first consider the theorem

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

as a candidate for an invariant. The function itself may serve as an invariant. This leads in general to very elegant, short, and transparent expressions, for example:

```
∝ B λ N λ K ΔΔ ((N=0) v (N=K)) 1
     (..B (N-1) (K-1) + ..B (N-1) K)
```

Application of the alpha-reduction rule leads to a binary call tree, which is not very economical, since the function is computed several times for the same arguments. Note, that the function B is never really computed, instead its functional properties are used to perform substitutions, which are performed by the reduction language machine while the expression above remains invariant. The invariant expression becomes larger and larger because of the order terms are associated. Arithmetic reduction can take place only after all terms up to the one resulting from the true clause have been generated.

In order to find a more economical solution we start with the identity

$$\frac{n!}{k!\ (n-k)!} = \frac{n\ (n-1)\ \ldots\ (n-k+1)}{1\quad 2\quad \ldots\quad k}\ .$$

To compute the value of this expression one has obviously to take factor after factor from this expression and multiply it with a partial result. There are several ways to remove factors in an orderly way from the expression. Let a denote the operation of taking a factor from the left of the numerator, b from the right of the numerator, respectively, and let c and d denote the corresponding operations for the denominator. The following is a selection of nine basic patterns to design a solution:

```
F1   a* c*
F2   a* d*
F3   b* c*
F4   b* d*
G1   (ac)*
G2   (ad)*
G3   (bc)*
G4   (bd)*
H    (abcd)*
```

The star denotes iteration. We reject the F plans because they would require two iteration control structures. We reject plan H because there are problems of termination due to an even or odd number of factors.

Let us first try plan G1. We define a function G1 of three parameters X, Y, M such that

$$\bullet\bullet\bullet\ G1\ X\ Y\ M = \frac{\overbrace{X\,(X-1)\ \ldots\ (N-K+1)}^{M\ \text{factors}}}{Y\,(Y+1)\ \ldots\ K}$$

The following equation holds since we have accounted for the factors taken away from G1:

```
••• G1 X Y M
      = {••• G1 (X-1) (Y+1) (M-1) * {X/Y}}
```

Our plan is to accumulate these factors into a partial result. Thus we multiply both sides of this equation by R.

```
{••• G1 X Y M * R}
     = {••• G1 (X-1) (Y+1) (M-1) * ((R*X)/Y)}
```

Note that the associative law has been applied to make certain that division leads to an integer. The parentheses () indicate computations which will be performed. If integers are substituted for X, Y, M, and R, and the computations are executed (except the topmost multiply) we obtain on both sides of the equation a datastructure with four integers. Their values have been systematically altered, but the relationship among them expressed by the datastructure has remained invariant.

Such datastructures denoting invariants are machine language objects in the reduction language machine. They may be manipulated interactively in the machine to facilitate program construction.

The initial state of the invariant is:

```
{••• G1 N 1 K * 1}
```

The final state of the invariant is:

$$\{\bullet\bullet\bullet\ G1\ (N-K)\ (K+1)\ 0\ *\ \binom{n}{k}\}$$

Moreover, the reduction language machine enables us to formulate the transformation of the invariant as a function. The purpose of this function is to recognize a datastructure combining four integers and produce another one with four systematically altered integers.

```
λ{ •••G1 X Y M * R }
      { •••G1(X-1) (Y+1) (M-1) * ((R*X)/Y)}
```

The expression following the lambda is a bound "variable", that is an expression, which will be matched against an argument structure. An application takes place, if the argument agrees. The variables G1, X, Y, M, and R are associated to corresponding substructures of the argument. In the body of the function one may refer to these substructures by mentioning these variables.

For example, the function

```
λ{ ••• G1 X Y 0 * R } R
```

gets applied if and only if M=0 and returns R. The argument 0 for M means, that all factors have been accounted for in the R-component of the datastructure. This R-component is therefore the result. Thus, we may use this to terminate the transformation of the invariant.

We may now assume that the transformation function, the termination function and the initial state of the invariant are stored in

the reduction language machine. We assemble the transformation function preceded by the termination function into a special list denoted conditional application, label it by F, and apply it to the initial state of the invariant.

```
: {••• G1 N 1 K * 1}
  ∝F
     ⊂λ{••• G1 X Y 0 * R} R
     ⊂λ{••• G1 X Y M * R}
        .F {••• G1 (X-1) (Y+1) (M-1) ((R*X)/Y)}
     ]
```

The labelled object ⊂ f ⊂ g] gets applied to the result of the transformation function, that is, the transformation takes place until the termination function gets applied.

The same datastructure denoting the application of G1 appears in various parts of the above computational structure but with different components. It is in these components where the actual computations are performed. This computational structure together with the definition of G1 should be sufficient to convince another programmer about its correctness. Moreover, it runs on the reduction language machine.

This computational structure contains highly redundant information about something which remains invariant. The result, however, depends only on those components which change. Thus, we may collect the changing components into the arguments of a four parameter function:

```
....∝F
     λX λY λM λR ΔΔ (M=0) R
      ....F(X-1) (Y+1) (M-1) ((R*X)/Y)
      N  1  K  1
```

Also, we replaced the conditional application by a simple test for M=0. An alternative set of initial values is N 1 (N-K) 1 for reasons of symmetry.

This is also a correct, executable program expression for the reduction language machine. All steps in obtaining this product have been justified by a valid argument. We have accounted for the fact that the arithmetic unit of the reduction language machine is not capable of handling a rational number representation.

Let us now consider a function G2 according to plan G2:

$$\bullet\bullet\bullet\ G2\ X\ Y\ M = \frac{X(X-1)\ \ldots\ (N-K+1)}{\underbrace{1\ \ 2\ \ \ldots\ (Y-1)\ Y}_{M\ \text{factors}}}$$

and the corresponding equation:

```
••• G2 X Y M
      = (••• G2 (X-1) (Y-1) (M-1) * (X/Y))
```

The removed factors cannot be accumulated into a partial result, since integer division would produce non-zero remainders. If we associate the terms differently, we postpone division as late as possible:

```
..G2 X Y = ((..G2 (X-1) (Y-1) *X)/Y)
```

We have already observed that the counting of factors can be taken over by parameter Y. Following similar arguments as in constructing plan G1 we obtain:

```
..∝F λX λY
      ΔΔ (Y=0) 1
      ((..F (X-1) (Y-1)*X)/Y)
      N K
```

This is a genuine recursive solution, because all multiplies and divides are stacked up before their computation begins.

Plan G3 corresponds to R. Strong's solution given in [6]. We define a function G3:

$$\bullet\bullet\bullet\ G3\ X\ Y\ M = \frac{N(N-1)\ \ldots\ (X+1)X}{\underbrace{Y(Y+1)\ \ldots\ K}_{M\ \text{factors}}}$$

and find the equation:

```
••• G3 X Y M
      = {••• G3 (X+1) (Y+1) (M-1) * (X/Y)}
```

Here we apply the same method as in plan G1 and finally obtain:

```
....∝F
     λX   λY  λM  λR ΔΔ (M=0) R
      ....F (X+1) (Y+1) (M-1) ((R*X)/Y)
     (N-K+1) 1   K   1
```

The initial values of the set of variables are according to arguments used similarly as before:

X	Y	M	R
(N-K+1)	1	K	1
(K+1)	1	(N-K)	1

where the second line indicates the symmetric case ((N-K) for K), and which is R. Strong's solution.

We are not following through the last possibility to define a function G4, since this would only lead again to the problem of the missing rational number representation arithmetic.

Although the different forms of solutions given here are all executable expressions for the reduction language machine, it is of interest to link these solutions to conventional higher level programming languages.

The semantics of the reduction language are such that the position of variables and arguments may be consistently permuted such that, for example, ↔ λX λY F 3 4 = ↔ λY λX F 4 3. If we translate the variable-argument associations into assignment statements in the same sequence as the machine can apply reduction rules we obtain one out of n! permutations. Unfortunately, due to the semantics of the assignment statement, most of these permutations, if not all, yield false programs. Only the substitutive architecture

clearly separates the old set and the new set of values. In the case of our solutions there exists at least one correct sequence. The program is:

```
FUNCTION G1 (N,K)
        X := N
        Y := 1
        M := K
        R := 1
F:      IF (M=0)    THEN RETURN R
        R := (R * X)/ Y
        X := X-1
        Y := Y+1
        M := M-1
        GO TO  F
```

We dispense with a discussion of leading or trailing decision, or fixing the decision for another permutation of the assingment statements. It is also not a matter of discussion that one could have written a procedure to circumvent the problematic semantics of the assignment statement, since to dispense with the assignment statement is not the objective of the higher level programming languages.

The process of compilation disguises even further the connection of the machine program to the mathematical definition. Before we discuss this process of alienation further let us have a look at another example.

EXAMPLE II. Let the problem be the computation of elements of the Fibonacci serie. We start out with the definition

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = F(1) = 1$$

A straightforward notation in the reduction language leads to a binary recursive call tree, where a function gets computed for the same argument over and over again.

We come by intuition to the invariant:

$$\{\bullet\ F\ K - \{\bullet\ F\{K-1\} + \bullet F\{K-2\}\}\}$$

which is equal to

$$\{\bullet\ F\ (K+1) - \{\bullet\ F\ K + \bullet\ F\ (K-1)\}\}$$

The transformation step is:

from $\{\bullet\ F\ K - \{X+Y\}\}$

to $\{\bullet\ F\ (K+1) - \{(X+Y)+X\}\}$

The initial state of the invariant is:

$$\{\bullet\ F\ 2 - \{1+1\}\}$$

Again, we do not intend to compute F or perform the operations in the braces. The reduction language machine enables us to formulate the transformation of the invariant as a function:

$$\lambda\{\bullet\ F\ K - \{X+Y\}\}\ \{\bullet\ F\ (K+1) - \{(X+Y)+X\}\}$$

The termination test function is

$$\lambda\{\bullet\ F\ "n" - \{X+Y\}\}\ (X+Y)$$

The argument "n" is a number, not a variable!

The complete function applied to the initial state of the invariant is:

$$.\alpha L \subset \lambda[\bullet\ F\ "n" - \{X+Y\}\}\ (X+Y)\ ] \subset \lambda[\bullet\ F\ K - \{X+Y\}\} .L\ \{\bullet\ F\ (K+1) - \{(X+Y)+X\}\}\ ]\ \{\bullet\ F\ 2 - \{1+1\}\}$$

By removing all constant ballast and replacing the conditional application by a simple test we obtain

$$\ldots\alpha L\ \lambda K\ \lambda X\ \lambda Y\ \Delta\Delta\ (K=N)\ (X+Y)$$
$$\ldots L\ (K+1)\ (X+Y)\ X\ 2\ 1\ 1$$

The expression may be prefixed by N to turn it into a function with the parameter N.

Discussion

Common to both examples is an approach pattern which begins with formulating an invariant expression, finding an initial state, finding a terminal state, and finding the transformation step. The function itself may serve as an invariant. This leads in general to very elegant, short, and transparent expressions, for example, to compute the sum of integers, we may write the function

$$\alpha S\ \lambda K\ \Delta\Delta\ (K=0)\ 0\ (K+\ .S\ (K-1))$$

or, to compute the elements of the Fibonacci series, we write:

$$\alpha F\ \lambda K\ \Delta\Delta\ ((K=1) \vee (K=0))\ 1\ (.F\ (K-1) + .F\ (K-2))$$

A similar recursive expression was given for the binominal coefficient. The transformation step consists of applying the alpha reduction rule. This is repeated as often as necessary, that is until the F-free branch of the conditional is selected. The invariant expression becomes larger and larger while its value remains constant. Because of the order in which terms are associated, arithmetic reduction can take place only after all terms have been generated.

In the two examples intuition and knowledge has been consulted to achieve solutions which are sometimes more efficient than the genuine recursive ones for various reasons. Calls of the same function with the same argument are not computed several times when possible. Terms are associated such that arithmetic reduction may take place as soon as possible. The structure of the corresponding invariants is a datastructure in the reduction language, too, and is regular input to the system. That means it is one expression and can be considered as one item if one wishes to do so. Let us denote this invariant structure by IS. It appears in four parts of the iterative program construction:

$$.\alpha F \subset \lambda\ \langle IS3\rangle\ \langle\text{'arbitrary'}\rangle \subset \lambda\ \langle IS1\rangle\ .F\ \langle\ IS2\ \rangle\ ]\ \langle IS4\rangle$$

IS1 } transformation step
IS2 }
IS3 termination state
IS4 beginning state as argument

This is excutable code for the reduction language machine, not efficient code, but suitable to convince another programmer about its correctness.

However, there is no recipe which leads to a unique solution. As we pointed out, several different transformation steps can be designed. Their difference is essential and will not disappear later on.

Additional knowledge has to be consulted to select an appropiate transformation, if, for example, as in plan G1 the effects of the implemented combination of arithmetic and number representation have to be considered.

As a general observation we can state that there is a fan out of solutions which differ in a non-trivial manner. Up to the point where conventional higher level programming languages are considered as target, these solutions remain executable by the reduction language machine, they might; however, take on a shape which does not show any relationship to the initial invariant formulation.

The transition to conventional higher level programming languages brings in a completely new dimension of variety. The transformation step becomes serialized to a sequence of assignment statements. A large number of permutations of them may be equivalent to the transformation step, an even larger number may not be. The reason for this lies in the interdependence of the components of the invariant structure in the transformation step and the particular semantics of the assignment statement. Old values get overwritten and destroyed by new values and cannot be used afterwards. If all new values depend each of all old values no direct solution exists except doubling all cells where the new values replace the old values only after all new values have been computed. One of the advantages of a functional programming language is that it will not give the programmer this kind of trouble.

One can easily see that the simplistic examples in this paper already lead to a large variety of procedural programs. If the number of statements in a loop is larger, the complexity rises enormously, and thus the difficulties of constructing a correctness proof from a given sequence of assignment statements. Coming from below, the peculiarities of the assignment statement language have to be filtered out, before an invariant can be formulated. Quite obviously, the way upward is much more difficult than the way downward. One of the advantages of a reduction language machine is the possibility to express facts about a problem as well as operations to drive an expression to a form which is considered to be the solution. It seems more natural to start out by expressing facts, then going on to specify operations which are transitions towards a solution, and remove the facts and invariant components only when they are not needed anymore. Similar concepts are considered in [7] with respect to general systems.

Conclusion

It has been pointed out that the lambda reduction language is very well suited to formulate numerical invariants because the language allows us 1) to mix freely reducible and irreducible expressions 2) to design transformation steps based on a generalized beta-reduction rule and the alpha-reduction rule. Hardware designed to perform these reduction rules constitutes a computer. Such a computer facilitates the application of well known methodologies to construct correct programs without the need for large software systems.

The thinking in terms of invariants is interdisciplinary. Invariants play an important role in analytical geometry. We have conservation laws in physics. The system of double entry bookkeeping aims also in the direction of keeping a control sum invariant.

Numerical invariants, although playing an important role, are by no means the only class of invariants. Any formal system which allows the formulation of an invariant is suitable.[2 6 14] However, to make a machine-processable object out of this formulation might require an extension of the reduction language.

The transformation of an invariant may be visualized as the driving of a point on a hyperplane defined by a function being invariant in the space of the involved variables until one assumes a desired value [See T. C. Chen [17]]. If this concept is used in the context of an operating system, for example, one probably has to consider closed paths.

Timing is another consideration. It takes some time to perform a transformation step. Such a step separates automatically into subtasks which may be processed in parallel, provided one does not use higher level programming languages which introduce unwanted interference of these subtasks. The "invariant" itself will be invariant only at certain times, moreover, there may not exist such times, since the subtasks might be synchronized in such a way that some bounded skew is permitted. Thus, a theory of invariants will have to consider synchronizing schemes in complex systems.

Acknowledgement

I would like to thank the referees for many helpful criticisms, and I would like to thank Mrs. R. Wagner for preparing the manuscript.

References

1) Berkling, K. J.
"Reduction Languages for Reduction Machines"
Proceedings of the Second Annual Symposium on Computer Architecture, January 1975, 133-140.

2) Burstall, R. M. (1969).
"Proving properties of programs by structural induction"
Comput. JL., 12, 41-8.

3) Burstall, R. M. and Darlington, J. (1975).
"Some transformations for developing recursive programs" Proceedings Internal Conference on Reliable Software,
Los Angeles, California.

4) Church, A.
"The calculi of lambda-conversion"
Anals Math. studies, No.6, Princeton Univ. Press, 1941.

5) Curry-Feys
"Combinatory-Logic", Vol.1
North Holland Publishing Company, Amsterdam, 1958.

6) Darlington, J. (1975)
"Application of program transformation to program synthesis"
Proc. of IRIA Symposium on Proving and Improving Programs.
Arc-et-Senans, pp. 133-144.

7) Genrich, H. J. and Thieler-Mevissen, G.
"The Calculus of Facts"
Proceedings, 5th Symposium on Mathematical Foundations of Computer Science 1976.
Lecture Notes in Computer Science 45, Springer, Berlin, Heidelberg, New York 1976.

8) Hewitt, C.
"Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot"
AI Memo No. 251, Project MAC, M.I.T., Cambridge, Mass.(April 1972).

9) Hommes, F.
"SIMULATION einer Reductionsmaschine"
Interner Bericht ISF-75-08, GMD, St.Augustin 1,
Schloss Birlinghoven, 1975

10) Hommes, F.
"The Transformation of LISP Programs into Programs written in the Reduction Language"
ISF-Bericht 77-4, GMD, St.Augustin 1,
Schloss Birlinghoven, 1977

11) Manna, Z. and Pnueli, A.
Formalization of properties of functional programs.
J. ACM 17(1970), 155-569.

12) Manna, Z. and Waldinger, R.
"Toward automatic program synthesis"
Comm. ACM 14(3) (March 1971), 151-165.

13) Manna, Z. and Waldinger, R.
"Knowledge and Reasoning in Program Synthesis"
Artificial Intelligence 6(1975), 175-208

14) Manna, Z. and Waldinger, R.
"Synthesis: Dreams => Programs"
Stanford AI Lab Memo AIM 302 Comp. Sci. Dept. Report No. STAN-CS-77-630

15) Milner, R.
"Implementation and Applications of Scott's Logic for Computable Functions."
Proc. Conf. on Proving Assertions about Programs. New Mexico State University.

16) Strong, H. R.
"Translating recursion equations into flow charts"
Second Ann. ACM Symp. on Theory of Computing, Northhampton, Mass. 1970

17) Tien Chi Chen
"Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots"
IBM J. Res. Develop., Vol. 16, No. 4, 380-388, July 1972

18) Waldinger, R. J. and Levitt, K. N.
"Reasoning About Programs"
Artificial Intelligence 5(1974), 235-316

Discussion following Berkling's Talk

Berkling Many people say that languages should be machine-independent. I disagree. A language should serve to give insight into the reality of a computer; it should not obscure it. This would make programs more portable because you know what must be changed when you move to another machine.

I don't think that high-level-language machine architecture helps us solve our problems. All that this does is to push the interface between hardware and software around — this doesn't help solve the problems of writing software because the user interface remains the same. Nor do the existing languages deserve to be cemented in hardware. We need to do something more radical.

My aim is to avoid software. I do not really fit in this session. The session is called *hardware for software*; I'm trying to construct hardware such that the need for software does not occur.

Mosses Have you managed to persuade programmers to program in your notation?

Berkling It is difficult but they are starting to do so.

Berkling The fundamental problem of machine architecture is getting together the operators and the operands. Conventional machines do this by giving the operators the addresses of their operands. An addressless computer architecture avoids the software associated with generating and manipulating addresses.

What should a machine architecture support? One needs a notation of facts and specifications with which to write down one's problem in a machine recognisable form. One should be able to put down what one thinks one's problem is, and give it to the machine. Then one should have access to meaning-preserving transformations. John Backus calls these *"reduction rules"*. Using these, one transforms the problem specifications into what one considers a solution. So one has to walk up to the machine with an already existing solution! All the machine can do for us is to transform this solution into another form that we find more agreeable.

There is no such thing as *artificial intelligence*, which does something for you. You have a machine — a mechanical device. It has transformation rules built into it. You can convince yourself that these rules are the right ones; but nothing more is made available to you. And because the machine contains no more than meaning-preserving transformation facilities, it cannot generate more meaning than it has originally received. Therefore, the natural stupidity of human beings cannot be improved upon or dispensed with. You have to have human interaction.

The set of reduction rules is not confined to the Lambda Calculus. There are a whole set of reductions that can be performed on strings — Lambda Calculus *Beta reduction* is only one of them, but I decided to start with that because it is difficult. We are presently looking at other reduction rules — for example those of Backus.

SESSION H

What Have We Learned Since ...

What indeed?

A Personal Peek into the Future

Mike Spier

A Personal Peek Into The Future

Mike Spier
Aarhus University

There is no uncertainty about the direction of hardware evolution. Things are becoming small and inexpensive. But the technology is sophisticated and expensive, requiring substantial capital resources to set up and operate and high volume sales to show a profit.

In the giant mainframe business, you sell the first one or two and break even; half a dozen and business is sweet. In the faraway past all computer business was like that. Much of the business still is like that, and there will always be a market for large computers.

The large computer was ideally sold while golfing with the client company's executive. The technical specifications were also transacted at lofty managerial levels, the necessary knowhow being acquired through monthly readings of Datamation or Computer Decisions. Consulting the people who actually use the equipment - as distinct from those who manage it - was the exception rather than the rule. Like royal marriages, negotiation of contract and approval of merchandise were deemed too serious to be delegated to the principals, who were supposed to enjoy the consumation anyway.

There are an order of magnitude more minis installed than there are traditional mainframes, and several orders of magnitude more micros than minis. Whether the word "installed" still applies to the micro computer is an interesting subject of contemplation. Technically, the act of putting one into the pocket calculator, pinball machine, wrist watch, intelligent terminal, camera, telephone, washing machine, Chrysler, bomb, RadaRange, Mattel toy, etc. is an act of installation. But the place in which such computer can be found is no longer a "computer installation". The above mentioned examples of micro computer usage are still relatively simple automata.

What will happen when we seriously make and market real, useful computer applications in micro packaging? I see the following circumstances:

- We will not be able to make it too expensive. Even now one pays no more than hundreds or a few thousands of dollars for a micro computer that is known to contain perhaps $37.85 of hardware devices. Push the price up too high, and any kid with a basement, a pencil and a paper-tape punch will be able to compete with you profitably. As a matter of fact that is exactly what they are doing right now. So the product must be priced sufficiently low to exclude the basement kids, and profits must be made on volume.
- Volume sales are no longer made to corporate entities equipped with golfing executives, and the customer can hardly be bedazzled with technical esoterica because he is not in the habit of reading Computer Decisions. * So (at least initially) he will take the non-professional and non-managerial attitude, that the thing is to be judged on the merits of what it actually can do for him. This naive attitude will ultimately become more reasonable and we shall be able to convince him to pay more attention to essentials such as gull wings, vinyl roof, decorator lighting group etc. After all, the chronometer/chronograph/alarm/scientific-calculator wrist-data-center is already upon us. Still, the problem of satisfying the actual end user will not go away completely. It is a novel problem for the industry to solve.

- Micros can be either totally hard-wired, or programmable. We know how to make and mass-merchandise the programless variety. But if there is a bug in it, or if whatever it does perfectly happens to be something nobody wants done, then we have a problem. Anyway, the usages to which programmed computers have traditionally been put do not admit of standardised mass production. [Proof: had there been any way to make the single computer system to satisfy all needs, that would have been done long ago. Nobody can accuse IBM of not having

---

* We will see this situation change as the computer industry discovers Madison Avenue and we are given the computer equivalent of "The Heartache Of Psoriasis", and "Lower In Polyunsaturates".

tried.] So it seems certain that the micros will be programmed, even if the program is permanently fixed in ROM. "Choose your personalised checking account management program from our new 1988 catalog. Any program only $19.95, two for $34.95. For purchases over $50 you get a free factorialisation subroutine, a $7.95 value."

- So programs, and programmers, will be around for a long time to come. Having personally seen three generations of hardware engineers become obsolescent (and unemployed), I find the thought most soothing. But the economics of the situation are becoming baffling. At a cost to the employer of at least $30K per year, and at a profit margin per sold unit of possibly less than $100, how many units have to be sold just to cover the programmer's cost? Now the programmer has to be able to perform two feats
  - the program must do something that will satisfy the masses, for masses of buyers are needed, and
  - the program must be bug-free.

  Because, what will it cost to provide software support and bug fixes to all those multitudes? Indeed, what would be the mechanics of such an undertaking?

At this point one perceives a disturbing difficulty. We do not know how to make bug and maintenance free programs. Even now the manufacturers' low-end minis maybe priced less than some of the software that can be run on them. How can we sell a $9.95 machine when its program costs another $200 to maintain? I foresee thriving business opportunities for both the kid with the basement and the neighbourhood pusher of bootleg programs. Inventors of tamper-proof, self-destructing programs and contraptions will also prosper.

Today's industry is shaped and dominated by the people who succeed when the business was mainframes. That same business is still there, but a present-day $100K computer system does things and has a performance unthought of in 1968 when a lesser machine cost five or ten times as much. And the software package that was produced for $100K in 1968 will cost five or ten times as much to produce today. Even in the traditional business,

the economics have been coming dangerously askew; what the economics of the software-for-micros business will be, cannot now even be guessed. There is need for new ideas and fresh insights. The present management class is realising that problems are brewing; it seems to be fighting back by intensifying its traditional activities, so to say "if this helped us succeed in the difficult days of the 360, doing more of it and harder should make us succeed with this 8080".

When confronted with this managerial view of the world, they would make us believe that they have mapped the land, improved the terrain, cut roads and laid rail beds; and that the only remaining problem is to increase the efficiency of traversing the land, of getting from here to there. Then I look at the world as I know it, and the best I can see is an outline of the land with here and there a familiar internal landmark. And the best I can do with the remaining space is to mark it with a warning, as did the cartographers of old, "here be dragons".

Editor's Note

The final session of the workshop was a general discussion. The discussion ranged over many topics, but the main theme was:

— What have we learned since 1968?
— What have we learned in this workshop?

1968 was the year of a software engineering conference in Garmisch-Partenkirchen, and several of the speakers referred to this conference.

I believe that the following is a faithful summary of what was said during the session, but I have completely reorganised the order in which things were said. The session started with a summary by Herb Grosch, and a short presentation by Mike Spier. I have re-ordered Grosch's and Spier's remarks, so that they appear in context with the discussions that they provoked instead of all-together at the beginning. I have also rearranged the discussions, so that contributions are more-or-less grouped together under topics. However many statements deal with several topics, for example Barton's, which I have collected together as a conclusion.

Computer Science versus Software Engineering

Spier I think we have learned very little since 1968. A lot of work has been done — but most of it seems futile to me. But perhaps I am being unfair.

In this workshop it has been clear that we need to define more clearly what we mean by certain words. We have spent a lot of time talking at cross purposes because we used words with different meanings.

There are problems whose solution is well understood (for example the *Eight-Queens Problem* and the *Towers of Hanoi*). The solutions to these problems have nothing really to do with computers. It is fun to solve them with computers, but rather unimportant. Let us call these the *class 1 problems*.

There are also problems which are well understood but whose solution requires thought. Let us call these the *class 2 problems*.

Then there is a gulf.

Then we come to the *class 3 problems* which software engineering deals with. These problems are largely undefined, and as soon as we think we understand them they have changed again. The solutions are unclear, and are complicated by various *"non-technical"* considerations, such as deadlines, efficiency problems, labour problems, contracts and so on.

Software engineering problems have to do with massive detail. We are not talking about small programs, written in a few lines and disposable, but huge programs like *OS/360*. We cannot afford to throw away such programs — even if they cost enormous amounts to maintain, because we cannot afford to produce new ones.

I think that there has been a lot of misunderstanding in this workshop between people who solve these various types of problems. Solutions from the world of the Eight-Queens Problem are of little use in the software engineering world. We need to be clear which problems we are talking about. I will define software engineering: It is a way of dealing with massive irrational complexity in the industrial production of computer programs. The problems are not restricted to coding,

elegance and efficiency; they are economical, social, psychological, educational, legal, political. There are no simple clear-cut issues. But we have to deal with these problems because they are there in the real world. We may not like them, but they exist.

Flynn I am a mathematician. Creative acts, in mathematics or in the arts, or anywhere must occur naturally — they cannot be forced. For example we cannot simply sit down and expect by hard work to produce a painting or a simplified machine architecture. If we do produce such a work of art or architecture then we can show it to others and share it with them, and in this way change the way they look at the world. But creative work cannot be factory-produced.

Spier's *class 1 problems* — the well-understood academic ones, are really the most important ones of all. We, the academic community, should concentrate on solving those kinds of problems to show people beauty, in the hope that if they see enough of it then it will change their view of the world, and one day maybe they will also produce something beautiful.

If enough people produce and share elegant solutions then maybe those people who have to find good solutions in a short period of time will naturally produce elegant and simple solutions in a short period of time.

Elegance and style are not timeless values. The beautiful abstract color works of Mark Rothko have almost nothing in common with the Flemish painters of the 15th century, even though you might find Rothko and Flemish painters in the same museum. Someday, soon I hope, people will describe a front-end editor as being *elegant*. Someone may, soon I hope, describe a piece of software as being elegant and beautiful, not just because it looks good, but also in part because it is genuinely robust, portable, and was developed in 4 weeks. I would love to hear someone describe a series of error diagnostics as being *brilliantly put together and phrased.* These daydreams have as much in common with Spier's class 1 problems as Rothko has in common with van Eyck — they are different aspects of style and elegance.

Grosch What we came here for was to look for tools and ways of doing things which would help the computer community. Inspirational happy moments, such as inventing a new simplifying idea, are great, but most of what we do is just grinding away. And I don't think we have come up with grinding tools in this meeting.

I am disturbed by the fact that the academics here were really more interested in talking about their theories than in the problems of the real world. The academic does not usually work with the huge systems which give problems. He tends to work with small systems and he does not work under the constraints of the commercial world.

In fact university systems are often not very good — they are poorly documented and not robust, and when the inventor leaves they collapse. In contrast consider a task like that of the *SABRE* folk when they switched to an entirely different mainframe system while staying online all the time.

I would advocate that if we hold future Software Engineering Workshops, we should concentrate more on the systems that give us trouble, rather than worrying about the troubles we might have with certain systems in the future, if we ever build them.

Stoy We academics are not doing software engineering; we are doing draughtsmanship. The measure of our success in draughtsmanship is whether we can write beautiful programs — I mean breathtakingly beautiful programs — for the simple problems. That is why we are interested in simple problems — they give us a way of testing our ideas and of seeing whether they will be any good for big problems.

This way of looking at things is not going to have any effect on the way programmers work today, because the draughtsman's tools available to them are so shoddy.

Our hope is that when the present system collapses under its own weight, we will have something ready to replace it.

System Costs

Grosch Another thing I would like to say is that we must be very careful when we look at measurements of speed and cost.

I remind you of Grosch's law. It was an easy discovery and was largely a self-fulfilling prophecy. Because people who have had to set a price on new equipment which is 4 times faster than the old, have found it convenient to use Grosch's law and to charge twice as much.

When we look at the new hardware available today, we tend to say *"It costs nothing"* and that Grosch's law is obsolete. I don't think this is so. Even the cheapest CPU will not work usefully unless it is surrounded by peripheral equipment to get the data in and out. And peripheral equipment has not dropped so fast in price. Also you need a great deal of software — and that software is going up in expense.

I predict that by the end of the century 80% of the production costs of a computer system will be human costs. It is already over 60%.

Stoy If we want cheap computing systems then there are two ways we can go. One is the *Model-T* systems which Grosch has talked about — cheap packages with no modification possible. There will be variety on the market as with all other consumer goods — perhaps not as much variety as some people would like, but most people will be satisfied. The other type of systems, for those who want to do it themselves, will be individually tailored systems based on published software. Software will be published in books in a readable simple notation. This second approach is what we are looking at in the new software engineering project at Oxford.

Grosch There has been a discussion about the use of money as a measure for software goodness. This also happened in the '68 conference.

Different users have different measures for what various things cost. For example some users are much more concerned with hardware reliability than others. So our money calculations of the value of features must not only be a function of the supplier but also of the user.

Beitz Money is not the only thing of interest to users. They are also frightened of their vendors — look how many software houses successfully offer alternatives to *IBM* products.

Halphen You talk about the users, but a user sees the world through a specific interface. Thus all your talk about standards and tools is no use unless the manufacturers take notice of them. We users are aware of what is going on and we use what ideas we can — for example structured programming techniques, but we are tied to the vendors.

The manufacturers, especially *IBM*, are extremely insensitive to the effects of university research, or even their own research. There are two big problems, complexity and compatibility. For example, *IBM* use the compatibility argument as a way to reject all suggestions from users for improvement — they say that the suggested system *"wouldn't be compatible"*.

What the users in the field need is some way to affect those monsters. We have plenty of tools

but we can't use them.

Communication and Correctness

Grosch I would like to draw your attention to what I regard as the fundamental problem of software engineering. We have almost unlimited hardware resources already, and more coming. There are two limits in hardware — the speed of propagation of signals and the problem of heat dissipation. But we don't have to worry about these barriers.

But there is a big barrier in software which affects us much more. This is the difference between the rich, redundant, ever-changing, poetically satisfying human languages, and the rigid, non-redundant, precise, difficult languages of the computer.

The task of the software artist (I won't say engineer) is to do what he can to penetrate that barrier — to accept the specification of a problem in human language, and to furnish the results, also in human language, but to do this by using a tool which does not accept human language (and which, in my view, is very unlikely ever to accept human language).

The toolmaker in our field tries to get round this problem. He tries to write packages which will make the computer more like the human being. But producing these tools is much more difficult than producing most application programs, and he has to produce the tools without having them. The problem is getting started.

Grosch We have talked about correctness. But the discussion was highly theoretical, as it was 10 years ago, and I doubt if a practical computer user will have learned anything which he can go away and use. The ultimate test of correctness is use. Proof rules are no use — we need programs that work in the field under all conditions. I think that the term *"Robust"* expresses the right concept — it expresses better than *"correct"* what sort of properties a program should have.

If we go back to the analogy of building bridges, then it is possible to make a complex simulation model of a bridge before we build it, or to make a model, but the ultimate test of a bridge is in its daily use and maintenance.

We do have large robust computer systems — I have already named *SABRE*, or the systems used by *Bell*.

Computer scientists in universities must beware of presenting the industrial users with beautiful but impractical solutions. A wonderful way of proving programs is no good if it takes years to prove a medium sized program.

Even if one can produce elegant mathematical proofs they will still be very long and it will still be difficult to see if the programs reflect the customer's wants, because he communicates his ideas to you in human language.

Beitz Large complex commercial systems can never be proved to be correct — there is no way to do it. We can only say whether they are in agreement with the designer's intent. In this case the problem is not proving correctness but establishing what the intent really is.

Gram If we can't eradicate bugs, could we take a more stochastic approach to programming? — produce programs which have a certain known probability of working properly in certain situations — is this feasible?

Beitz I am amused by the idea of balancing the payroll with a certain known degree of probability that is really what happens now.

(Ed.) Except that the degree of probability is unknown.

Organick I don't see why we don't regard bugs as a natural phenomenon. Hardware bugs are expect and fault-tolerant systems are built to deal with the problem. Why can't we do the same?

Beitz The objective of fault tolerance is not to tolerate the faults, but to circumvent them and to provi correct solutions in an environment which may be full of faults.

Somebody The problem does not lie in hardware or software. It has to do with people and the way we think If we are to use computers, then we need to learn to think in a completely different way. If we can think in an orderly way, then we won't have a software problem.

Grosch If you think like a machine then you will find it easy to talk to machines. And mathematicians think more like machines than ordinary people do.

Flynn But a person who thinks like a machine may find it hard to talk to a customer.

Barton Dijkstra has said that the first requisite for a programmer is ability in his native tongue.

Grosch Yes, and also for customers. What we need is customers who understand what they want and who can express what they want in clear concise English, and then explain it a second time so that it sounds like the first.

Grosch There has been a lot of talk about using new notations to describe things, and that improvements in notation would be an aid. I think that new notations are only of use and of interest in the universities; in industry we are forced to, and probably should go on using the old notations, because people in industry don't understand any others. That is why COBOL is still alive.

Stoy Barton remarked yesterday that programs should get more like mathematics. That is a dangerous sort of remark to make — because it raises hackles. So perhaps we ought to think about what it means.

A simple programming notation does have application in the messy areas that Spier was talking about. The important thing is that the coding should not introduce new mess. If we have a messy engineering problem then the program should provide an obviously appropriate model of the real-world messiness. And that is what correctness is all about. I am not looking to a future where computers produce beautifully automatic proofs of correctness; I am looking to a future where the coding is obviously correct, because it is simple. Such notations will have to be mathematically rigorous, but ordinary users won't have to bother about that, any more than users of real numbers have to bother about Dedekind cuts.

Barton Schumaker has introduced the words "*convergent*" and "*divergent*" to describe different kinds of problems. Engineers are taught that correct answers exist for their problems, and that the methods they are taught will provide these solutions. But real life is filled with divergent problems — we can never "*solve*" these problems, we can only cope with them.

Petri has brought forth a beautifully simple idea. This is that we use machines as communications media. If we take that point of view then we always realise that everything starts with people, takes the form of a message through the system, gets delayed and interacts with other messages, and then goes out to be used by people.

Beitz I am amused by the idea of balancing the payroll with a certain known degree of probability. But that is really what happens now.

(Ed.) Except that the degree of probability is unknown.

Organick I don't see why we don't regard bugs as a natural phenomenon. Hardware bugs are expected, and fault-tolerant systems are built to deal with the problem. Why can't we do the same?

Beitz The objective of fault tolerance is not to tolerate the faults, but to circumvent them and to provide correct solutions in an environment which may be full of faults.

Somebody The problem does not lie in hardware or software. It has to do with people and the way we think. If we are to use computers, then we need to learn to think in a completely different way. If we can think in an orderly way, then we won't have a software problem.

Grosch If you think like a machine then you will find it easy to talk to machines. And mathematicians think more like machines than ordinary people do.

Flynn But a person who thinks like a machine may find it hard to talk to a customer.

Barton Dijkstra has said that the first requisite for a programmer is ability in his native tongue.

Grosch Yes, and also for customers. What we need is customers who understand what they want and who can express what they want in clear concise English, and then explain it a second time so that it sounds like the first.

Grosch There has been a lot of talk about using new notations to describe things, and that improvements in notation would be an aid. I think that new notations are only of use and of interest in the universities; in industry we are forced to, and probably should go on using the old notations, because people in industry don't understand any others. That is why COBOL is still alive.

Stoy Barton remarked yesterday that programs should get more like mathematics. That is a dangerous sort of remark to make — because it raises hackles. So perhaps we ought to think about what it means.

A simple programming notation does have application in the messy areas that Spier was talking about. The important thing is that the coding should not introduce new mess. If we have a messy engineering problem then the program should provide an obviously appropriate model of the real-world messiness. And that is what correctness is all about. I am not looking to a future where computers produce beautifully automatic proofs of correctness; I am looking to a future where the coding is obviously correct, because it is simple. Such notations will have to be mathematically rigorous, but ordinary users won't have to bother about that, any more than users of real numbers have to bother about Dedekind cuts.

Barton Schumaker has introduced the words *"convergent"* and *"divergent"* to describe different kinds of problems. Engineers are taught that correct answers exist for their problems, and that the methods they are taught will provide these solutions. But real life is filled with divergent problems — we can never *"solve"* these problems, we can only cope with them.

Petri has brought forth a beautifully simple idea. This is that we use machines as communications media. If we take that point of view then we always realise that everything starts with people, takes the form of a message through the system, gets delayed and interacts with other messages, and then goes out to be used by people.

If we take this view that computers are a new kind of communications system where people are a critical part, and that people deal with divergent problems, then we realise that many programs which get written are subject to change, and should not be thought of as fixed.

Convergent problems belong to a different world. They are often based on mathematics and their solutions may have some permanence.

The Organisation of Software Engineering

Grosch Somebody, I can't remember who, said that software engineering problems are like those of the city planner as compared with the house architect. It is very nice to be able to build a nice house, or a cathedral, but the problems of designing a town are much more complicated — you need all sorts of buildings, and roads and sewers and so on.

There are ways of making the job easier — for example a doorknob standard will help. But it won't help much, and nobody judges a cathedral by its doorknobs.

We need to have people who have city planner-like skills, and then under them can work people who are experts at building houses, churches and so on. Then in the end we might produce enormous programs which are interesting to live with, even if they do still contain bugs.

Spier One of our problems is that we don't know what design is. We have something we call *"design"*, but this is usually wishful thinking — *"we want this and that"*. Then we get an implementation (which is the first specific complete design) — we have produced a prototype. At this stage we ought to make the production model, but instead we sell the prototype. Then customers start using it, they find bugs and they ask for new features. Thus we get into a horrible spiral of maintenance with no sensible way out of it.

Mechanical engineers (who really know much more about the materials they work with than we do) expect to have to make a couple of prototypes before they get the design right for manufacture. But we continue to ship prototypes.

The process of manufacture should be as follows:

1. Produce the first version, as we do now.
2. Do performance measurements to make it more efficient.
3. Rewrite the whole thing from scratch.

Shelness has told us that rewriting a complete operating system only costs one tenth of the original — so it isn't really very expensive.

Then we can ship the product to the customer and we can maintain the product and improve it intelligently.

Organick I think that Spier is doing some wishful thinking. He wishes that programmers would be forceful enough to change the current process of design into a better one. But commercial programmers are caught up in the process that exists now, and it seems to me to be a natural one. I don't see any way that we can force a change overnight.

There are two worlds — the academics, who work on problems which interest them; and the people who write production software. Occasionally the worlds meet but there is not much connection and ideas from the academic world only filter through very slowly.

I see no end to the current (bad) way of writing large software products the way Spier described.

Bennett There has to be an end.

Spier I predict that things will change within the next 20 years. People will not be able to afford mistakes in software. We must bear in mind what is the penalty of making a mistake. If we sell a computer for 5 dollars then we have made less than 5 dollars profit on it. In this case we simply cannot afford to write to the buyer and tell him about bugs in his machine, or to visit him and correct them. In the future a computer manufacturer will not be able to afford the luxury of a single mistake.

Wait until a couple of computer companies have been driven into bankrupcy by bugs and then we will see that software products are carefully tested and rewritten before they are released.

Shelness Structure and function are related. In other disciplines function remains fairly static, and appropriate structures are built to implement it. However we allow the functional specifications of our systems to change very rapidly while we are building the systems, and as a result we cannot build appropriate structures.

Perhaps we should stop this — and say that functional specifications must change more slowly.

Spier I think you are right. We ought to stabilise the problem long enough to be able to solve it.

Gram Perhaps one of the reasons that software engineering problems are so difficult is that we don't impose elegance early on. Elegance must never be thought of as a bad thing.

Spier Elegance can come in once we have a reasonable understanding of the problem and the solution.

I approve of elegance. But people seem to get carried away with the idea, so that they don't care if they make an elegant bug. In this case their priorities are wrong.

Beitz I don't think that the big software problem is in producing good code — we ought to de-emphasise this. The big problem is in the social environment the programmer works in — *"how many lines of code have you produced today?"*.

I remember one successful 9 month project where after 8 months I still hadn't used any computer time. My manager was wild about this, but we produced the code easily in 3 weeks.

The business of producing visible output has become far too important.

Beitz I have been in a firm which used the following design process: Whenever a program was to be produced two people were given the job. They flipped a coin to see which of them should produce the program and which should produce the tests for it. They were not allowed to come to the computer until they had written their programs and given them to each other. Then they discovered all sorts of misunderstandings and mistakes. Sometimes they had to go back to the user and ask him what it was he really wanted. That design process was a very good one.

Lee Spier and Beitz seem to want to impose improvements top-down — by altering the management structure of software projects. But perhaps the only way we in the universities can improve things is bottom-up — by teaching our students to work in the right way.

In fact it may be in the high-school that we should be teaching responsible programming. I am very concerned about high-school programming courses — they teach programming tricks, not programming methodology.

Gram We are all eager to learn new manual tools and skills — give me a new pocket calculator and I will play with it for hours finding new ways of using it. But we are reluctant to change our intellectual work habits. And that is the big problem in Software Engineering.

All of the good methods you are looking for exist already — *Jackson*, *SADT*, *Syskon* in Denmark . . . lots of them. We, you and I, read the papers and books, we listen to the inventors, we shrug our shoulders, and we go home and find good reasons for not using these methods and continuing the way we have always done.

The Future

Lauesen I have a feeling that most of the people here believe that there has been very little progress in software engineering in the last few years. I am not sure that you are right. The size of the problems which we are trying to solve is growing, but the tools we are using are becoming better. Thus the size of the problems which we can solve properly is also growing. Today we can write a well-structured small operating system. Nobody could do that 10 years ago.

Grosch There are other Grosch's Laws. The third one is: *"Things can get worse without limit"*. Spier is mistaken when he says that things will be better in 20 years — they will be worse than they are now. And in 40 years they be worse still.

There is no way round the basic problem — that machines don't think like people, and people don't think like machines. As we tackle harder and harder problems we will get messier and messier answers.

Mosses Why do people want to try to solve such large problems?

Gram It is human nature. We always attack problems which are slightly bigger than the tools we have.

Spier People aren't upset because software engineering is difficult, but because it is a mess. We produce messes and we sell messes and we expect the users to cope with the mess. In 20 years, because of the changing technological situation, nobody will be able to afford to sell a mess. Bugs will be eradicated.

Grosch No — look at automobiles. They don't work properly but we have all become accustomed to living with them, even though some of their systems don't work.

We should remember that there are lots of users who are pretty happy with their accounting systems and other programs — even though these are full of bugs and produce erroneous results. Some of the users don't know about the bugs, and are foolishly happy, but others have learned to live with them. In general we have learned to live and cope with complex systems which are not perfect.

Organick The complexity of computer systems will be reduced as we produce better structures in hardware and software.

Spier No. Certain systems are intrinsically complicated and irrational. In this case all we can do is push the complication about from one place to another — fancy hardware will not remove it.

Grosch One of the solutions to software engineering problems put forward in this workshop was to change the computer architecture. That is not realistic. Of course it is nice to have better machines — but they don't make the other problems go away.

Of course certain things help — standards and performance measurements for example. But there are infinite numbers of problems waiting for us, so a new tool doesn't get us out of the morass: we just move to a different part of the swamp. We have a bigger paddle to paddle with, but the swamp is deeper, so we have to paddle harder.

In 1968 McClure produced a paper which showed that the number of lines of code written to support various machines increases exponentially. That prediction still holds today. This means that as we go on we will have hundreds of thousands of people writing system code. We are nearing that figure now.

Beitz Kronos was written in its entirity by 2 people, and it now takes 30 people to support it!

Grosch I would like to quote from Al Perlis, he said:

> *"We kid ourselves if we believe that software systems can only be designed and built by small numbers of people. If we adopt that view this subject will remain precisely as it is today, and will ultimately die.*
>
> *We must learn how to build software systems using hundreds or possibly thousands of people. It is not going to be easy. But it is quite clear that when one deals with a system beyond a certain degree of complexity, for example IBM's TSS/360, regardless of whether well designed or poorly designed its size will grow, and the sequences of changes which one wishes to make on it can be implemented in any reasonable way only by a large body of people, each of whom does a small job."*

Now Barton and I are agreed that we deplore this. But he deplores it and disagrees with it and I deplore it and agree with it. I don't claim that the world I am describing is a nice world; I think it is a bad world and I would change it if I could. I do what I can in the ACM and so on to improve things — but I don't believe it will do any good.

Barton You have to try though.

A friend of mine used to say *"Don't talk about complexity when you really mean complication."* Complexity is inevitable and we have to cope with it, but we don't have to live with the complications which are the artifacts of money-grubbing business.

Those of us who started early had no doubts about what we were doing with computers. The 1968 Software Engineering Conference was the first occasion that I know of where people dared to say in public that we are working with machines that can hurt us if we are not careful.

Barton I believe that there are some computer science students in the audience here, and I would like to address some remarks especially to them.

When we start a project we ought to consider whether we should use a machine at all. Only when we have shown that it is necessary and desirable should we use a computer. This idea can do more to avoid complication and burgeoning complexity than any other single thing.

Also, if you start off with the idea that a system can be free of human factors, then you are on the road to disaster because any human being is immensely complex and makes a terribly important contribution to any system he is part of.

Some of you may have noticed that as airline reservation systems become more and more

automated and complicated, it has become more and more difficult to tell whether one is talking to a person or a machine on the other end of the telephone. First comes a recording; then there is a compulsory music interlude, during which time you can relieve your tensions by swearing into the phone — nobody's listening so it doesn't matter; then a human voice comes on — but even the voice sounds mechanical. The people have become parts of a machine, and that is bad, because people don't function well as parts of a machine.

A system without people in it isn't a system at all. And if a system does have people in it then there must be value judgements, approximate solutions, brilliant make-do's, and so on connected with it.

Barton I am interested in the history of technology. One thing which I have learned, and which is very comforting to me, is that all technologies in the past have had a period of growth, a life, a decay and then they have died. It is possible that the sort of technology that Grosch is talking about is getting close to the time when it will die. We can look at many examples of industries which were prominent for a time, but in a couple of decades were gone. My favorite example is the ice industry in New England in the early 1800's. The New Englanders found that they could sell ice all over the world. The industry became very big — some of the largest buildings then built were ice houses. But then the industry disappeared.

An important fact about the computing business is that it has fed on growth, producing its own problems.

Barton I don't think that we should accept things as they are. We should be more anarchistic:

- Things have a right size — find out what it is.
- Computers have an enormous effect. Much of it is not good.
- We should think about the effect of our technology on the world.

For example computers are likely to have an unpleasant impact on the whole world of printing, book-making, publication of periodicals and so on.

Why do I say that this will be an unpleasant effect? — because I don't think that anybody wants to spend more time staring at a TV screen than they do now.

If we become the animal who depends on a cathode ray tube for his contact with reality, then we are not long for this world.