SMALL SCALE
SOFTWARE ENGINEERING

Robert W. Witty

Submitted to Brunel University in partial fulfilment
of the requirements for the degree of Doctor of Philosophy

Brunel University
Dept. of Computer Science
September 1981

Paginated blank pages are scanned as found in original thesis

No information is missing

ABSTRACT

In computing, the Software Crisis has arisen because software projects cannot meet their planned timescales, functional capabilities, reliability levels and budgets This thesis reduces the general problem down to the Small Scale Software Engineering goal of improving the quality and tractability of the designs of individual programs.

It is demonstrated that the application of eight abstractions (set, sequence, hierarchy, h-reduction, integration, induction, enumeration, generation) can lead to a reduction in the size and complexity of and an increase in the quality of software designs when expressed via Dimensional Design, a new representational technique which uses the three spatial dimensions to represent set, sequence and hierarchy, whilst special symbols and axioms encode the other abstractions. Dimensional Designs are trees of symbols whose edges perceptually encode the relationships between the nodal symbols. They are easy to draw and manipulate both manually and mechanically.

Details are given of real software projects already undertaken using Dimensional Design. Its tool kit, DD/ROOTS, produces high quality, machine drawn, detailed design documentation plus novel quality control information A run time monitor records and animates execution, measures CPU time and takes snapshots etc; all these results are represented according to Dimensional Design principles to maintain conceptual integrity with the design. These techniques are illustrated by the development of a non-trivial example program.

Dimensional Design is axiomatised, compared to existing techniques and evaluated against the stated problem. It has advantages over existing techniques, mainly its clarity of expression and ease of manipulation of individual abstractions due to its graphical basis

TABLE OF CONTENTS

3. DIMENSIONAL DESIGN: ABSTRACTION

4. DIMENSIONAL DESIGN: REPRESENTATION

7. DIMENSIONAL DESIGN: PRACTICAL EXPERIENCE

8. DIMENSIONAL DESIGN: ASSESSMENT

ACKNOWLEDGEMENTS

I wish to thank those friends and colleagues, including TA, CB, JRG, LH (my first helper), JH, FRAH, CP, JMR, MRS, DCS, PCT, and especially DAD, WW, and VMW, who have helped and sustained me during this pastime.

DECLARATION

The Safe Programming idea was due to Dr. T. Anderson. Dr. D.A. Duce designed and implemented the ROOTS preprocessor. Several sandwich students (SW, FL, GP, PG and JM) helped to implement and develop the Dimensional Design drawing programs and the ROOTS tools.

ABSTRACT

Chapter 1 introduces the overall problem, the Computer Software Crisis, which has arisen because software projects cannot meet their planned timescales, functional capabilities, reliability levels and budgets. Chapter 2 outlines the Software Life Cycle before focusing down to the small scale software engineering problem of improving the quality and tractability of the designs of individual programs.

Chapter 3 introduces eight abstractions, set, sequence, hierarchy, hierarchical reduction, integration, induction, enumeration and generation, showing examples of how their application can lead to a reduction in the size and complexity of the software design product, to the natural production of structured programs, to a reduction in the difficulty in reasoning about the design product and an increase in the quality of the design product. The key to these improvements is Dimensional Design, a new representational technique which explicitly records each use of the eight abstractions, allowing easy visual recognition of their occurrences and scopes.

Chapter 4 informally introduces Dimensional Design which uses an analogy with the three spatial dimensions to represent the Hierarchy, Set and Sequence abstractions. The Enumerative and Generative use of Hierarchical Reduction, Integration and Induction are explicitly represented by special symbols. The ability of Dimensional Design to portray the key features of software design is brought out in a wide range of examples, including instruction sequences, data structures, parallelism, design versions, step-wise refinement and input/output files. A Dimensional Design is a tree of symbols whose edges perceptually encode the relationships between the nodal symbols.

Chapter 5 discusses how Dimensional Designs may be drawn and manipulated, both physically and logically. Chapter 6 axiomatises the underlying rules

which govern the construction and manipulation of Dimensional Designs.

Chapter 7 gives details of real software projects already undertaken using Dimensional Design, an outline of the overall method they employed and the Dimensional Design specific software tools they utilised. One such set of tools, called Dimensional Design/ROOTS, is used at each stage of the Software Life Cycle in the production of a non-trivial example program. Dimensional Design/ROOTS produces high quality, machine drawn, detailed design documentation, contains novel, experimental quality control tools, a run-time monitoring system which measures CPU time, animates and records execution and contains other more conventional debugging tools, all the results of which are represented according to Dimensional Design principles to maintain conceptual integrity with the design. The representational technique is the key feature of Dimensional Design.

Chapter 8 assesses Dimensional Design by first looking at existing techniques such as high level programming languages, traditional flowcharts, trees and nested boxes. Nested boxes emerge as the most well developed competitor to Dimensional Design so a detailed comparison of the two is undertaken. Finally Dimensional Design is assessed against its stated goals - has it actually solved the small scale software engineering problem? No, so Chapter 9 suggests two sets of ideas for further work.

The appendices include four previously published papers and a large example program. The paper in section 10.1.1 gives a concise summary of the scope of the thesis and might be a suitable starting point for a reader wishing to gauge the overall range of discussion. Section 10.2 contains a large example program and details of its of design, construction, quality control and animation. A quick look at section 10.2 might give a useful, concrete, first impression as much of the main thesis talks about practical issues in abstract terms.

To the Qoheleth,

who tackled the real problem.

PREFACE

"The beginning of wisdom for a programmer is to recognise the difference between getting his program to work and getting it right. A program which does not work is undoubtedly wrong; but a program which does work is not necessarily right. It may still be wrong because it is hard to understand; or because it is hard to maintain as the problem requirements change; or because its structure is different from the structure of the problem; or because we cannot be sure that it does indeed work."

M.A. Jackson.[37]

CHAPTER 1. SOFTWARE CRISIS

OUTLINE

Chapter 1 introduces the overall problem addressed by this thesis, the Software Crisis, which has arisen because software projects cannot meet their planned timescales, functional capabilities, reliability levels and budgets. The outstanding symptom of the Software Crisis is the massive cost and effort of maintaining a software system following its initial delivery.

The Software Crisis exists because projects are undertaken with expectations of success which are beyond the capability of present day software technology to fulfil. The software industry has responded by increasing its research into ways of improving the technology of Software Engineering.

Chapter 1 discusses the state of maturity of Software Engineering relative to other technologies and concludes that it is still in its infancy. The current state of Software Engineering is discussed in Chapter 2, which closes by selecting a sub-problem from the overall Software Crisis. Succeeding chapters propose a solution, examine the solution's feasibility and compare it with existing techniques. The appendices contain previously published papers, which summarise the scope of the thesis, and a large example program illustrating concretely the techniques described abstractly in previous chapters.

1. SOFTWARE CRISIS

1.1. The Symptoms of the Software Crisis

Contrary to that regal advice "Begin at the beginning...",[11] this thesis begins in the middle, for computing currently stands in the middle of the so called Software Crisis. What is the Software Crisis and how does it manifest itself?

The symptoms of the Software Crisis are shown by the computer industry's customers and vendors alike, both of whom are weighed down by the vast amounts of time, effort and money needed to produce today's inadequate software. Most software projects are unsuccessful at achieving their planned timescales, functional capabilities, reliability levels and budgets.

Software is now the major expense in most large computing projects. Studies by the U.S. Navy have shown that in the last twenty years the cost of software has risen from below 20% to over 75% of the total cost of computing projects and it is estimated that by 1985 the figure will be 90%.[7]

Users in the U.S.A. alone spend over ten billion dollars on software every year. Why is current software so unsatisfactory and where does all the money go?

1.2. The Causes of the Software Crisis

Figure 1-1 illustrates the breakdown of software costs for most typical large software projects. The costs of testing and maintenance (correcting errors and making changes after the system is installed) account for 75% of the cost of the software ie 75% of 75% of the total cost of the project. For instance, the SAGE system, a military defence system, had a software maintenance cost of approximately 20 million dollars per year after ten years of operation, compared to an initial production cost of 250 million dollars.[76] In

typical releases of the IBM OS/360 operating system, approximately 60% of the costs were incurred *after* the system was made available for use. In both of these examples the costs given are for maintenance only. The maintenance *plus testing* costs in these two cases probably exceeded 80% of the total cost. Although no one knows the annual world wide costs for testing and maintenance, it is known that one organisation, the U.S. Air Force, spent over 750 million dollars in 1972 on software testing alone.[72]

This first symptom of the Software Crisis, the high cost of maintenance, and thus software, is due to two causes. One is the difficulty of finding and fixing errors and inadequacies after installation (Rectification) and the other cause is the disproportionate difficulty in changing an existing software product slightly to reflect a small change in the user's requirement (Development).

The second symptom of the Software Crisis, the dissatisfied customer, stems directly from the continuous necessity for Rectification and from frustration due to the slow rate of Development. Is there a remedy for these ills?

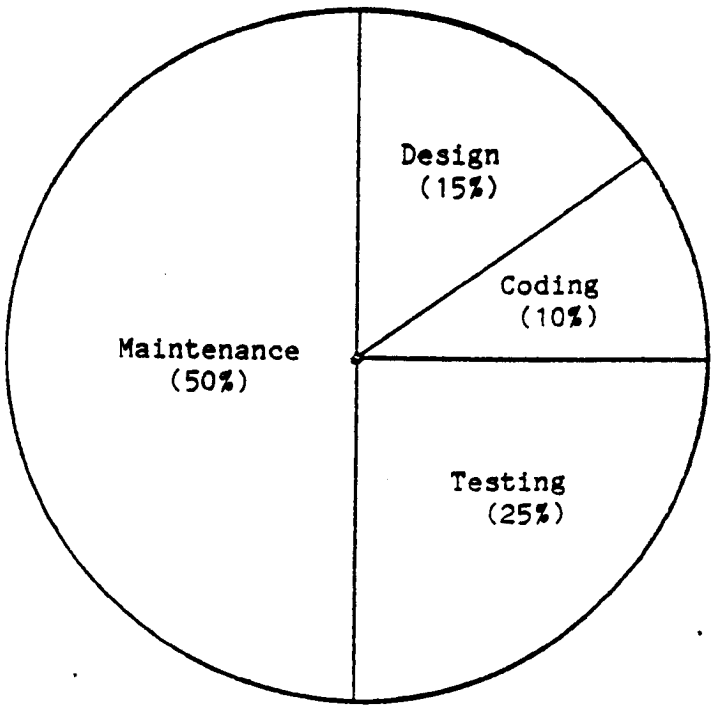


Figure 1-1. Typical Breakdown of Software Costs.

1.3. The Cure for the Software Crisis

The Software Crisis could be greatly ameliorated by two things, better customer education and improved software technology. A more knowledgeable customer would not always believe the sales pitch and his expectations and demands of his computer system would be more realistic. This avenue will not be explored further as 'caveat emptor' and (in extremis) training the user to regard bugs as features are negative, unprofessional solutions. The answer to the Software Crisis must come from improved software technology.

Although software production is a labour-intensive industry (or more accurately 'mind intensive'[55] industry), software costs would not be significantly lowered by increasing programmer productivity if the latter is a measure of the speed of designing and coding programs. In fact such an attempt would be more likely to increase costs by increasing the error rate. The best way to reduce software costs would be to reduce the maintenance component. This is not likely to be achieved by devising means to make programmers code faster, but by devising means whereby software may be designed and built with greater precision and accuracy, eliminating the need for maintenance at its source.

Precision, accuracy, reliability, value-for-money - these are terms from the realm of engineering. The Software Crisis can only be solved by the evolution of a discipline of Software Engineering, based on a sound scientific foundation. Is Software Engineering a dream or a reality?

1.4. The Birth of Software Engineering

Software Engineering, if it exists, must be an engineering discipline which is a "...systematic body of knowledge capable of transmission from one generation to the next; it also implies an established method of applying that

knowledge to solve a problem".[33]

The 'systematic body of knowledge' is the scientific foundation (Computer Science for Software Engineering). Such knowledge is passive and is the domain of the scientist for whom knowledge is an end in itself. The 'established method' is the technology used to build real products. The 'method' is an active component and is the domain of the engineer who, according to Webster, is an ingenious fellow who puts scientific knowledge to practical use. For the engineer, scientific knowledge is only a means to an end. He is also prepared to use experience, tradition and intuition, which when combined with scientific knowledge form a technology. Sometimes experience, tradition and intuition are the only aids an engineer can use when tackling a relatively new field of endeavour. Engineers did not wait for the establishment of a scientific basis before building the first cathedral, steam engine, aeroplane or OS/360 release.

Scarrott[70] proposes a general model of the evolution of a technology consisting of four phases - birth, adolescence, maturity and senility. He suggests that when a new product first appears the technology for making it is primitive but, if the product serves a useful purpose, there will be pioneering users whose need for the new product is so pressing that they are willing to adapt their practices to take advantage of it. Thus in the birth phase most of the effort is concentrated on *how to make* the new product and there is little discussion about *what purpose* the product should serve or what should be its technical specification to best serve such a purpose. In the adolescent phase the main population of users begins to appear. "Many of these (users) do not have such a clearly recognised need (as the pioneers) and, indeed, some of them may be only following fashion".[70] As a result of this expanding usage discussions regarding the new product begin to tackle the more fundamental

issues of 'what'. Nevertheless, the technology is still immature so that the adolescent stage is roughly characterised by the 'how' and 'what' efforts being about equal.

The product is mature when the technology and its scientific foundation are fully developed. Any reasonable product specification can now be made so that the crucial questions are entirely concerned with framing a desirable specification. Thus at the mature stage, the 'what' factor is dominant. Finally the product becomes senile when its social purpose either disappears or is met by other products and it goes out of use and into history.

Figure 1-2 compares the evolution of the steam engine, the aeroplane and the computer. Scarrott[70] comments on figure 1-2 that "it suggests that (computers) are still in the adolescent stage. The first useful electronic computer was developed during the Second World War. Since that time, the physicists and electronic engineers have done a splendid job introducing solid-state devices and large scale fabrication techniques, which have removed many of the technological constraints that shaped the early (computers). However the refinement of technology has not yet been complemented by an understanding of the natural properties of information adequate to guide the deployment of our new found technological mastery, so that a first approximation to an understanding of the present situation in (computing) would be to liken it to the situation in the evolution of steam engines in the early 19th century after techniques for casting, forging and machining had provided the 'means' but before the theoreticians such as Carnot and Rankine had illuminated the 'ends' for steam-engine design".

It took over 150 years to perfect steam technology but computer hardware has progressed from the huge, unreliable 20,000 valve, 30 ton ENIAC of 1945 to today's microprocessor in only 30 years. This rapid progress has been

exclusively in the direction of *how* to build computers to almost exactly the same design that von Neumann et al invented in the 1940s. It is depressing to contrast this stagnation of computer architecture (the 'what' aspect of hardware) with von Neumann's[60] vision of self-reorganising, self-reproducing machines operating on a probabilistic logic principle to give reliability from unreliable components which he saw from the earliest days of computing.

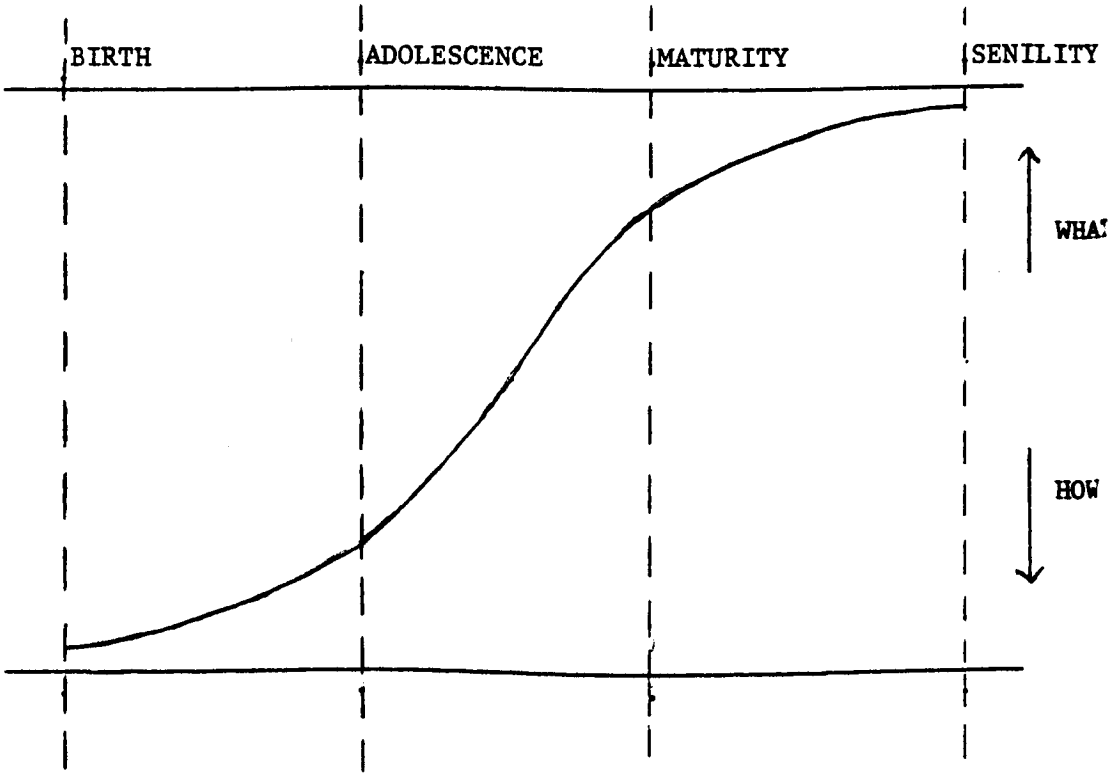
On balance it seems as though computer hardware technology is progressing satisfactorily through adolescence, according to Scarrott's model, with the prophets of 'what' just beginning to be recognised,[4,16] but how about software technology? How far has it evolved?

Software is in the middle of the Software Crisis with no 30 years of spectacular progress to report. On the contrary, no one today knows *how* to design satisfactory, reliable new software nor does anyone know *how* to rectify or develop existing software. Software technology today is at an equivalent stage to hardware during its valve era 30 years ago. Software Engineering is in its infancy. Figure 1-3 illustrates the gap between the adolescent hardware and the infant software technology. It is the width of this gap which has provoked the Software Crisis. The gap represents the vast difference between the hardware's capability to execute instructions quickly, cheaply and reliably and the software's inability to quickly, cheaply and reliably translate the user's problems into a form capable of execution by today's machines. Modern hardware's increased capacity has led pioneering users to build increasingly more complex software systems using inadequate software techniques, instead of trading greater functional capability for greater reliability. This mistake and its consequence, the huge cost of propping up these fragile systems, are poignantly summarised by Brooks who said of his experience in leading the development of OS/360 "It is a very humbling experience to make a multi-

million dollar mistake, but it is also very memorable".[9]

The technology, the discipline of Software Engineering, was conceived by von Neumann[59] in the 1940s, born out of Silicon in the 1950s and christened in the 1960s.[58] The 1970s have taught Software Engineering a self-awareness of its infancy so that today, though the power of maturity is still a dream, the days of childish play and irresponsibility are almost over and the rapid bitter-sweet changes to adolescence approach with the 1980s.

What exactly is Software Engineering?



Steam Engines	1700	1800	1900	now
Aeroplanes	1900	1950	now	2000
Computers	1945	now	2000	

Figure 1-2. Technological Evolution.

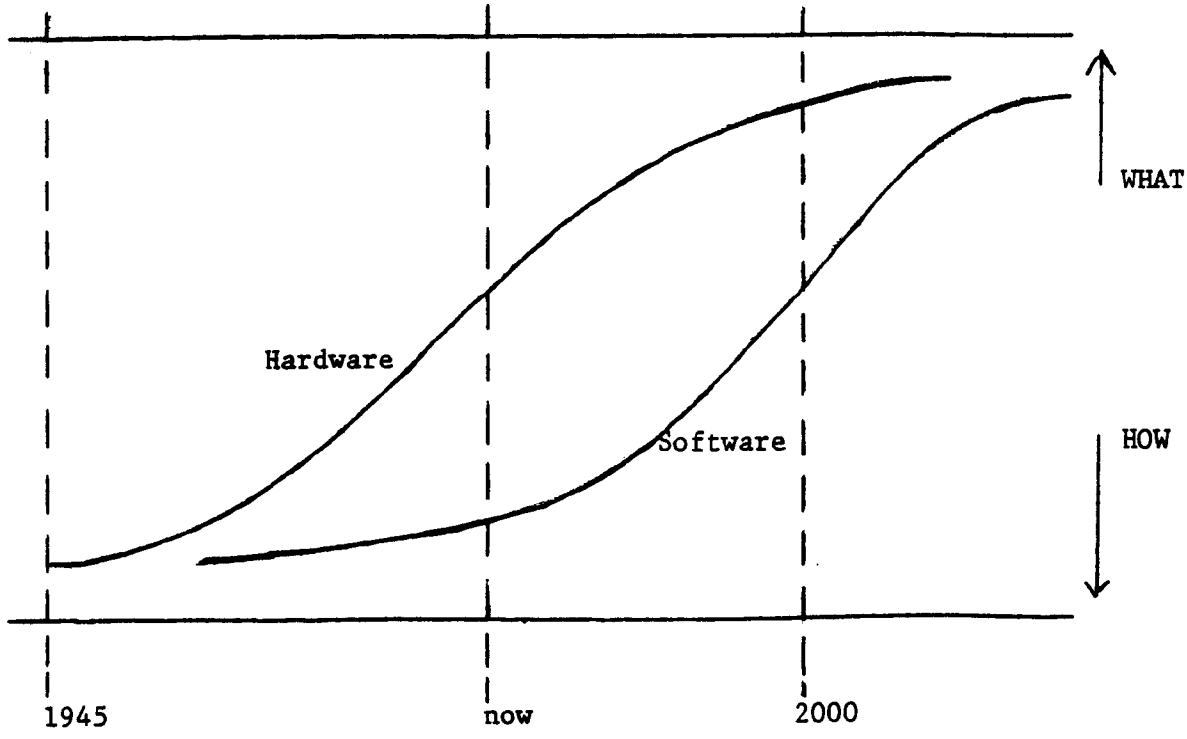


Figure 1-3. Hardware - Software Technology Gap.

CHAPTER 2. SOFTWARE ENGINEERING

OUTLINE

Chapter 2 reviews the present state of Software Engineering beginning with the people involved. Modern programming began with von Neumann and today thousands of people write programs, but how many of them are Software Engineers? The creation of a program begins with a requirement for that program and progresses through design, construction, operation and maintenance phases, collectively known as the Software Life Cycle, which is described in detail. What does the Software Life Cycle produce that is useful? For the customer it produces an operational system. For the maintenance engineer it produces a design product, something which can be realised in many different forms.

Many of the problems associated with large software systems occur with the production of individual programs, too. Techniques to produce such 'small scale' software, Structured Programming and Proof for example, show the importance of Abstraction. The principle of abstraction is examined so that in succeeding chapters examples of abstractions relevant to programming can be introduced and shown to be the basis for a practical step forward in tackling Small Scale Software Engineering.

2. SOFTWARE ENGINEERING

2.1. Who, How, What?

Chapter 1 showed that the Software Crisis has been caused by the constant increase in software complexity, from the 1940s when 'software' meant one program of less than a hundred machine instructions to today when 'software' can be a system of many cooperating programs formed from millions of machine instructions, whilst the machine instruction, the basic building block of software, has itself remained unchanged. It is the greater scale and complexity that separates Software Engineering from Programming.

For Dijkstra[15] the "basic problem is that precisely this difference in scale is one of the major sources of our difficulties in programming! ...any two things that differ in some respect by a factor of already a hundred or more are utterly incomparable". Of disregarding differences in scale he says "We tell ourselves that what we can do once (ie build a slightly bigger program), we can also do twice and by induction we fool ourselves into believing that we can do it as many times as needed, but this is just not true! A factor of a thousand is already far beyond our powers of imagination!". A software system can now be over 10,000 times the size of programs in the 1940s and 1950s. Note that Dijkstra says "our difficulties". Machines have no such problems in executing these millions of instructions.

Naur[57] feels that "modern computers are so effective that they offer advantages in use even when their powers are largely wasted. The stress has been on always larger, and, allegedly, more powerful systems, in spite of the fact that the available programmer competence is often unable to cope with their complexities". For Naur, complexity is a problem for "programmer competence" not machine capability.

Dijkstra and Naur see the Software Crisis as essentially the human problem of coping with the scale and complexity of modern software. So of all the questions one could ask about Software Engineering

How is it done?
What does it produce?
When, where and why is it done?

perhaps 'Who does it?' should come first.

2.2. Who Does It?

The first modern programmers were von Neumann and his colleagues who wrote programs to compute ballistic firing tables. They were scientists writing small, simple programs to implement well formulated mathematical procedures. The tradition of one man-one program has continued to this day amongst scientists.

When commercial Data Processing blossomed in the 1950s and 1960s it needed not individual programs but systems ie suites of cooperating programs operating on large amounts of stored data. This increase in scale and complexity brought forth first the specialist programmer and then, later, the systems analyst, reflecting the two stages to producing a commercial system. The systems analyst studied the user's requirements and produced an overall design of the proposed system's inputs, outputs, files and programs. This specification was divided up amongst the programmers who designed and coded up the individual programs. This division of labour allowed several people to work on one project. It remains the prevalent Data Processing technique.

The production of huge military and civilian systems such as anti-ICBM, airline reservation and general purpose operating systems was the next order of magnitude increase in scale and complexity. These projects ran for up to ten years, involved hundreds of staff and millions of instructions. Although by

now some scientists were spending an ever increasing fraction of their time on programming (and thereby a decreasing fraction on actual research) and although most Data Processing departments were putting 80% of their effort into maintaining existing systems,[52] it was the spectacular expense and painful inadequacy of these big, prestigious projects which forced the software industry to realise that the traditional methods were not powerful enough but that better techniques were not available.

Other, older industries have been through similar periods of evolution. The building and construction industry, for instance, developed its scientific foundations and today calls upon, for example, materials science, stress analysis and aerodynamics. The construction industry has evolved a set of project organisations to suit the scale of any given job and has produced a range of skills which can be combined on any one project (figure 2-1). The Do-It-Yourself amateur never tackles a skyscraper but he can build a garden wall, though a professional craftsman should make a better job of even such a lowly task.

Most programmers are D-I-Y men. According to Naur[57] "Historically this state of affairs is easily explained. Large scale computer programming started so recently that all of its practitioners are, in fact, amateurs". The better programmers only qualify as semi-skilled as they have not usually absorbed the full range of experience and tradition available through a proper apprenticeship. Real craftsmen are rare.

Big systems must presumably have architects but no one has emerged who is recognised as a good software architect let alone a Vanbrough or a Wren. There are no generally recognised, professionally qualified software engineers equivalent to modern civil, mechanical or electrical engineers. Software Engineering is still too young. No one is really 'doing' software engineering but

some people are actively experimenting with how to do it? How is it done today?

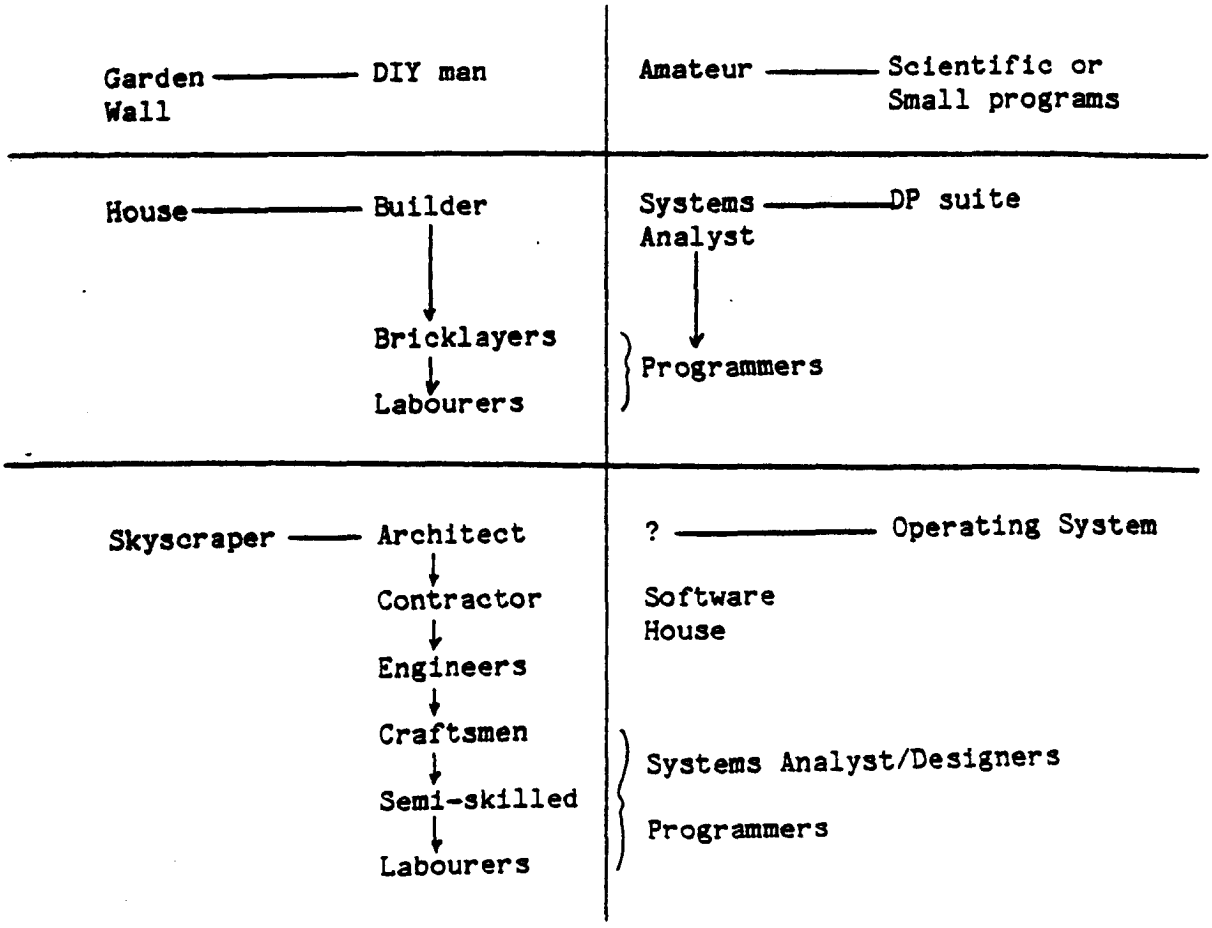


Figure 2-1. Scale and Skill.

2.3. How Is It Done?

2.3.1. The Software Life Cycle

The various phases in the production of a software system are collectively called the Software Life Cycle. This name reflects the fact that a software product exists for, usually, several years during which time it grows, changes and ages until, finally, it is replaced and dies.

It is both necessary and useful to break up the work on a project into the various phases of the Life Cycle. However there are dangers in organising the work as a single pass through each stage, the open-loop development of figure 2-2, rather than as an iterative process involving customer feedback, the closed-loop development of figure 2-3. Traditionally many projects have been open-loop developments in which the designers and better programmers moved onto new projects after the first release leaving the maintenance to lesser mortals - too often the customers. Their kind of maintenance is usually of the 'patch' variety and often makes the situation worse by introducing further errors. The designers of such a system are not involved with its maintenance at all and, after a few years, would probably find 'their' system unrecognisable. Without feedback the designers cannot learn from their mistakes and Boehm[7] reports that up to two-thirds of all software errors are introduced as the results of incomplete or inconsistent requirements specifications and overall designs. Thus the first job on any project is to work out exactly what it is the customer really wants.

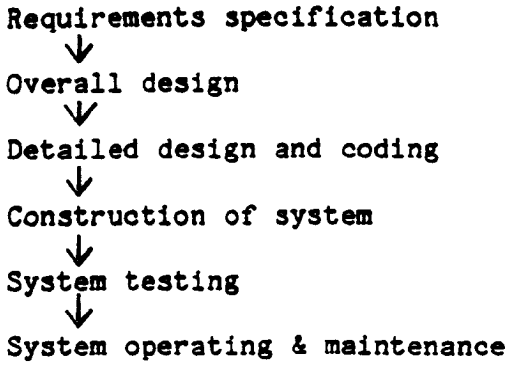


Figure 2-2. Software Life Cycle - Open Loop.

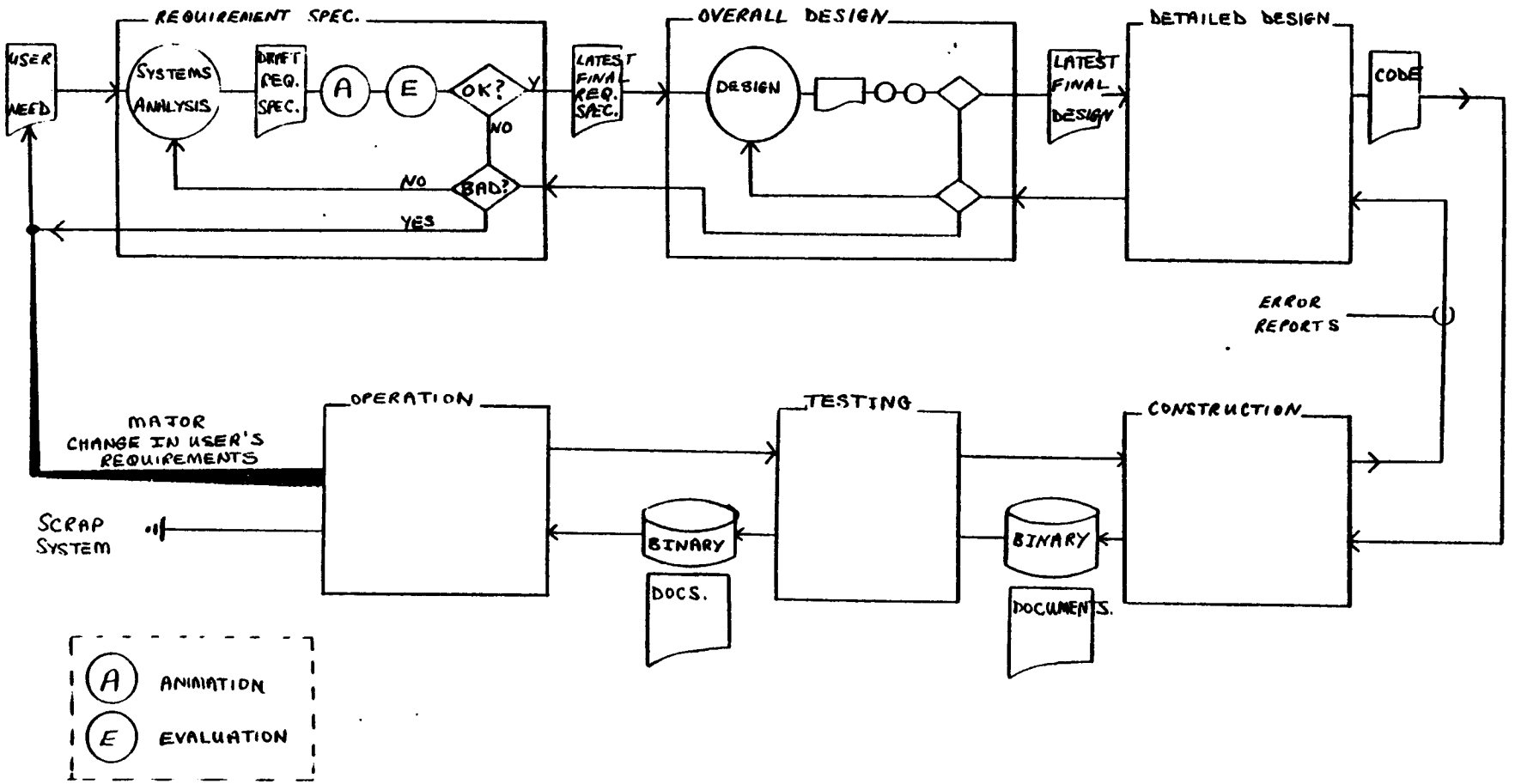


Figure 2-3. Software Life Cycle - Closed Loop.

2.3.2. Requirements Specification

The Requirements Specification phase of the Software Life Cycle is the analysis of the user's or customer's problem to produce a general statement of the proposed system's anticipated inputs and outputs, their functional relationships and the system's behavioural and performance constraints. This stage is crucial because, as Hoare[33] remarks, "It is characteristic of engineering that the problems which it undertakes are never clearly defined to begin with. It is the duty of a good engineer to elucidate the problem, not only to himself but to his customer. He must do this successfully right at the beginning of the project. If he fails or makes a mistake at this stage, the true nature of the problem may come to light only on completion of the project. I fear that in computer programming we have perpetrated many such projects."

A Requirements Specification is often a bulky, ambiguous document, written in English. Current research is aimed at finding a more concise, formal representation (eg SADT[67] and algebraic specification[28]) and automating the task of verifying the correctness and consistency of the specification (eg PSL/PSA[75] and AFFIRM[54]). The Michigan research has produced a working PSL/PSA system which tries to

1. produce comprehensive data and function dictionaries
2. perform static network analyses to ensure the completeness of derived relationships
3. perform dynamic analyses to indicate time-dependent relationships between data
4. analyse volume specifications.

The result of these analyses should be an error free statement of the problem to be tackled.

PSL/PSA (Problem Statement Language/Analyser) helps to *evaluate* and *animate* the Requirements Specification, see figure 2-3. When any component part of a software project (design, code, test result) has been produced it must be evaluated to check that it meets its required targets (cost, response time, correctness etc). This is true of both open- and closed-loop development. In the closed-loop case the recently produced part must then be animated and shown to its customer to give him the necessary insight into the project's direction and progress (cf Hoare's remark about "elucidation" above). For example, a Requirements Specification may first be evaluated for correctness, completeness, consistency and projected cost. It may then be animated, by some kind of simulation technique or verbal explanation, for presentation to the customer, in terms the customer can understand, so he can sensibly give or withhold his approval of the progress to date.

An architect designing a house may periodically check his customer's requirement by showing him a simple sketch. The civil engineer might build a scale model of a proposed bridge. "It is one of the unfortunate aspects of computer programming that there is no intuitively acceptable method of summarising the major external characteristics of a computer program by means of a two-dimensional picture or a three-dimensional model".[33]

When the customer and the systems analyst have agreed upon a final version of the Requirements Specification the design stage can begin.

2.3.3. Design

Software design is "the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible".[15] A key tool in making a design intellectually manageable is conceptual integrity. For Brooks[9] "conceptual integrity is *the* most important consideration in system design". One way to achieve this integrity is to employ a software architect whose vision

and ability will unify the efforts of a whole project team. As section 2.2 pointed out software architects are in short supply. Suppose a software architect existed and was given a new commission. He would be faced with supervising a whole range of design tasks. To begin with he must design the project itself. He must ask himself how many staff he needs, at which points in the project he needs them and what skills they must have. He must decide how the staff are to work together; will they be organised according to the traditional analyst/programmer split or as Chief Programmer Teams? The architect must establish a plan of work,. such as a PERT network, to determine timescales, costs, critical paths and the required resources and software tools. These organisational decisions must be made on any project, not just a software job.

The architect is responsible for the logical design of the software system, but before this work can begin he must decide on a design methodology, a representation scheme and a documentation technique. He must also decide whether or not a formal proof of correctness will be attempted. He must study the problem. He must instigate exploratory work to help him understand the problem, to choose the best algorithms and data structures. Only after much preparatory work can detailed design work begin.

The architect must also plan the physical construction of the product, choosing the programming language, compiler and operating system, for example. A plan of how to test the system must be created. Acceptance tests and operating procedures must be agreed with the customer. All of these activities have been called Overall Design in the Life Cycle of figure 2-3.

The above description of the architect's role illustrates the fact that software design is much more than the production of program logic flowcharts, which is called the Detailed Design stage in the Life Cycle of figure 2-3 to distinguish it from the many other problems a software architect and his assistants

must conquer. The architect must be involved in all aspects of the project, not only to make decisions and supervise the team's work, but to inspire the team with his vision of the final product, to ensure its conceptual integrity.

During the last ten years the software industry has been inundated with new approaches to software design. This is perhaps symptomatic of the *how* preoccupation common in new born technologies (see section 1.4). The availability of so many approaches has left the industry wondering which ones fit which classes of problem and which, if any, to adopt.

Designers have to think creatively, intuitively, logically, formally and procedurally, all at the same time; as a project progresses only the emphasis shifts. At the outset must come some spark or vision which sets a design in motion. The designer, having obtained an intuitive solution, scrutinises it carefully and documents his conclusions. This process has been characterised as divergence, transformation and convergence. The big problem is broken down into smaller parts, these are solved individually and their sub-solutions reassembled into *the* solution. If only it were so simple!

On examining the nature of design it becomes apparent that there is little agreement on how to describe the design process and/or its product. Peters and Tripp[63] suggest the definition, which fits most methodologies, that "a software design method is a collection of techniques based upon a concept". A representative list might include

Structured Design[87]
Warnier Method[77]
High Order Software(HOS)[30]
Jackson Method[37]
META Stepwise Refinement(MSR)[45]

The creator of each software design method has structured his technique to address the design issue(s) he views as most germane. Each creator holds a different opinion as to which issue this is.

Advocates of Structured Design declare that the key to a successful design is the identification of the data flow through the system and the transformation(s) on the input data to produce, finally, the output data.

The view held by those who lionise the Jackson and Warnier methods is that the identification of the inherent structure of the data is vital and that the structure of the data should be used to derive the structure of the program's executable instructions.

Supporters of High Order Software, HOS, are provided with a set of axioms which must be used to attain success. These axioms explicitly define a hierarchy of software control, wherein control is a formally specified effect of one software object on another. This control applies to resources and input/output data as well as control flow.

Devotees of META Stepwise Refinement, MSR, state that success is assured if the problem is solved several times, each solution being more detailed than its predecessor. MSR is a combination of Mills' top-down design,[51] Wirth's Stepwise Refinement[80] and Dijkstra's level structuring.[17]

Jackson enthusiasts tackle problems whose data structures can be represented as regular expressions. The transformations on these data structures are modelled on the data structures ie the instruction steps are expressed as regular expressions, too, by only using sequences, alternatives and repetitions. A mismatch between an input structure and its target output structure, a structure clash, is resolved by postulating intermediate data structures and transformations and then removing them by 'program inversion', a technique to simulate these phantom files. The restriction to regular expressions has been eased to context-free grammars by Coleman et al.[14] The Jackson method has its own representation scheme too (see section 2.4).

All of the above shiny new design methods are still essentially ad hoc heuristics to guide the designer. They do not necessarily achieve correctness, completeness or conceptual integrity. The principal basis for maintaining conceptual integrity is rigorous design. It was imagined, in computing's early days, that heuristic design methods were sufficient; the possibility of rigorous methods was hardly considered. The lesson has been learnt now but the industry has fallen into a malaise of heuristic thinking in software design and development that will still be painful to cure even after the invention of truly rigorous design techniques.

This section could not close without mentioning the traditional strategies of 'top-down' and 'bottom-up'. Using the top-down approach attention is first focussed on global aspects of the overall system. As the design progresses the system is decomposed into subsystems and more consideration is given to the specific issues (figure 2-4). The concept of backtracking is fundamental to top-down design. As design decisions are decomposed to more elementary levels it may become apparent that higher level decisions have led to an awkward or inefficient modularisation of lower level functions. Thus a higher level decision may have to be reconsidered and the system restructured accordingly.

In the bottom-up approach the designer first attempts to identify a set of primitive concepts and actions. Higher level concepts are then formulated in terms of the basic ones (figure 2-5). System design is thus facilitated by identification of the 'proper' set of primitive ideas. If composition of the existing primitive ideas runs into difficulties the primitive set must be changed by a procedure analogous to top-down backtracking.

In practice the design of a software system rarely proceeds in either a pure top-down or bottom-up fashion, and, indeed, it should be impossible to tell which methodology was used to produce the final design (figure 2-6). It is

only the immature state of Software Engineering today that produces so much emphasis on *how* the designers proceed. The recipients of the finished design, the programmers, do not care (nor should they) about *how*, but they are very concerned about *what* is produced, the final design product itself.

Building architects produce plans, mechanical engineers produce blueprints and electrical engineers produce circuit diagrams. What do software architects and designers produce? What do they pass on to the programmers?

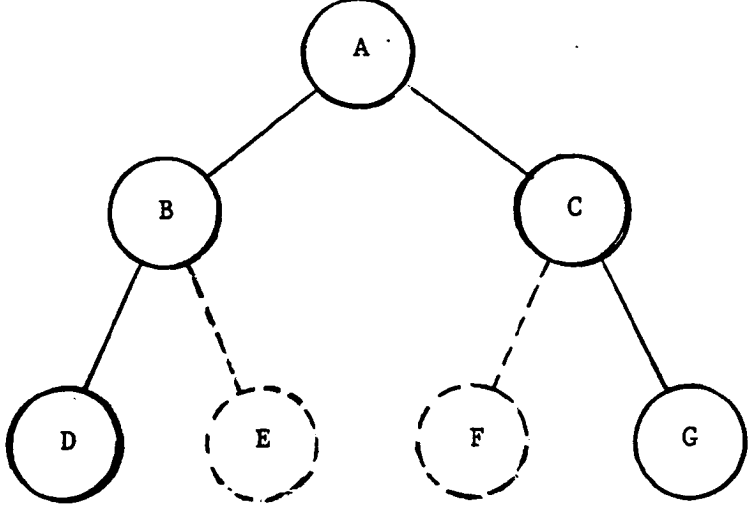


Figure 2-4. Top-down.

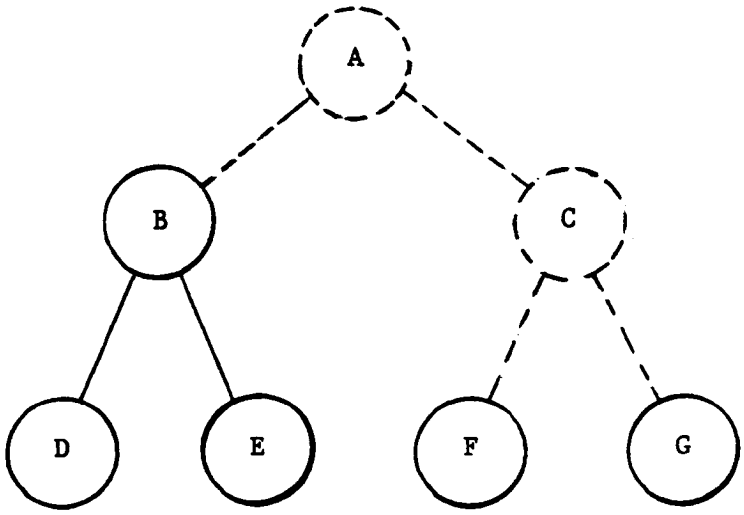


Figure 2-5. Bottom-up.

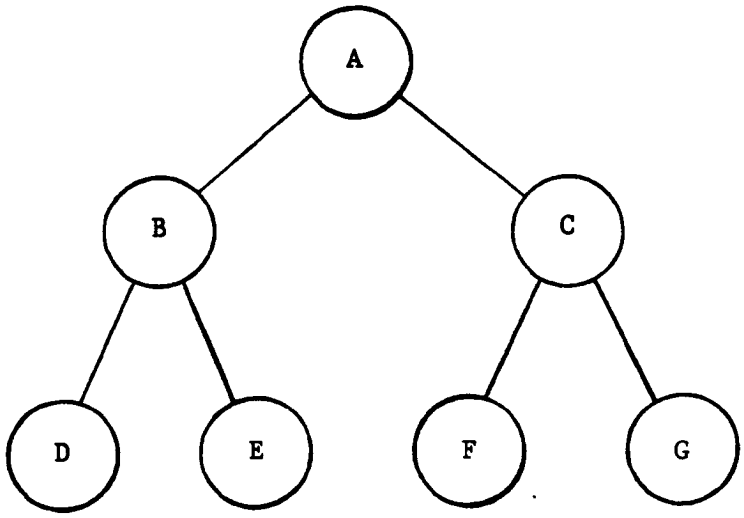


Figure 2-6. Final Design.

2.3.4. Coding

What the programmers receive from the designers is examined, in detail, in section 2.4. Basically they receive a detailed specification of what their 'piece' of the system must do. From this they produce source code, the final refinement of the design. Coding, the ultimate design phase, is considered in more detail in section 7.5.1.

The whole software design process is finally complete when coding is totally finished. However, this is not the last stage as the executable program must be constructed from the source code.

2.3.5. Construction

The construction of an executable binary program is one of the best understood processes in the software industry. Construction has developed from a painstaking human activity of hand loading individual bits into a machine's memory to today's optimising compilers, link-editors and loaders producing binary programs of millions of instructions directly from high level source language programs. The compilation process is fortunate in having a reasonable, scientific basis for its syntax analysis, one of the few successes in Computer Science. However compilers are no more error-free than any other software product.

Developing the process to produce a large binary program, such as a general purpose operating system, is almost a project in itself. It is not usually feasible to recompile all of the source code each time a minor (in textual terms) change is made, so existing compiler output has to be somehow combined with the results of recompiling just the altered portions of the source code, to form a new version. A general purpose software product is often 'tailored' to suit individual customer requirements, causing many different

versions to exist. More versions are created as new releases are developed. Keeping control of a product which exists as a time dependent set of versions (the versions problem) is an outstanding Software Engineering problem. ICL's CADES system[64] is one promising line of research in this area.

Many construction factors influence the binary program's performance. Overlay organisation and the placement of routines and data structures within a virtual memory address space, whilst logically inconsequential, can dramatically affect the system's performance. Thus the production of a software system does not finish with the completion of the source coding, for even the best compiler cannot produce the best binary program from a given source program. The software designer must consider many 'physical' issues as well as 'logical' ones, in exactly the same way that the electronics engineer must consider both the logical circuit design and the physical chip masks, tracking, board layout and power supply.

Production engineering is a well established branch of most engineering disciplines. Quality control, inspection and testing are natural features of any production process. However, in Software Engineering, quality control is still a research topic and inspection is still a rarity because of poor management and programmer neurosis[78] (see section 7.5.3 for further discussion). Testing is an established and widespread activity. Is it effective?

2.3.6. Testing

Software may be tested at four points in the Life Cycle:

1. during construction
2. during initial development prior to the first release
3. during customer acceptance trials
4. during maintenance

The software architect must plan how and when to test his creation. If the software is being constructed 'top-down' then care must be taken to organise the order in which the stubs are expanded and tested. If the system is being constructed 'bottom-up' then the construction of the numerous test harnesses must be scheduled and the integration tests designed.

Testing is only useful if it is undertaken as a scientific experiment performed on a system in order to test the validity of a hypothesis concerning that system. It is normal engineering practice to experimentally confirm that the designer's predictions, based on sound theory, have been realised in practice. If the validity of the theory is in doubt then the whole project cannot be classified as engineering but must be treated as research.

Such a view of testing can be used to test the performance of a software system. Detailed design studies should have been previously undertaken to establish a model of the system's dynamics from the theory on which the design is founded. Experiments can then be sensibly designed to validate the design predictions and find a set of limits within which the system's performance can be accurately interpolated. If experimental evidence does not confirm the design theory then the whole of the design and construction of the system must be examined to resolve the discrepancies. Unfortunately the above procedure is not standard practice in the software industry.

The most widespread reason for software testing is to establish that the system is functioning correctly. There usually is no theoretical basis for this activity. Tests are usually designed around experience, intuition and prayer. The infeasibility of comprehensive 'correctness' testing is contained in Dijkstra's classic remark that "program testing can be used to show the presence of bugs, but never to show their absence!".[15]

Until the software industry eradicates the all too pervasive technique of run-it-till-it-crashes-and-then-try-to-fix-it, there can be no professional respect for the budding software engineer and no peace of mind for his customers.

2.3.7. Operation

Computing's customers have long suffered from the double penalty of being pioneering users in a seller's market. Until the software industry becomes technically more competent and professionally more responsible the customers will continue to pay the heavy costs of the Software Crisis.

It is the customer who operates the final product and so great attention must be paid to the user interface. However it is not the user interface design or the normal day-to-day operation of the system, when all is running smoothly, that is of interest in this thesis. It is the instability of the product, the enormous cost to the customer of the software's failures and inadequacies; this is the crop of weeds that grows from such tiny seeds to choke the software harvest. Ironically both the good seed and the bad are sown by the same men at the same time.

2.3.8. Maintenance

Section 1.3 illustrated how the enormous maintenance overhead on software projects is caused by inadequate design techniques and section 2.2

hinted at how the problem is aggravated by the poor quality of the maintenance work itself.

Software maintenance implies that a program is restored to its original correct state, but maintenance is a misleading term because the program never was correct initially! Maintenance is an unfortunate euphemism for Rectification and Development. Almost all software errors are design errors. Rectification therefore requires design changes. All development work is, by its very nature, making changes to the design. It is thus vitally important that Rectification and Development are designed and implemented just as thoroughly as the initial product. Throughout the whole Life Cycle the one thing which must be maintained is the integrity, conceptual and physical, of the whole design. Software tools are one aid to establishing, checking and maintaining the integrity of the design.

2.3.9. Software Tools

The software industry has been remarkably reticent to utilise the power of the computer to help itself. Many programmers spend a large amount of their time on the routine clerical tasks of manipulating source code and documentation without the benefits of major computer assistance.

The programmer does have his compilers, debug packages and operating systems, but it is most unlikely that the source code he produces is ever checked for any quality other than syntactic correctness; certainly no attempt is likely to be made to check its design quality or conceptual integrity. The software designer, on anything but the most sophisticated project, is unlikely to receive any help at all in the way of tools.

Perhaps one should not be too suprised to find few sophisticated tools in general use when no established methodologies exist to receive computerised

support, nor when any modern design representations are widely enough accepted to be worth the investment.

Maybe this is a good point to ask what is it, this thing that is so inadequate, so difficult to produce? What exactly is a software design?

2.4. What Is Produced?

A software project typically produces an executable binary program (or set of programs), backed up by a comprehensive design, plus documentation to assist the product's users and, sometimes, an adequate Rectification, Development and User Enquiry service. Binary programs and documentation are familiar objects but exactly what is a software design?

2.4.1. Software Design Product

A software design is really the total sum of information about the software product. The design constitutes a communications medium between several groups. The major groups involved are the customers, designers, coders, testers and maintainers. The information they require from the design varies from group to group, and, for any one group, varies with time. Figure 2-7 illustrates the design's user community.

A software system is a very complex object. No one single diagram or text can represent all aspects of its design at once. Similarly no one man can understand all aspects of the system at once. He must study a series of 'projections' of the design, each of which contains the information about one aspect of the design. Figure 2-8 shows some examples of projections, control flow, data flow, fault history etc., which are produced by studying a design from a specific viewpoint. The level of detail required from a specific viewpoint might vary. For example, control flow might be required in simple terms, say as an overall flowchart, or it might be required in great detail, say as the actual

source code instructions.

A projection may be static or dynamic. A flowchart is a static projection of control flow; the instruction address lights on a computer's control panel form a dynamic projection of control flow. The difference between static and dynamic projection is worth emphasising especially when considering the best way to present or *animate* an aspect (see section 2.3.2) in order to better understand or demonstrate its dynamic components.

Considering a design as a single, multi-dimensional, complex object which can only be viewed through a set of projections (ie formally derived simplifications eg 2-D pictures as projections of 3-D objects) enables two things to be explained. Firstly, it helps to explain the importance of conceptual integrity for in trying to understand a complex system from a series of simpler projections the viewer, the user of the design, must rely on the validity of a set of underlying axioms and assumptions which, he hopes, characterise the whole system. This is how he interpolates the role of those aspects of the system which are missing from any given viewpoint. Uniformity of approach is a key tool in managing complexity.

It is an underlying goal of research projects such as PSL/PSA[75] to allow a design to be expressed as a single, complex object from which projections can be formally derived. The realisation of such a goal would be of great help to both designers and developers alike. A much greater benefit would accrue if the reverse process were possible ie if the designer could design a single aspect and then feed this into the 'design object' so that the design was actually built up from a set of aspects.

A software tool which could *analyse* a design to produce a set of projections of various aspects and could perform the reverse process of *synthesising* a complete design from a set of descriptions of individual aspects might be a

line of research worth pursuing if Software Engineering is to cope with increasingly complex systems. LOGOS[66] was perhaps an early attempt at a tool which tried to combine the control flow and data flow aspects.

The second thing that the model in figure 2-8 helps to explain is the multiplicity of methodologies and representations currently in existence. Each representation is an attempt to help analyse a system from a particular viewpoint, to produce a particular projection. Each methodology is an attempt to synthesise a design from one particular viewpoint. It thus seems likely that no one method will emerge supreme but that perhaps one will dominate each aspect if a way can be found to combine the various aspects. Figure 2-9 lists some representation techniques with an indication of the viewpoints they best project. Outline examples of two representational techniques, MSR and Jackson (see section 2.3.3), will now be given as they project aspects of design that will be discussed in greater detail in succeeding chapters.

When using META Stepwise Refinement (MSR) the designer starts with a simple general solution and builds in increasing amounts of detail at lower levels in the design. MSR requires that the designer actually develops several potential solutions at each level, ultimately discarding all but the best of these. A diagram such as figure 2-10 thus represents the various design alternatives considered. This information is very useful to students of the design. It also allows representation of the development of different versions. MSR designers produce hierarchical, tree-structured programs in which a 'root' module contains an outline of the general image of the program, while lower levels contain increasing amounts of implementation detail (figure 2-11). If separate nodes are refined into identical subnodes then the tree is a redundant notation and could lead to unnecessarily large programs. Thus, to eliminate this redundancy, MSR designers organise their modules into a level-structured program

in which modules at one specific level invoke only modules at the next lower level and never the reverse, as in the T.H.E. style of construction.[17] This is represented in figure 2-12. MSR is thus a design tool for a medium level of detail design work and can represent design alternatives, hierarchical module structures and module level structures.

The Jackson method is based on the analysis of data structures into regular expressions. Jackson has his own 2-D representation for data structures (figure 2-13) which expresses sequences, alternatives, repetitions and hierarchical component structuring. Jackson designers build the program code to match the data structures and so the Jackson technique is for finely-detailed design work. Jackson's representation projects data structures, MSR's projects module structures; both are complementary aspects of design.

In reality the practical answer to the question 'what is a design?' is Source Code. This is the ultimate authority to which all programmers finally turn. All other documents are really aids to understanding the source code. If these other documents are not accurately related to the source code they will be abandoned as worse than useless which is sadly often the case in practice. The source code is the finest level of detailed design. It is not the (executable) product though, as anyone who has run into a compiler bug knows from painful experience. Too often the source code is the *only* component of the total design which ever exists, no attempt ever being made to produce a Requirements Specification or an Overall Design. Perhaps Programming could be defined as the creation and manipulation of source codes in contrast to Software Engineering which is concerned with all aspects of software systems throughout their entire life cycles.

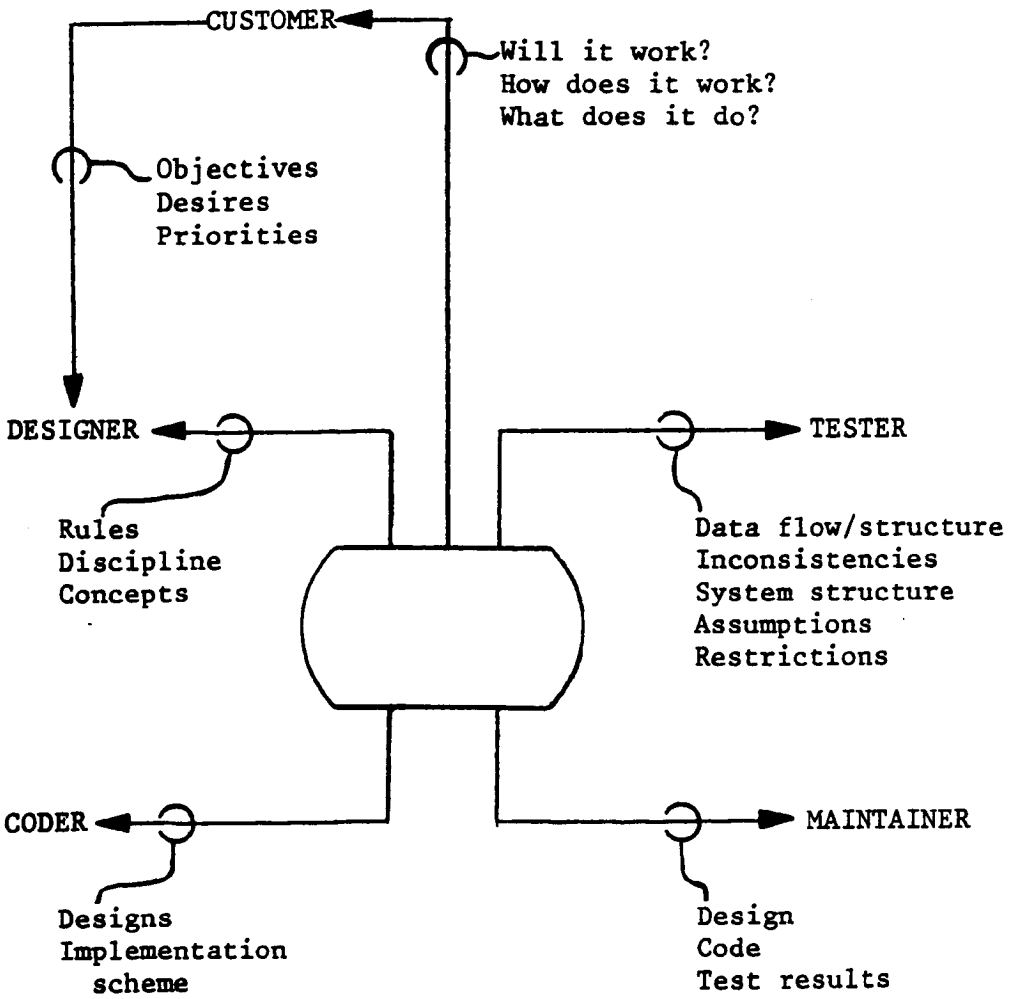


Figure 2-7. Information Flow From Design to Users.

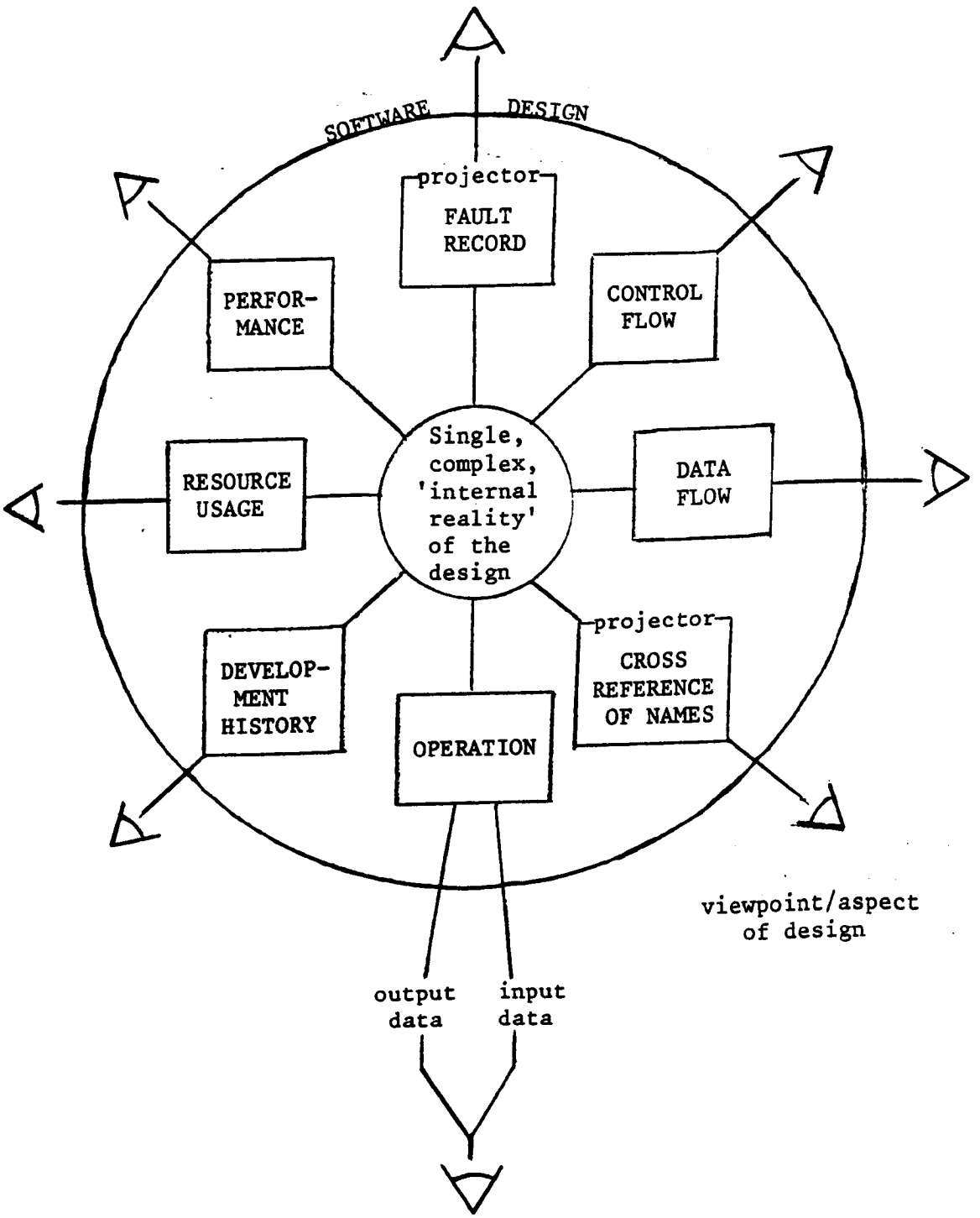


Figure 2-8. The Software Design - Internal & External.

Representation Technique	Projected Aspect
Jackson	DS, CS, H
MSR	H
Structure Charts	H
Dimensional Design	DS, CS, H, CF
Warnier Diagrams	DS, CS, H
Flowcharts	CF, H
Flowcharts ('structured')	CS, H
Data Flow	DF, H
SADT	CF, DF, H
English text	?
Pseudocode	CF, DS, H
ADS	IOD

DS - Data Structure
CF - Control Flow
H - Hierarchy
DF - Data Flow
CS - Control Structure
IOD- Input/Output Document Layout

Figure 2-9. Representations.

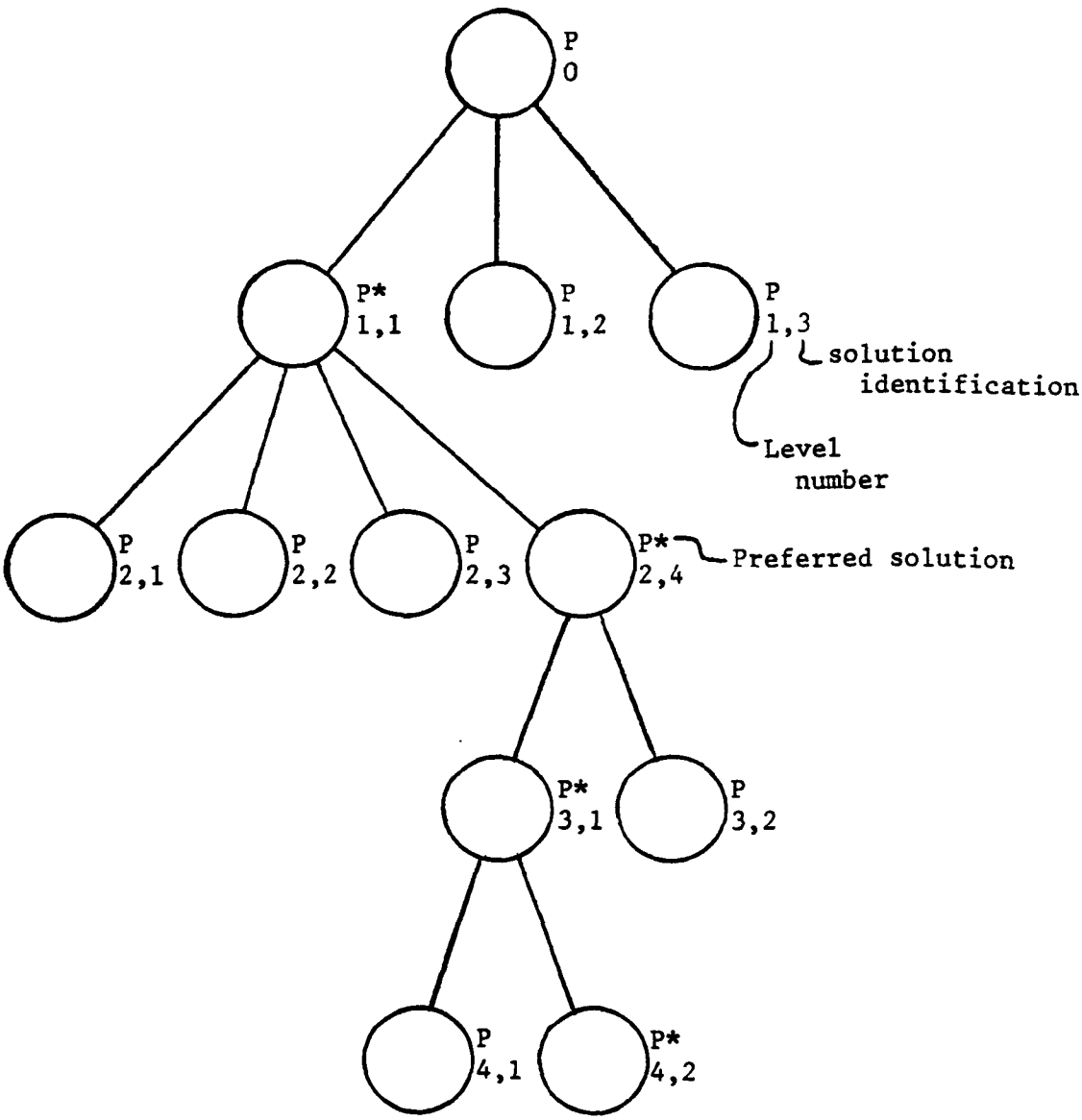


Figure 2-10. MSR Design Alternatives Tree.

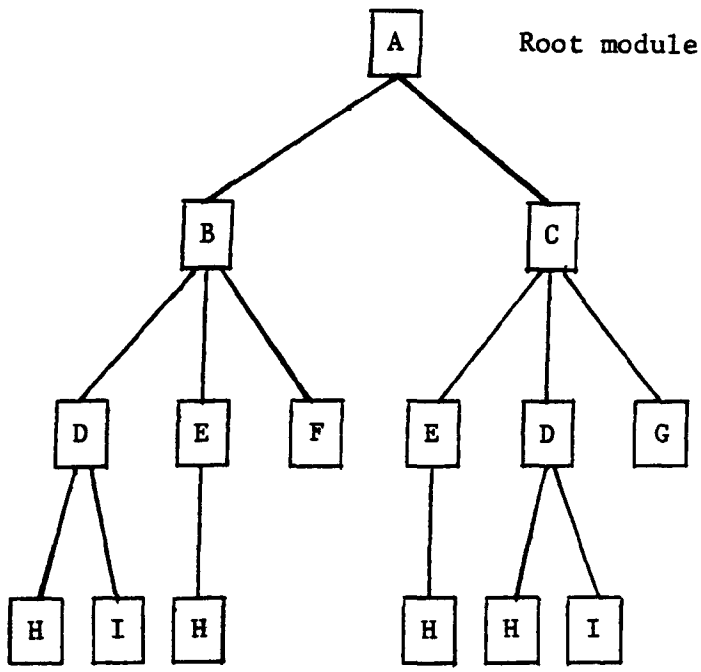


Figure 2-11. MSR Module Tree (redundancy).

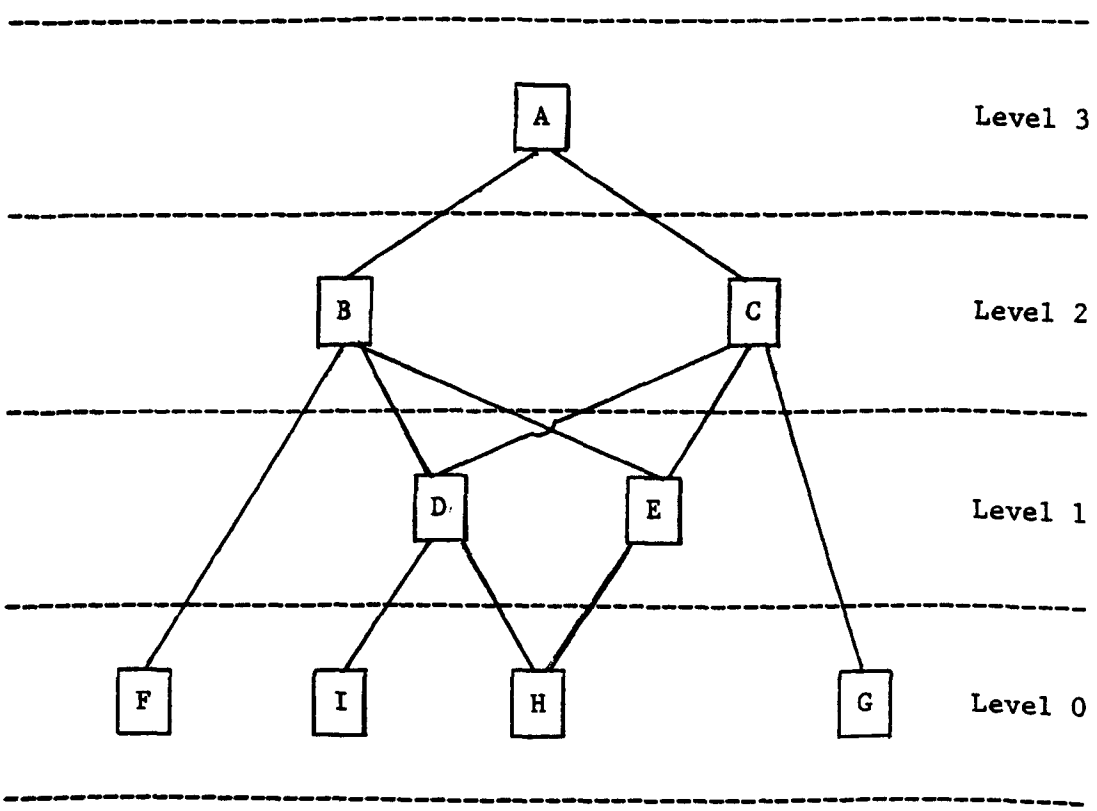


Figure 2-12. MSR Level Structure (no redundancy).

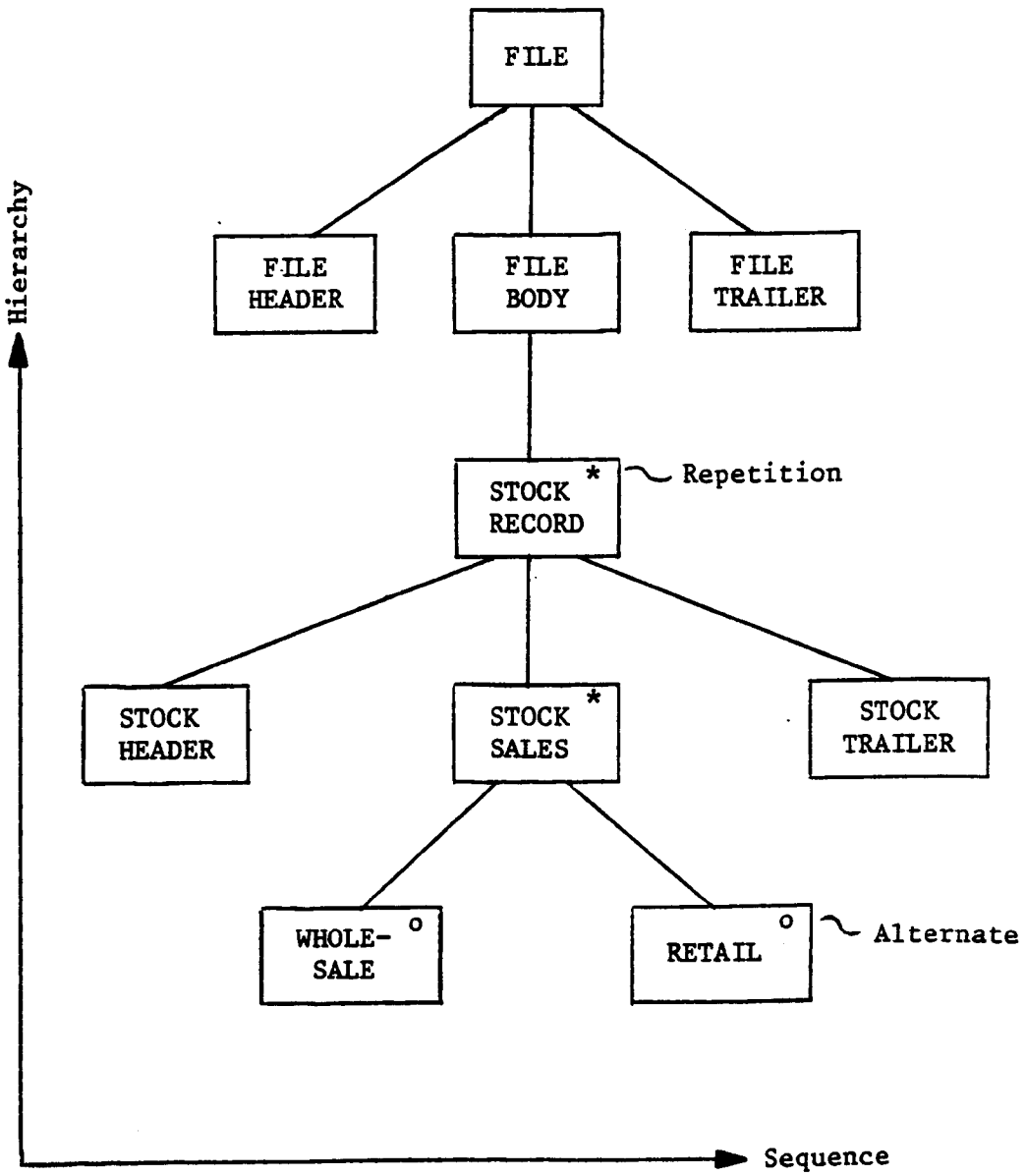


Figure 2-13. Jackson Data Structure.

2.5. Small Scale Software Engineering

The above whimsical survey of the origins of the Software Crisis, the activities enshrined in the Software Life Cycle and the nature of Software Design is hereby concluded with the pessimistic observation that the infant Software Engineering discipline has so far achieved little advance over 'programming' and is still beset by many problems; a thought which admits the optimistic observation that the Software Engineering field is new, exciting and wide open to those who choose to tackle its problems.

The obvious problem with which to begin is the maintenance problem. How can the maintenance programmer be given a design which is not made obsolete by the first change he makes to its source code? How can the maintenance programmer maintain the design as well as the source code? How can his workload be reduced? One answer to the last question is to create a better design product (and hence operational product) initially. If, in general, a product is a set of cooperating programs then is it still difficult to design the individual programs, to properly engineer software on this smaller scale? Yes it is. So, what is currently the best way to produce an individual program?

2.5.1. Structured Programming

Detailed design is the stage in the Software Life Cycle when the source code is produced. The source code might be the final refinement of a module specification or an individual (small scale) program overall design. Currently one of the best ways to produce source code programs is by Structured Programming[15] which is now the generic name for a family of methodologies which produce Structured Programs.

During recent years the computing literature has been swamped by superstition and dogma about what is or is not a Structured Program; for example,

"a Structured Program can be recognised by the presence of five positive characteristics and the absence of two negative characteristics".[12] The major negative characteristic is the *goto* construction, the elimination of which leads to software quality. Like many superstitions before it, *goto* elimination has evolved from a reasonable basis.

Why is truth so often corrupted into superstition? Knuth explains that "there has been far too much emphasis on *goto* elimination instead of on the really important issues (because) people have a natural tendency to set up an easily understood quantitative goal like the abolition of jumps, instead of working directly for a qualitative goal like good program structure...If such *goto* elimination procedures are applied to badly structured programs, we can expect the resulting programs to be at least as badly structured".[42] Dijkstra's original article,[18] which gave Structured Programming its name, never even mentions *goto* statements. Dijkstra is primarily concerned with proof.

2.5.2. Proof

In the paper[18] Dijkstra directs his attention to the critical question "For what program structures can we give correctness proofs without undue labour, even if the programs get large?". By correctness proofs he does not necessarily mean formal derivations from axioms but any sort of proof that is 'sufficiently convincing' for there is no such thing as an absolute proof of logical correctness; there are only degrees of rigour. Dijkstra sees Structured Programming as a deliberate attempt by the programmer to design both the data structures and control structures of his program so that they facilitate the construction of a proof of correctness. Structured Programming is thus a proof-directed methodology. Dijkstra states his case as follows[19] "The first message is that it does not suffice to design a mechanism of which we hope that it will meet its requirements, but we must design it in such a form that we

can convince ourselves - and anyone else for that matter - that it will, indeed, meet its requirements. And, therefore, instead of first designing the program and then trying to prove its correctness, we develop the correctness proof and program hand in hand. (In actual fact, the correctness proof is developed slightly ahead of the program; after having chosen the form of the correctness proof we make the program so that it satisfies the proof's requirements). This, when carried out successfully, implies that the design remains 'intellectually manageable'. The second message is that, if this constructive approach to the problem of program correctness is to be our plan, we had better see to it that the intellectual labour involved does not exceed our limited powers".

What form does a correctness proof take? Hoare,[32] describing the axiomatic technique says that "Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms". Axioms are a formal statement of the elementary operations from which a program may be constructed.

One well known proof technique is the method of Inductive Assertions.[59,26] This method regards a piece of program text as a process which progresses a computation from an initial state to a final state. These states are characterised by assertions about the states, usually expressed in the first order predicate calculus. Using axioms and inference rules an argument is constructed to show that the program text turns the initial state into the final state when executed (the operational view) or that the final state can be achieved, given the program, if a particular initial state exists (the predicate transformer view). Such arguments or proofs are usually split into two components, one

showing that the initial state is transformed into the final state assuming the computation terminates (a proof of correctness mirroring the execution of the code) and the other component proving that the computation does indeed stop (proof of termination). If proof-directed design and construction is employed the proof of termination can be made very simple and formally convincing.[3]

"It is a deplorable consequence of the lack of influence of mathematical thinking on the way in which computer programming is currently being pursued, that the regular use of systematic proof procedures, or even the realisation that such proof procedures exist, is unknown to the large majority of programmers. Undoubtedly this fact accounts for at least a large share of the unreliability and the attendant lack of overall effectiveness of programs as they are used today. Historically this state of affairs is easily explained. Large scale computer programming started so recently that all of its practitioners are, in fact, amateurs... However a reaction is bound to come. We cannot indefinitely continue to build on sand" wrote Naur in 1966.[57] Unfortunately his words are still true thirteen years later. However program proving must in time, if the industry is to mature, become a major technique for ensuring reliability because program proving tries to show the *absence* of bugs as opposed to program testing which can only show their *presence*.

Program proving is also a useful documentation technique because the initial and final state descriptions and assertions demonstrate *what* the program does, something simpler to understand than the source code itself which shows *how* the objective is achieved. A program proof shows *why* the program behaves as it does in achieving the final state. Why a program does something is always the most difficult question a maintenance programmer has to tackle; usually this question can only be extracted from the 'how' story of the source code after much hard, tedious detective work. A simple view of documentation

(and the product of detailed design work) is that the program specification, assumptions and assertions show *what* a program does, the source code shows *how* a program does it, and the proof shows *why* the source code is so organised.

Proofs thus aid the designer to build more reliable systems by removing errors at their source. Proofs help the program's developers by supplying otherwise unavailable information about *what* and *why*. A proof helps developers to design and assess the ramifications of potential changes, and if each time a change is made the proof, changed accordingly, remains valid, then an increased reliability of development activities will ensue.

Today, whenever a programmer writes a section of code he tries to convince himself, usually very informally, that it will work. When writing a small section of code to perform a common, well understood function, such as table searching, he *just knows* it is correct. This is an unconscious appeal to a 'theorem' proven by experience. What the software designer desperately needs is a set of formally proven theorems about useful programming tasks. "There is no set of theorems of the type illustrated above, whose usefulness has been generally accepted".[15] Today's proofs are long, tedious and impractical for non-trivial programs. Theorems would be one way to reduce this burden for "the length of the proof ... is a warning that should not be ignored ... let us be honestly humble and interpret the length of the proof as an urgent advice to restrict ourselves to simple structures whenever possible and to avoid in all intellectual modesty 'clever constructions' like the plague".[15] This is the essence of Structured Programming and the key tool which can reduce the length and complexity of proofs and increase the reliability of software is Abstraction.

2.5.3. The Principle of Abstraction

Abstraction is one of the most powerful, most general tools available to help the human intellect master complexity. One understands a complex thing by recursively breaking it down into successively simpler parts and understanding each of the parts and how they fit together. Thus there exist, simultaneously, different levels of understanding and each of these levels corresponds to an *abstraction* of the lower level details. For example, at a given level of abstraction one can manipulate an integer without considering whether it is held in a machine as binary coded decimal digits or a two's complement bit string. However, in certain circumstances, this lower level of detail may become important. At a higher, more abstract level the precise value of the integer may be irrelevant, the important thing being its role as, say, a running total of input records, one of the instances of the general abstraction 'counters'.

The power of abstraction, in the software environment, lies in the separation of 'what a thing does' from 'how it works' because *what* is simpler to understand than *how*. Using the above example it is much simpler to understand *what* the integer does (represent the number of records read so far) from *how* it does it (result of a monotonically increasing integer function held as a two's complement bit string in RAM constructed out of TTL...). The process of abstraction is involved in *naming* an operation and using it on account of what it does rather than how it works. There is a strong analogy between using a named operation in a program regardless of how it works and using a theorem regardless of how it has been proved. Even if the theorem's proof is highly intricate it may be a very convenient theorem to use. If subroutines could be invoked as theorems in a proof then correctness proofs would be greatly simplified. In the same way that one theorem or routine may invoke

another so understanding by abstraction naturally generates many levels of detail. Knuth[42] feels that a talent for programming largely consists of the ability to switch readily between the microscopic and macroscopic views of things ie to change levels of abstraction fluently. In the same paper Knuth quotes Hoare as defining Structured Programming as "the systematic use of abstraction to control a mass of detail, and also a means of documentation which aids program design".

Abstraction, according to Hoare,[15] consists of four stages:

Abstraction: The decision to concentrate on properties which are shared by many objects or situations in the real world, and to ignore the differences between them.

Representation: The choice of a set of symbols to stand for the abstraction; this may be used as a means of communication.

Manipulation: The rules for transformation of the symbolic representations as a means of predicting the effect of similar manipulation of the real world.

Axiomatisation: The rigorous statement of those properties which have been abstracted from the real world, and which are shared by manipulations of the real world and of the symbols which represent it.

Which abstractions may be "systematically used" by a software designer to "control a mass of detail"? The following chapter discusses eight abstractions (Sequence, Set, Hierarchy, Hierarchical Reduction, Integration, Induction, Enumeration and Generation) which have "properties shared by many (programs) in the real world". Further, these eight abstractions may be represented (Chapter 4) and manipulated (Chapters 5 and 7), but above all, they may be axiomatised (Chapter 6); therefore their use might lead to

provably correct individual programs and a possible step forward in solving the Small Scale Software Crisis.

2.5.4. The Small Scale Software Engineering Problem

Before investigating, in detail, the use of the above mentioned programming abstractions it might be worthwhile to define a little more closely the Small Scale Software Crisis which they might help to overcome. Chapter 1 outlined the large scale or general Software Crisis which came about because software tends to be delivered late, over budget, unreliable and costs a fortune to rectify and develop. Smaller, individual programs also tend to suffer a similar fate.

Chapter 2 outlined the Software Life Cycle through which all software projects progress even if small scale, individual programs pass through each stage less formally than their larger counterparts. Both Chapter 1 and Chapter 2 frequently highlighted the unacceptably high cost of maintenance or rectification and development of software, which runs at around 50% of total cost, as being the major symptom of the Software Crisis which needed curing.

Both rectification and development depend crucially on the quality and difficulty of modification of the design itself. So the Small Scale Software Engineering Problem amounts to

1. the *poor quality* of the initial design product which necessitates rectification and development
2. the *intractability* of the design product which makes rectification and development difficult and expensive

How can the programming abstractions of Set, Sequence, Hierarchy, H-Reduction, Integration, Induction, Enumeration and Generation be used to improve the form in which a software design product is realised so that the

design product helps solve the Small Scale Software Engineering Problem?

CHAPTER 3. DIMENSIONAL DESIGN: ABSTRACTION

OUTLINE

Chapter 1 introduced the Software Crisis. Chapter 2 outlined Software Engineering's current practice in terms of the Software Life Cycle before focusing down to the small scale software engineering problem of improving the quality and tractability of the designs of individual programs. Structured programming and correctness proof were helpful techniques whose utilities were based on the principle of abstraction. Chapter 2 closed with the hint that eight specific abstractions could form the basis for an attempted solution to the small scale software engineering problem, an attempt named as Dimensional Design by Chapter 3.

Chapter 3 introduces each of the eight abstractions, set, sequence, hierarchy, h-reduction, integration, induction, enumeration and generation, showing examples of how their application can lead to a reduction in the size and complexity of the software design product, to the natural production of structured programs, to a reduction in the difficulty in reasoning about the design product and an increase in the quality of the design product. The key to these improvements lies with the representation of the design product.

Dimensional Design is a new representational technique which explicitly records each use of the eight abstractions, allowing easy visual recognition of their occurrences and scopes. Dimensional Design is informally introduced in Chapter 4: Representation. Succeeding chapters discuss its manipulation, axiomatisation, use in actual software production and comparison with

existing techniques.

3. DIMENSIONAL DESIGN: ABSTRACTION

The concepts of Set, Sequence, Hierarchy, H-Reduction, Integration, Induction, Enumeration and Generation have been employed by programmers and hardware engineers right from the beginning of modern computing. The application of these concepts to programming has often been intuitive, but the modern trend is towards a more formal treatment due to the greater discipline of structured programming. These concepts are *abstractions* and are discussed below in terms of the four aspects of abstraction, as seen by Hoare,[32] namely:

Abstraction .....(this chapter),
Representation.....(Chapter 4),
Manipulation.....(Chapter 5),
Axiomatisation .....(Chapter 6).

The eight concepts were successfully abstracted and adapted for computing many years ago; they have been formalised by, for example, Set Theory and Predicate Transformers; they have been successfully manipulated by every programmer, consciously or unconsciously and they have been represented by a variety of techniques. These eight programming abstractions have been chosen for an attempted solution to the small scale software engineering problem (see section 2.5.4). This attempted solution has a name: Dimensional Design.

Dimensional Design is a representational technique (informally explained in Chapter 4: Representation) which tries to improve the quality of the initial design product by making every use of each of the eight programming abstractions an explicitly obvious feature of the design product itself. Being easily manipulatable by both hand and software tools, Dimensional Design tends to reduce the intractability of the design product which makes rectification and development so difficult (see Chapter 7: Practical Experience). The Dimensional Design representational technique is more formally described in Chapter

6: Axiomatisation; but first, in this chapter, the eight programming abstractions themselves are described and their practical use illustrated to justify them as a reasonable basis for an attempt to solve the small scale software engineering problem.

The aim behind choosing this set of abstractions is to improve the quality and tractability of the design product by reducing the complexity of the software. To illustrate how these abstractions achieve this goal a simple model of computation is introduced which is then refined, by exploiting the abstractions, into the modern structured programming model.

3.1. Set & Sequence

The classical von Neumann computer consists of a store and a processing unit. The processing unit can execute a sequence of instructions and each instruction can act upon the contents of the store. From this simple model two fundamental abstractions can be extracted.

Set: The von Neumann computer's store can be considered as being an (ordered) set of objects which coexist simultaneously.

Sequence: A program executed by a von Neumann computer can be considered as being a time-ordered sequence of instructions. No two instructions can ever exist (ie be executed) simultaneously.

Computational 'time' denotes the operational concept of computational progression and is the axis of the computational history. Execution of an instruction transforms one set of objects, the initial state of the store, into another set of objects, the final state of the store. This process is represented by the computational history in figure 3-1a.

Consider a von Neumann machine simplified in the following three ways:

1. its storage for instructions is totally separate from its storage for data;
2. its instruction repertoire only includes instructions which operate on its data store and excludes any instruction which operates on its program store or program counter;
3. its program counter is always initially *one* and is incremented by one on completion of each instruction. The value of the program counter is increased until the computation is stopped by either executing an explicit stop instruction or by the program counter exceeding the limit of the instruction store. Therefore termination is guaranteed.

Such a restricted von Neumann computer (RVNC) can only execute totally sequential, non self-modifying programs. Thus a general model for all RVNC computational histories looks like figure 3-1b. Executable programs for this restricted machine are identical to their computational histories with the details of the store contents removed ie the static representation of the program corresponds exactly to the dynamic sequence of instructions executed by the processing unit.

To understand a program written for such a RVNC one need only execute it 'mentally' ie starting from the initial state, mentally simulate the execution of each instruction in turn, generating the intermediate store states until the final state is produced. To prove the correctness of such a program one could regard the program as a predicate transformer and, reversing the sequence of instructions, show that this 'inverse program' produces the initial state from the final state. Proof of termination for such a program is simply 'restriction 3'.

Understanding or proving a program as above, by mentally processing each and every instruction in turn, is called Enumerative Reasoning. A typical contemporary, useful program might run for an hour on a machine performing

a million instructions per second, thereby executing around 3,600 million instructions. If a man could reason or simulate perfectly at the rate of one instruction per second, it would take him about 750 man-years of effort to prove the correctness of such a 3,600 million instruction program. Obviously something must be done to reduce this impossible burden of Enumerative Reasoning. Section 2.5.3 hinted that trying to understand a program at different levels of abstraction might be attempted. The separation of understanding into various layers suggests a hierarchical arrangement of levels of detail, so can the concept of a Hierarchy help to reduce the complexity of understanding and proving a program?

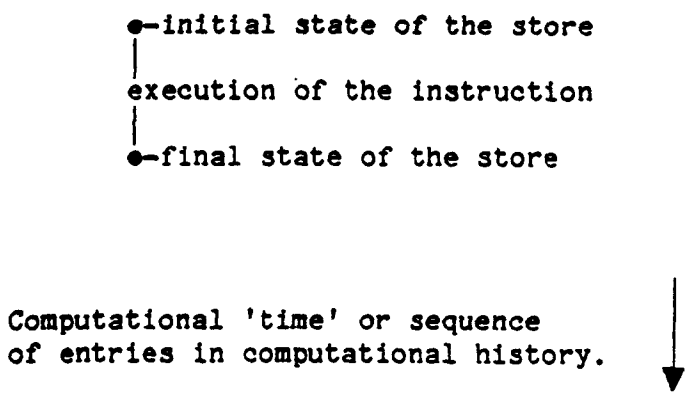


Figure 3-1a. Computational History.

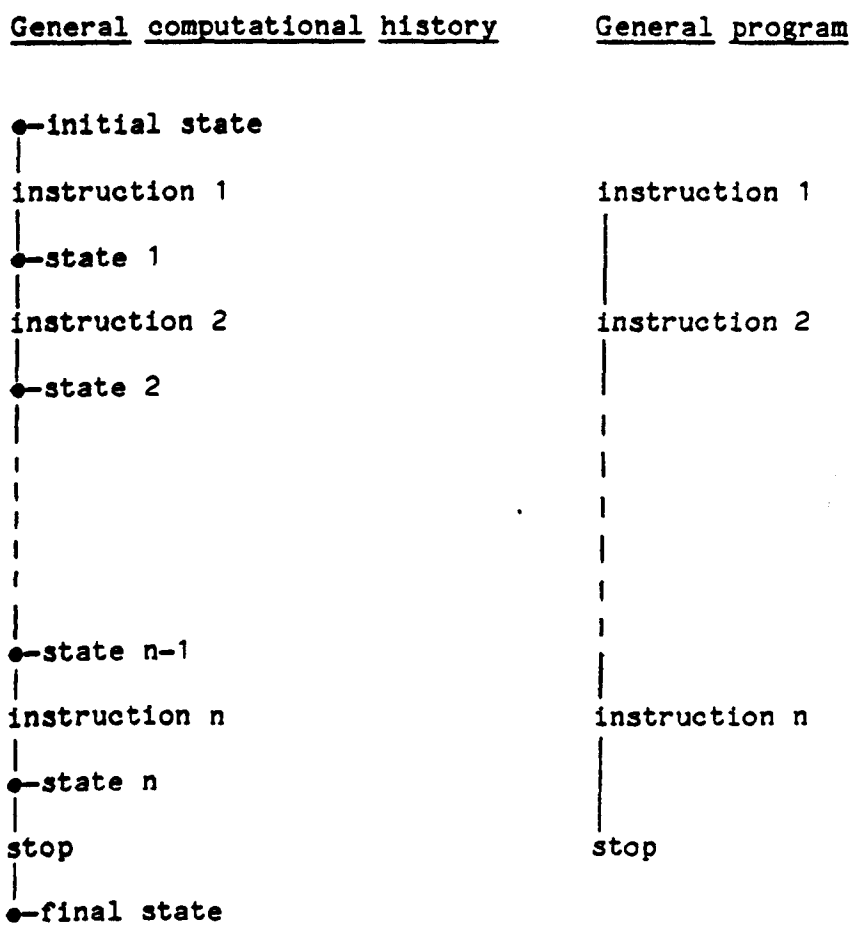


Figure 3-1b. Restricted von Neumann Program.

3.2. Hierarchy

A hierarchically structured system generally means a system that is composed of interrelated subsystems, each of the latter being themselves hierarchic in structure. In a finite hierarchy some lowest level of elementary, indivisible subsystems must exist. In his classic paper on hierarchies in both natural and artificial worlds, Simon[73] observes that "it is somewhat arbitrary as to where we leave off the partitioning, and what subsystems we take as elementary. Physics makes much use of the concept of 'elementary particle' although particles have a disconcerting tendency not to remain elementary very long". An applications programmer considers the statements in a high level programming language to be elementary - until he encounters a compiler bug!

When a system is said to be composed of subsystems the very word 'subsystem' implies a subordinate relationship and the original meaning of Hierarchy comes from the organisational structure of a priesthood. Hierarchies are thus built from three things; the first is the set of parts or subsystems, the second is the relationship(R) between a system and its subsystems and the third is the relationship(T) between the subsystems. Parnas[62] more formally defines a hierarchy as a structure for which there exists a relation or predicate R(A,B) on the pairs of subsystems such that levels may be defined where

1. Level 0 is the set of subsystems, A, such that there does not exist a B such that R(A,B) (the elementary subsystems).
2. Level i is the set of parts, A, such that
   a) there exists a B on level i-1 such that R(A,B);
   b) if R(A,C) then C is on level i-1 or lower.

From this definition a hierarchy is a structure with a relation R between

its subsystems only if the directed graph representing R has no loops. Hence, for example, figures 3-2a and 3-2b are hierarchies in which R is 'is invoked by'.

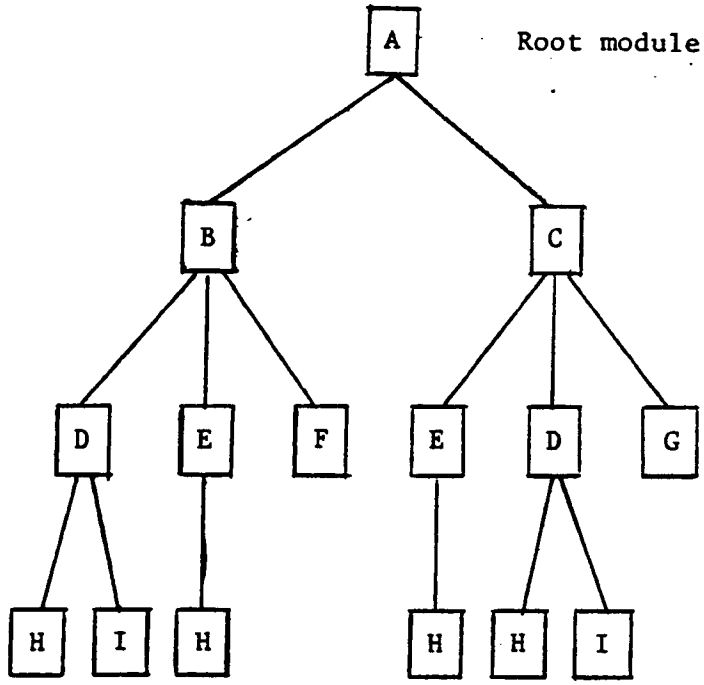
The *Span* of a system is the number of its constituent subsystems, ie if R(A,B) then the Span of A is the cardinality of B. For example in figure 3-2b the Span of A is 2, of B is 3, of E is 1 and of H is 0.

The *Depth* of a subsystem is the number of levels it is below the top most level. The maximum Depth is the number of levels from the topmost level down to the elementary subsystems (4 in figure 3-2b).

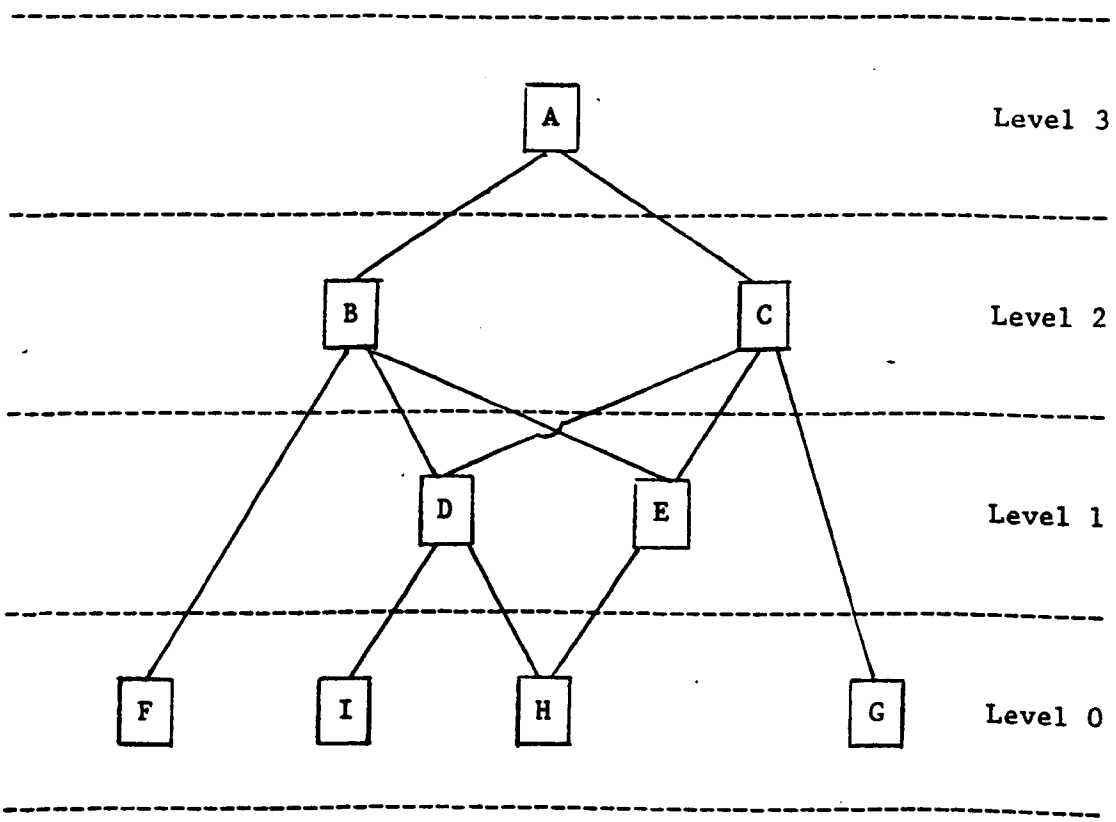
As Parnas remarks, the statement "a system is hierarchically structured" is meaningless because *any* system can be represented as the trivial hierarchy having one part and one level. More importantly, it is possible to divide *any* system into parts quite arbitrarily and contrive a meaningless relation such that a system appears to be hierarchically structured. Before such a statement can convey any valid information at all, the way that the system is divided into parts and the nature of the relationship between them must be specified.

The introduction of hierarchical structuring into the world of software brings at least two advantages:

1. Hierarchy facilitates human understanding by:
   a) quantitatively reducing the description of a program;
   b) quantitatively reducing the necessity for Enumerative Reasoning.
2. Hierarchy increases the quality and capability of software.



(a) MSR Module Tree (redundancy) [Figure 2-11].



(b) MSR Level Structure (no redundancy) [Figure 2-12].

Figure 3-2. Example Hierarchies (taken from Chapter 2).

3.3. Reduction of Description

3.3.1. Hierarchical Reduction

Consider the realistic example program running on the RVNC mentioned in section 3.1 which consisted of 3,600 million sequential instructions. It can be considered as a hierarchy of two levels, the first level being the program name and the second level being its instructions. Such a hierarchy has a depth of 2; the span of the level 1 program name is 3,600 million and the span of each level 0 instruction is 0. The relation between level 1 and level 0 is 'is the n^{th} instruction of'. Figure 3-3 is a somewhat smaller program illustrating these points. This is an example of a trivial hierarchy. However, a real machine only has a few hundred different types of instructions in its repertoire so a program of 3,600 million instructions must be made up from many instances of each instruction type. Figure 3-3 shows that although example program 1 contains 36 instructions it is a combination of only 12 different varieties. Repeated subsequences exist and are identified in figure 3-4. Figure 3-4 is a more complex description of the program because it contains more information. However this description can be compressed by eliminating redundant subsystem definitions as in figure 3-5. The technique of reducing the size of a description by eliminating redundant subsystem definitions is called Hierarchical Reduction, normally shortened to *H-Reduction*. (See section 4.3.1 for further details of the Dimensional Design representation of H-Reduction.) Note that though figure 3-5 is a more compact description, it no longer shows the *real* program, but a more abstract, though still complete derivative. To see what the program really looks like figure 3-3 must be *generated* from figure 3-5 by expanding figure 3-5 to restore the redundancy (figure 3-4) and then removing the *artificially introduced* hierarchy.

If a complex structure contains absolutely no redundancy - if no aspect of its structure can be inferred from any other - then it is its own simplest description. Luckily, software for the RVNC will usually contain massive redundancy. Figures 3-3, 3-4 and 3-5 illustrate the fortunate fact that the RVNC shares with more natural systems the following properties, identified by Simon[73] as:

1. Hierarchic systems are usually composed of only a few different kinds of subsystems, in various combinations and arrangements.
2. By adopting a descriptive technique which allows the absence of something to go unmentioned, a nearly empty 'world' can be described quite concisely.
3. By appropriate 'recoding', the redundancy that is present but not obvious in the structure of a complex system can often be made visible.
4. By adopting a generative descriptive technique the redundancy in a complex system can be eliminated from its description thereby simplifying it.

A description which *explicitly* represents all of the components of a system is henceforth called an Enumerative description. One which implicitly represents some or all of the components is called a Generative description because work must be done to generate the Enumerative description from the Generative version.

It is important to realise that the quantitative reduction of an Enumerative description to a Generative representation is achieved by constructing an artificial hierarchy 'on top of' the system being described. This is analogous to simplifying a geometric proof by adding a geometric construction to a diagram. The process of regenerating the Enumerative description is similar to macro-processing. The final product is still a 'flat' sequence of say 3,600 million instructions. *It is the description which is being modified not the system it*

describes.

Example Program 1

```
B
D
A
N
P
M
B
D
A
N
P
M
S
U
R
I
K
H
S
U
R
I
K
H
B
D
A
N
P
M
B
D
A
N
P
M
```

Figure 3-3. 36 instructions.

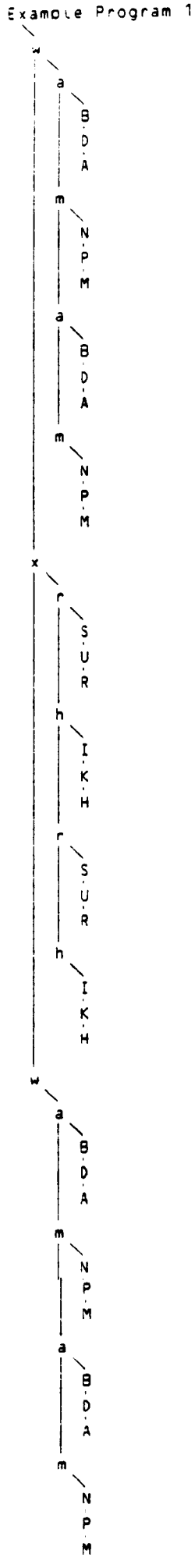


Figure 3-4. 1+3+12+36=52 Symbols.

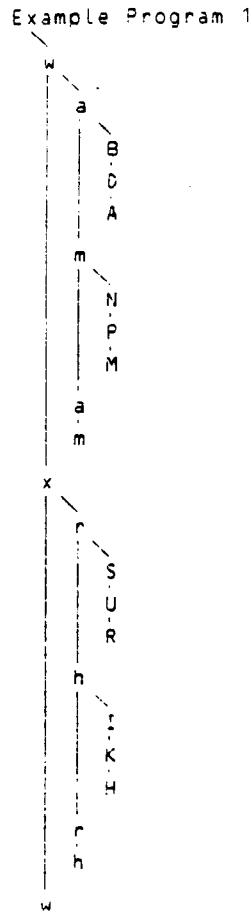


Figure 3-5. 1+3+8+12=24 Symbols.

3.3.2. Integration

There are other ways to simplify the description of an RVNC program. The next technique to be illustrated is Integration. Suppose there exist two programs (figure 3-6) whose individual descriptions cannot be reduced as they contain no redundancy. When considered as a pair of programs they can be seen to have identical subsections. It is possible to produce a single description of the two programs by enumerating, once only, their similarities and combining their differences (figure 3-7) such that at each point of difference, the required alternatives are enumerated and marked with an appropriate *Selection* criterion, the name of the original program in this example (figure 3-8). Thus the descriptions of the two independent programs have been *Integrated* to reduce their combined descriptions from 24 instructions to 17 instructions and 2 selections. (See section 4.3.2 for further details of the Dimensional Design representation of Integration.) Again note that regeneration will produce only one of the two possible programs depending on which selection criterion is used; only the descriptions have been integrated not the actual programs. This process is similar to conditional compilation.

It was possible to integrate example programs 2 and 3 because of their common structure and purpose; their similarities outweighed their differences. Figure 3-8 is typical of a whole family of programs to calculate functions of three numbers. Can the descriptions of all of these programs be integrated easily? Of course the answer is yes - this time Parameterisation is the tool.

Figure 3-7 may be re-worked to describe a whole family of functions of three numbers as in figure 3-9 where 'func' is a formal parameter to be used during the regeneration of the Enumerative description. The 'select function' subsystem allocates an actual value to the formal parameter. Parameterisation helps eliminate some of the selection operations occurring at points of

difference in integrated descriptions. It is an optimisation exploiting the frequent occurrence of identical selections. Parameterisation is an implicit, symbolic mechanism. The description of the two example programs has now been reduced to 16 instructions, 1 selection and 1 parameter.

The program in figure 3-9 may be re-worked again to produce example program 6 in figure 3-10. Such a rearrangement produces different but equivalent programs in the sense that they compute the same functions as figure 3-9. The rearrangement allows even more redundancy to be removed by a two step process of first parameterising the variable name in the subsequence 'Read; func; Print' to produce figure 3-11 and then eliminating the redundant subsequence definitions as shown in figure 3-12.

It is a simple job to modify example program 8 in figure 3-12 to calculate functions of 4 numbers; just add an 'RFP(P=4)' subsystem after 'RFP(P=3)'. Similarly programs for functions of 5,6,7 etc are easily produced. Such programs exhibit another form of redundancy - the repeated juxtaposition of identical subsystems. This redundancy could be reduced by the Hierarchical Reduction technique of creating a new subsystem containing, say, two RFP subsystems and then replacing all pairs of RFPs by the new subsystem and so on. However the special property of contiguous redundancy can be exploited by a more powerful tool - Induction.

```
Example Program 2 - SUM
  Read num1
  Read num2
  Read num3
  Total := 0
  Total := SUM(num1,Total)
  Total := SUM(num2,Total)
  Total := SUM(num3,Total)
  Print num1
  Print num2
  Print num3
  Print 'SUM='
  Print Total
```

```
Example Program 3 - PRODUCT
  Read num1
  Read num2
  Read num3
  Total := 0
  Total := PRODUCT(num1,Total)
  Total := PRODUCT(num2,Total)
  Total := PRODUCT(num3,Total)
  Print num1
  Print num2
  Print num3
  Print 'PRODUCT='
  Print Total
```

Figure 3-6. 12+12 Instructions.

```
Example Progs 2 & 3 Integrated
  Same:Read 3 numbers
  ????:Calculate (SUM or PRODUCT) function
  Same:Print 3 numbers
  ????:Print name (SUM or PRODUCT) of function
  Same:Print answer
```

Figure 3-7. Integration.

```
Example Program 4 - SUM and PRODUCT
  Same:Reads 3 numbers
    Read num1
    Read num2
    Read num3
  Diff:Calculate (SUM or PRODUCT) function
    SUM                                  PRODUCT
    Total := 0                           Total := 1
    Total := SUM(num1,Total)             Total := PRODUCT(num1,Total)
    Total := SUM(num2,Total)             Total := PRODUCT(num2,Total)
    Total := SUM(num3,Total)             Total := PRODUCT(num3,Total)
  Same:Print 3 numbers
    Print num1
    Print num2
    Print num3
  Diff:Print name (SUM or PRODUCT) of function
    SUM                  PRODUCT
    Print 'SUM='         Print 'PRODUCT='
  Same:Print answer
    Print Total
```

Figure 3-8. 17 instructions + 2 selections.

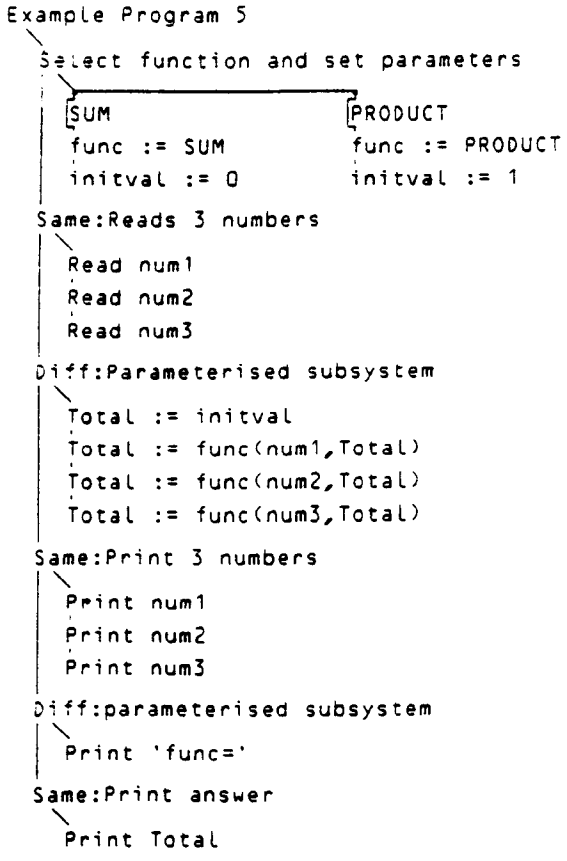


Figure 3-9. 15 instructions + 1 selection + 1 parameter.

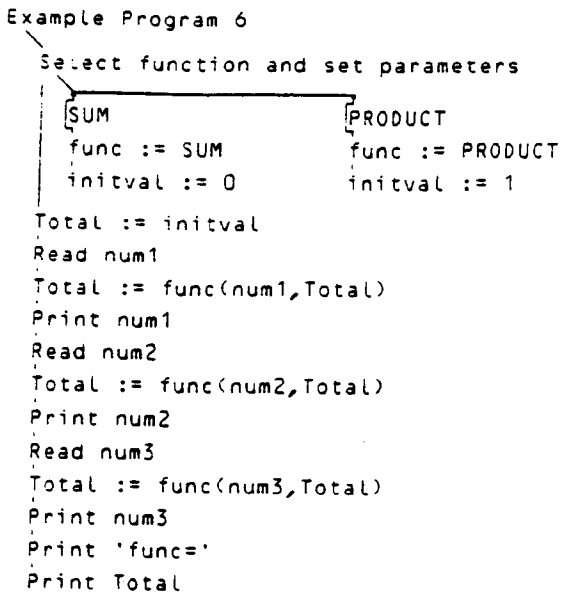


Figure 3-10. 16 instructions + 1 selection + 1 parameter.

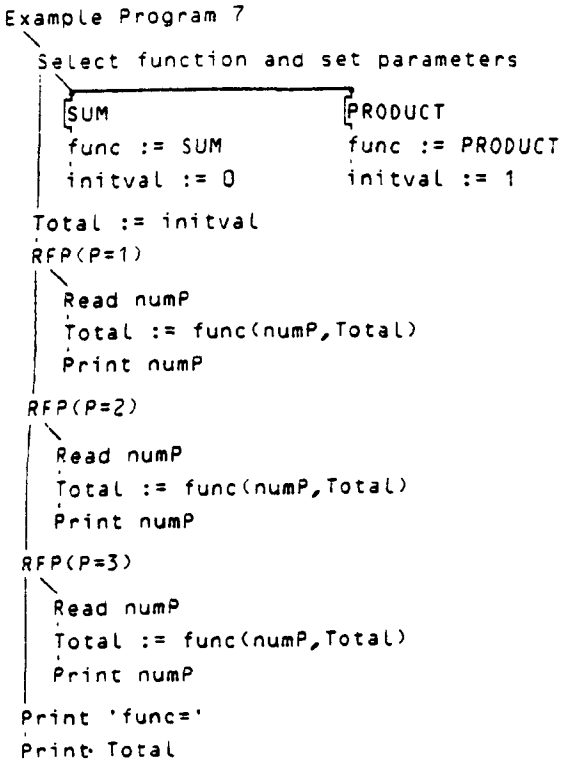


Figure 3-11.

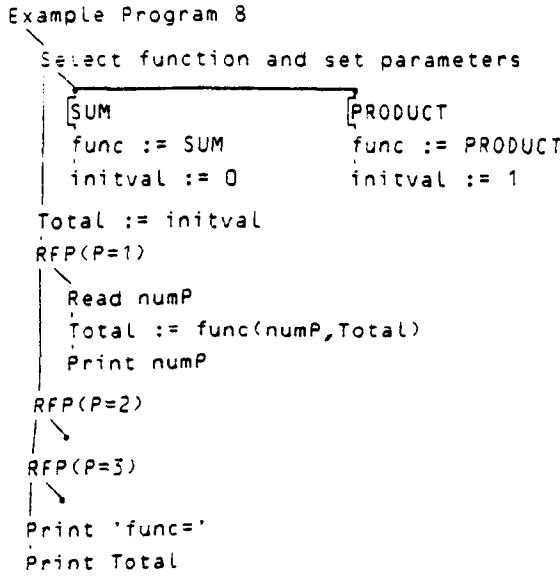


Figure 3-12.

3.3.3. Induction

Induction is the Generative technique to reduce multiple copies of a subsystem to a single, generative instance. It is the most powerful tool to be illustrated. Induction may be used to generate either a finite or an infinite number of copies of a subsystem. Induction has two manifestations, Iteration and Recursion.

Iteration produces an infinite number of *contiguous* copies of a subsystem. This is the tool needed to reduce figure 3-12 still further. Figure 3-13 shows that the Iteration produces a sequence of copies of the RFP subsystem, that P is a parameter of the Induction, and that the Selection mechanism is employed to limit the otherwise infinite reproduction to just three copies of RFP. If the specific limit of three were to be replaced by a parameter N (N>0), then figure 3-12 could generate a whole family of programs to calculate a function of a sequence of N numbers. (In figure 3-3 '*' represents an iteratively generated, infinite subsequence whose description has been removed. See section 4.3.3.2 for further details. The '$\overline{-}$' indicates the end of a subsequence and acts as a (redundant) contrast to '*' - see section 4.2.2.)

Iteration is an optimisation of a special case of Recursion, Recursion being the more general of the two inductive techniques. The Recursive formulation of the example is shown in figure 3-14. (In figure 3-14 '▣' represents a recursively generated, infinite subsequence whose description has been removed. See section 4.3.3.1 for further details.) The generations of the (identical) Enumerative descriptions by both types of induction are animated in figures 3-15 and 3-16 (see also section 3.5.1.3). The Iterative generation requires more of its reader/generator because the reader must replace 'Iterate' by an infinite sequence of copies of the subsequence '?P<=3?; RFP(P); P:=P+1' and then resolve the newly generated Selections to give the required number of

instances of the subsequence. The Recursive generation is simpler because it only utilises the familiar substitution-to-regenerate-redundancy process employed by H-Reduction (see section 3.3.1). The power of Recursion is achieved through the *self-referential definition of* the 'Recurse' subsystem. This reproductive mechanism can be used to generate much more complicated objects than the simple repetitive structures produced by Iteration (see section 3.5.1.3).

Induction is the last of the description-reducing techniques to be presented. Like its predecessors it is a very useful abstraction based on a commonly occurring phenomenon. The above mentioned abstractions reduce the size of a program's Enumerative description by eliminating the redundancy fortuitously present in most RVNC programs. The reduction is achieved by constructing an artificial, hierarchically structured Generative description. The relation of the hierarchy is R='is generated by', the elementary subsystems are the RVNC instructions and the non-elementary subsystems are built from the Generative abstractions which may be summarised as follows:

Hierarchical(H)-Reduction

Reduction: name and enumerate one instance of common subsystem. Only name all other instances.

Generation: completely enumerate all instances.

Integration

Reduction: combine descriptions of similar but non-identical programs. Eliminate common subsystems, retain differences as Selectable alternatives.

Generation: Reconstitute individual program by selection of appropriate alternate; discard others. Alternatives can be explicitly chosen by *selection* or implicitly by *parameterisation*.

Induction:

Reduction: Exploit multiplicity of identical subsystems. Avoids need for huge hierarchical elimination structure. Capable of parameterisation to integrate family of programs.

Generation: Use *iteration* to generate contiguous multiplicity or *recursion* for more complex cases.

The Generative description technique may obviously be used to describe systems other than sequences of RVNC instructions. For example, the fact that the elements of the RVNC's data store are all, at a given time, equal in value to zero would be more conveniently expressed inductively rather than enumeratively, especially if the store contains thousands of elements.

Having demonstrated at length that hierarchical structuring, aided by h-reduction, integration and induction can quantitatively reduce a program's state and instruction sequence descriptions, the discussion is now focussed on the second benefit attributed to the exploitation of the abstraction Hierarchy,

the quantitative reduction in the requirement for Enumerative Reasoning.

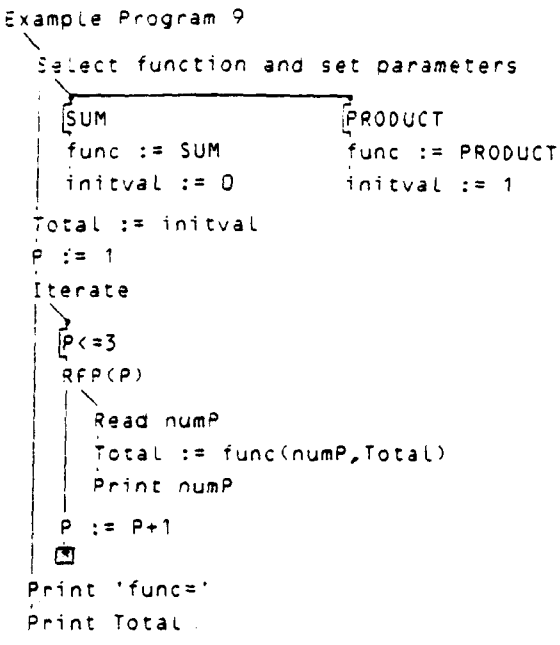


Figure 3-13.

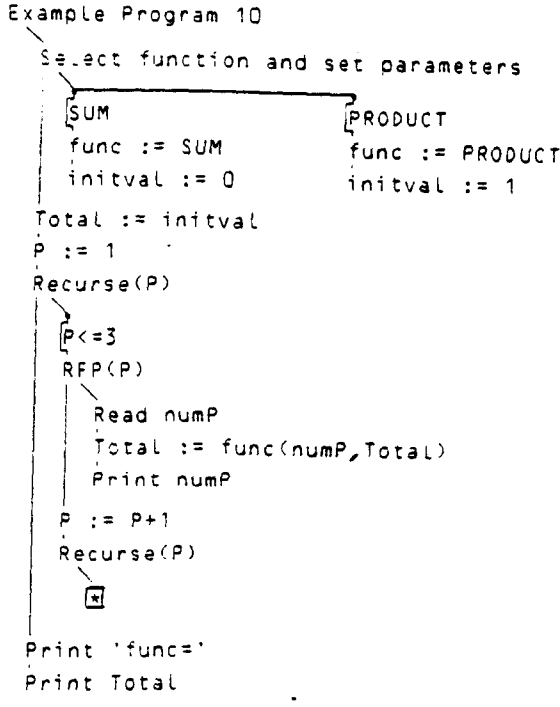


Figure 3-14.

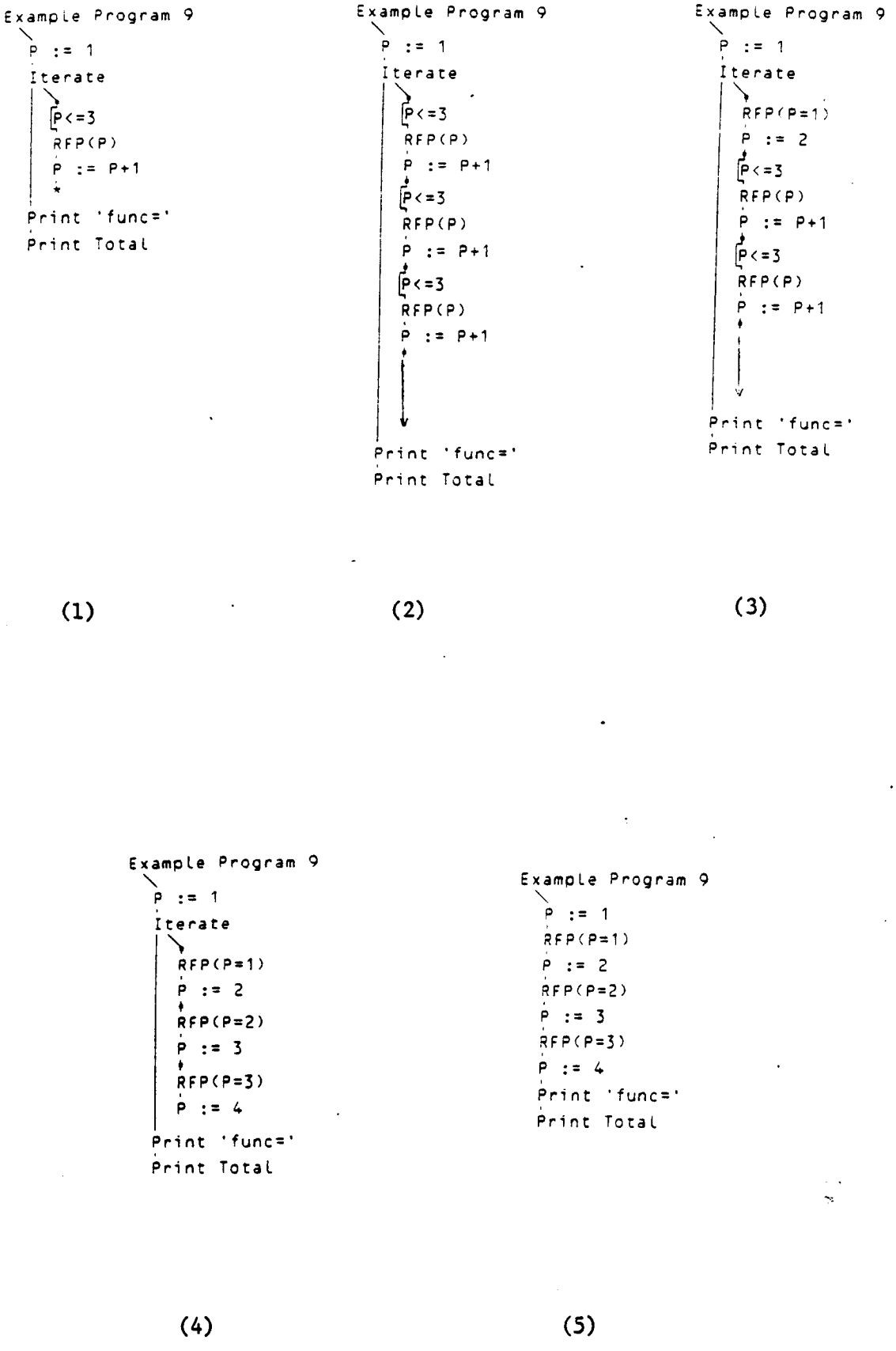
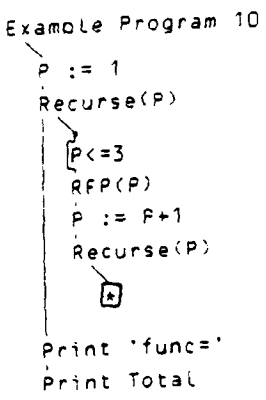
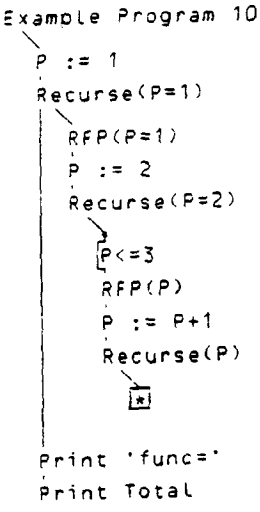


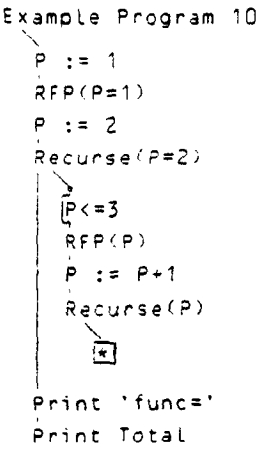
Figure 3-15. Iteration Animated.



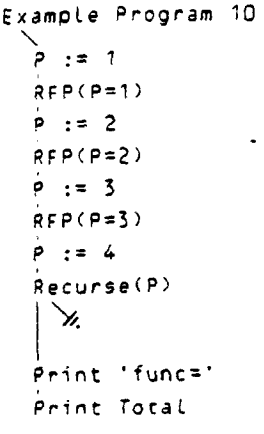
(1)



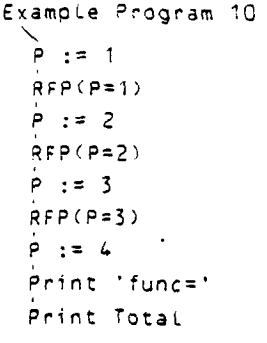
(2)



(3)



(4)



(5)

Figure 3-16. Recursion Animated.

3.4. Reduction of Enumerative Reasoning

Enumerative Reasoning is the understanding or proving of a program by mentally processing each instruction in turn, thereby simulating the RVNC's execution of the program. If a program contains N instructions then the magnitude of the task of understanding the program will be O(N). For a typical useful RVNC program N is 3,600 million, $O(10^9)$, requiring well over 750 man-years of mental effort (see section 3.1). The way to reduce the scale of Enumerative Reasoning, not too suprisingly, closely parallels the technique for reducing the size of the Enumerative Description. To cope with complex systems a hierarchy of understanding is constructed.

If a program is represented by a hierarchically structured description then the program can be understood one subsystem at a time. This is the famous 'divide and rule' principle. It seems unlikely that any intellectual tool will ever enable a detailed understanding of a sequence of 10^9 instructions to be achieved, but if a program can be described hierarchically then it is possible to understand the whole system generally and any nominated subsystem more specifically. It is as though the human mind, having a finite capacity for understanding, can choose one of two alternatives when faced with a program whose detailed understanding exceeds the mind's capacity. The first alternative is to ignore successive levels of detail until the program is shrunk, by abstraction, to fit the available intellectual capacity. The second alternative is to understand, in complete detail, successive parts of the system - a classic space-time tradeoff!

The key to limiting Enumerative Reasoning by a hierarchical approach is the relation R which governs the hierarchy. Any valid, but arbitrary, relation R will not reduce the burden. The descriptive relation R= 'is generated by' is not a strong enough relation. The construction of the Generative Description

hierarchy required no understanding of or reasoning about the system being described. All the techniques employed were totally independent of the system being described. The forms of subsystem (figure 3-5) abstracted from the target system were irrelevant. Their objectives were solely the mechanical reduction of the description. The subsystems were never intended to be understood in their own right as abstract descriptions of the target system. This is why the Generative hierarchy was described as *artificial*. Recall that a hierarchy was originally defined as having *three* constituents: the subsystems, the relation between the system and its subsystems(R), and the relation between the subsystems(T). In a Generative hierarchy R= 'is generated by', but what is T, the inter-subsystem relation?

A first approximation for T is that all the elementary instructions are independent of each other; this implies that the program is a perfectly decomposable system. At a purely descriptive level this is true, for the description of any given instruction is independent of all the other instructions. When viewed as an executable program each instruction is not totally independent, for although the individual instructions are immutable and their sequence likewise (RVNC restriction 2), their individual behaviours influence each other implicitly through the changes they make to the state of the data store. There are many natural and man-made systems exhibiting this 'almost-independent' structure and Simon calls them "nearly decomposable systems".[73]

The benefits of nearly decomposable systems are best appreciated by considering the opposite extreme, total inter-dependence. Suppose RVNC restriction 2 was lifted so that programs could be self modifying. To reason about each individual instruction now requires reasoning about all the other instructions simultaneously to encompass the effects of possible modifications. The reasoning burden now becomes proportional to the number of possible

interactions ie $O(N^2)$ which is $O(10^{18})$ for typical programs!

One of the most successful ways to achieve near-decomposability and reduce the reasoning burden is to build a 'function' hierarchy (see figure 3-17).

The governing relation R in such a hierarchy is the 'two-way' compound relation (figure 3-18)

$$R = R_{abs} \text{ and } R_{ref}$$

where

$R_{abs}(Q,P)$=P specifies *what* Q does eg R([A;B],subf1) ie P is the abstraction from Q.

and

$R_{ref}(P,Q)$= Q shows *how* P works (includes generation) eg R(subf1,[A;B]) ie Q is the refinement of P.

In such a hierarchy a sequence of 'what's must constitute the 'how' of a higher level function. This reduces enumerative reasoning because having once understood how, in figure 3-17, 'subf1' produces state S_3 from 'initstate' by way of [initstate; A; S_2; B; S_3] the reader should be able to generate the 'final state' from state S_5 *without* having to generate [S_5; A; S_6; finalstate].

If the effort expended in understanding how the function 'subf1' itself works is U, in applying 'subf1' is A and if 'subf1' is used X times then the total Enumerative Reasoning required is

$$U + A*X$$

If the program did not use 'subf1' but was left 'flattened' then every instruction must be individually understood which would require X*F reasoning effort, where F is the effort to understand one 'flat' instance of the instructions which would have been generated by 'subf1' (One assumes F<U because

'subf1' will be more general, involve parameters etc.)

Therefore the saving in Enumerative Reasoning through using a function

$$= (X*F) - (U + A*X)$$

$$= X*(F - A) - U$$

Therefore the greatest savings are made when

$$(X*F) - (U + A*X) >> 0$$

that is when

1. $(X >> 1)$ ie the function is applied many times.
2. $(A << F)$ ie the effort to apply the function and understand the consequences of its application (A) is much less than the effort to understand its constituent actions and deduce its consequences (F).
3. $X*(F - A) >> U$ ie the effort to understand the function itself (U), as opposed to its 'flat' instances (F), is smaller than the use made of the function.
4. $X*F >> (U + A*X)$ ie unnecessarily complex functions are not needlessly introduced. $X*F < (U + A*X)$ can arise when functions are used instead of scoped comments - see section 7.5.1.1 and reference.[85]

Thus separating the 'what' from the 'how' aspects allows a 'what' to be utilised in an analogous fashion to a theorem, which may be applied without an understanding of its proof (its 'how' aspect) if the appropriate conditions hold, to produce a conclusion. Reasoning about the action of a function or instruction involves reasoning about its associated initial and final states, its appropriate conditions and conclusion. This form of reasoning, too, can be reduced by exploiting hierarchical structuring (figure 3-19).

The functional approach to understanding programs obviously affects their initial design and shows that the purer the functions (no side effects) and the more restricted and well defined their ranges and domains, then the closer a system can come to the ideal of being perfectly rather than nearly decomposable, and the greater is the decrease in the burden of Enumerative Reasoning. If the concept of a Hierarchy is useful in understanding programs, perhaps its companion description-reducing techniques, Induction and Integration, may be similarly helpful.

Induction has its own formal patterns of reasoning eg the method of the invariant assertion. Reasoning about an induction requires an understanding of a specific application of the induction, usually the first, to ensure it keeps the invariant P(1) true, plus an argument to show that, for any i, given P(i) then the i^{th} application of the induction implies P(i+1). Usually a separate demonstration that the induction terminates is also required (see section 4.5), so that the effort to understand an induction, of difficulty H(induction), is say O(3*H(induction)) thus saving O(H(induction)*(multiplicity-3)) units of enumerative reasoning effort, usually an enormous saving. The invariant assertion is also the key to formulating the specification of the function of the instructions generated by the induction so that it can be successfully incorporated into a functional hierarchy.

Integration does not obviously reduce the need for enumerative reasoning, on the contrary, it would seem to increase it! This is because in a program description with alternatives the number of programs represented is, in general, the product of all the alternatives, and every individual selection must be understood by enumerative reasoning (but see section 4.4.1.2). The danger of Integration is shown in figure 3-20. Such code sequences are unfortunately all too common in today's software. They are produced not as the result of deli-

berate Integration but haphazard design thinking and modification. A correct use of Integration can be characterised by a higher level functional description free from alternatives producing a single final state irrespective of the actual alternative selected. Such a construction reduces rather than increases the requirement for enumerative reasoning. Figure 3-21 illustrates the correct use of Integration and should be related to figures 3-6,3-7 and 3-8.

The above discussion has shown how the abstract concepts of Hierarchy, Integration and Induction can be utilised to reduce both the size of an Enumerative Description and the extent of the associated Enumerative Reasoning, thereby facilitating the human understanding of an RVNC program. This is the first of the two advantages claimed for these abstractions. During the above discussion great stress has been laid on the artificial nature of the hierarchies which have always been additional to the program. The program itself has so far been a very 'flat' sequence of instruction (figure 3-22).

The program and its associated hierarchy will now be combined to allow the executable program itself to be hierarchically structured. This example of Shanley Integration[42] will demonstrate the second advantage of hierarchical structuring: that it increases the quality and capability of the executable program itself.

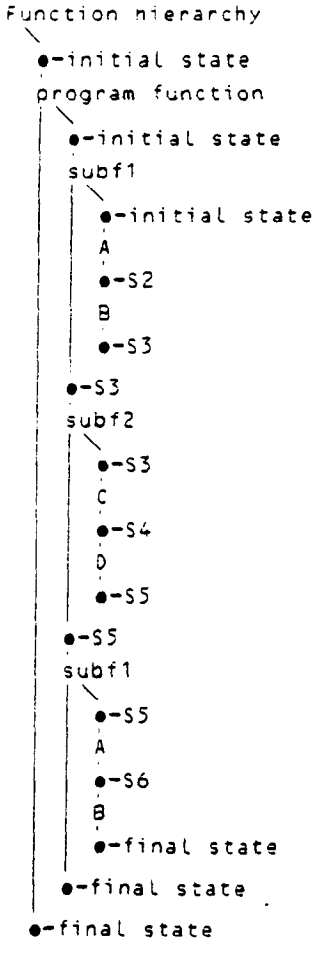
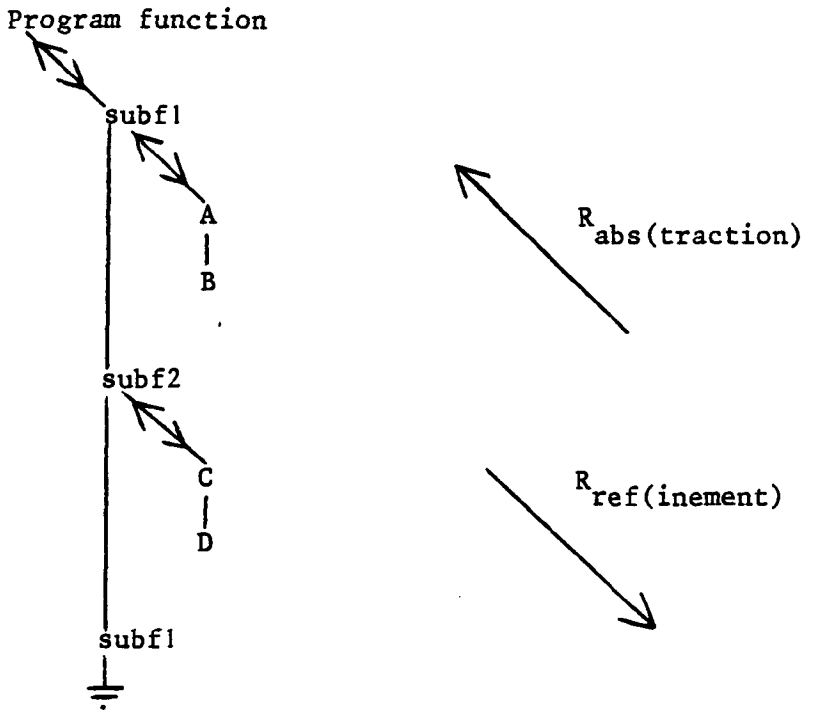


Figure 3-17. Function Hierarchy.



SPECIFICATION OF WHAT THE FUNCTION DOES
(increasing level of abstraction)

HOW THE FUNCTION WORKS
(increasing level of detail/refinement)

Figure 3-18. What-How Hierarchy.

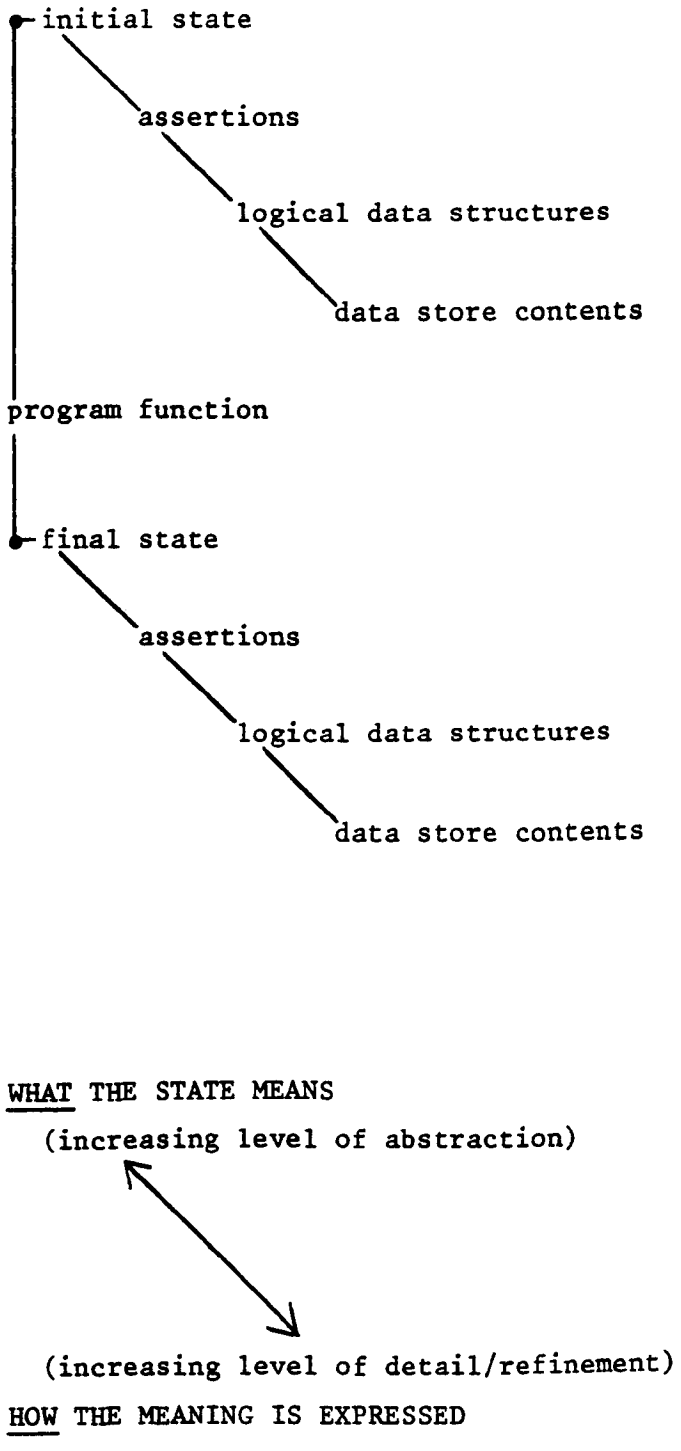


Figure 3-19. Hierarchically Structured State Description.

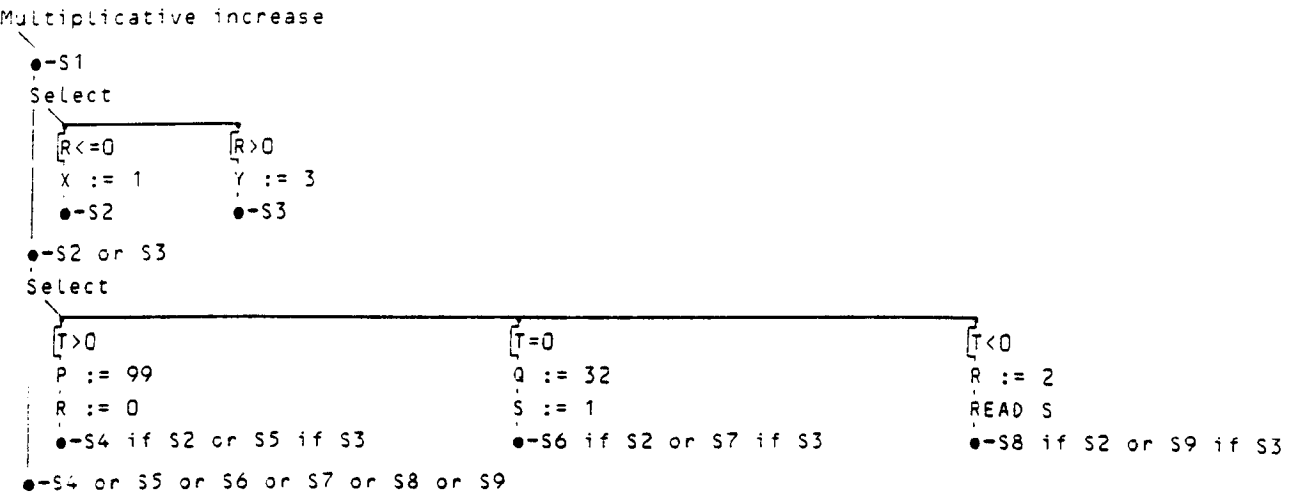


Figure 3-20. 2x3=6 programs: Multiplicative Increase.

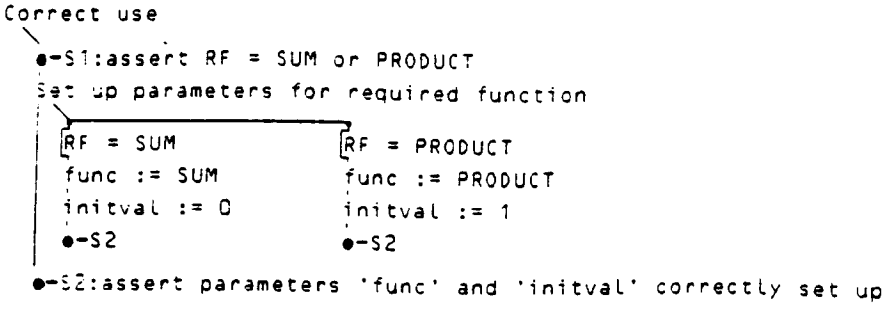


Figure 3-21. Correct use of Integration.

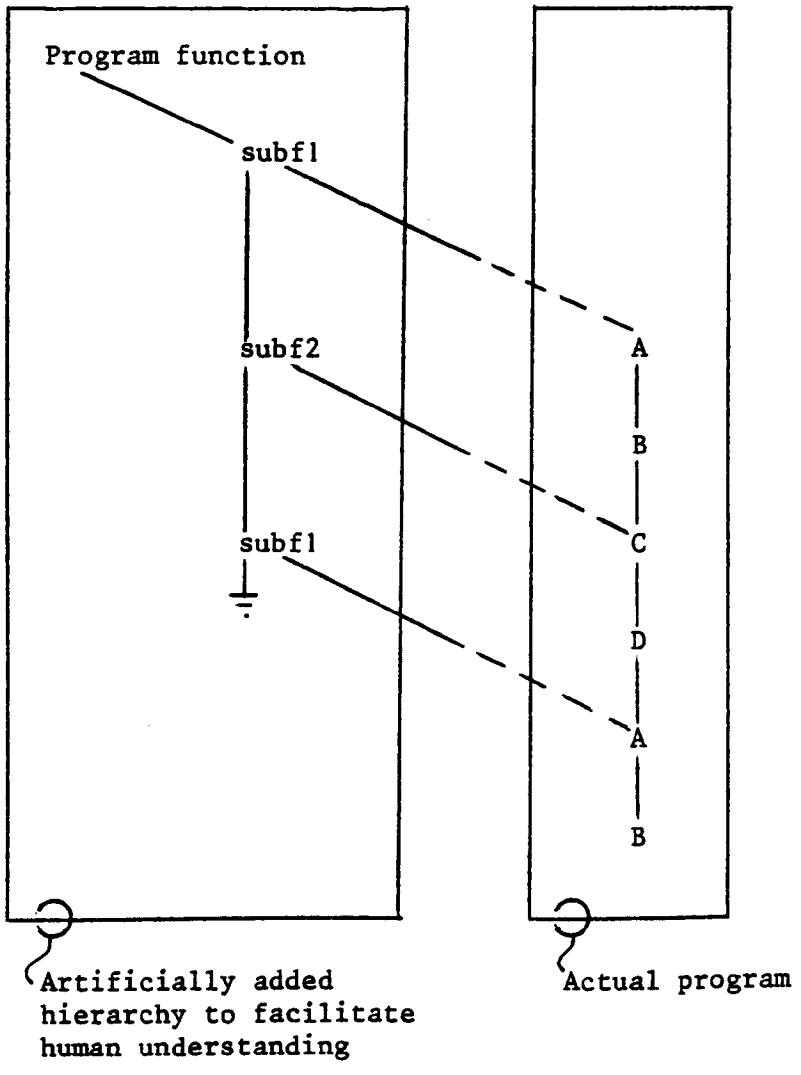


Figure 3-22. Relationahip Between Program and Hierarchy.

3.5. Increase in Quality & Capability

3.5.1. Hierarchical Structuring of the Executable Program

It is now feasible to describe and reason about useful programs. However such programs, still consisting of $O(10^9)$ executable instructions, are unlikely to fit into the RVNC's program store and therefore cannot be executed! Fortunately the same techniques that reduced the program's description and reasoning requirement can be used to reduce its representation in the actual machine. It is one of the great strengths of the von Neumann computer that it is capable of both *generating* its own instruction sequence and *executing* it.

If the two processes of Generation and Execution were distinct then intermediate storage for 10^9 instructions would still have to be found. As only one instruction at a time is executed, the Generation may proceed by outputting one instruction at a time. This interleaving of the two processes allows feedback between them, increasing the generality of the Generation by delaying binding (see section 3.5.4). So perfectly are the Generation and Execution functions integrated in modern machines, that the distinction between them is sometimes not appreciated.

The RVNC cannot generate its own instruction sequence. Its processing unit can only execute directly 'useful' operations. The RVNC must be modified somehow to allow it to trade time for space ie save program storage space by performing the extra workload of generating the program instruction sequence during the execution itself. What increase in machine complexity is required to implement run-time Generation?

3.5.1.1. Hierarchical Reduction

Hierarchically structuring the executable program can reduce its size in exactly the same way as its description is reduced ie by storing the definition

of a subsequence only once and expanding the higher level subsystems into the subsequence as required. The run-time Generation mechanism to achieve this is obviously the closed subroutine.

If the program description is a functional hierarchy and the non-elementary functions are mapped onto subroutine calls in the executable program then the generated sequence of 'useful' executable instructions will be identical to the enumerated version and so a perfect correspondence between program description, proof and execution is maintained.

To implement the subroutine mechanism let the RVNC now include a new instruction 'execute the named subsequence'.

3.5.1.2. Integration

When a set of program descriptions has been Integrated the execution of one particular program involves, at certain points in the program, the Selection of one subsequence from an enumerated set of subsequences according to some selection criterion. The RVNC may handle run-time resolution of Integration alternatives by generalising the subroutine mechanism to 'execute one subsequence selected from an enumerated set of named subsequences according to a given selection criterion'.

The run-time resolution of alternatives does not decrease the size of the executable program; the size is increased because now all the alternatives have to be stored in the executable program. However this increase is a small price to pay for the massive increase in generality and capability which comes from executing an integrated program capable of handling a multiplicity of alternatives and usually the size of an integrated program will be less than the total size of the sum of the individual alternatives (expressed as complete RVNC programs) because of the elimination of identical subsections ie the commonali-

ties should outweigh the differences (see section 3.3.2).

3.5.1.3. Induction

The executable program size may be further reduced by storing any Recursively defined subsequences just once and letting the subroutine mechanism handle the self-referential generation; this naturally generates the required sequence of instructions without the need for further modifications to the extended RVNC. Iteration can be regarded as a special case of Recursion because any Iteration may be re-expressed as a tail-recursion (see figure 3-13) in the executable program so eliminating the need for any special Iterative instructions in the RVNC.

However there are practical drawbacks to such an approach. In general, an enumerative description could be generated from a recursive one by performing the various generative operations asynchronously in parallel, a process which requires, on completion of any operation, a search of the whole description to see if any recursive sub-descriptions are 'eligible' for generation. The conventional von Neumann machine exploits its sequential nature to reduce this complexity to the simple (recursive) subroutine call mechanism which eliminates the need for searches at run-time but requires a way (eg stack) to remember 'return addresses'. Such a mechanism is an optimisation. Iteration, when expressed as tail recursion, permits a further optimisation.

Figure 3-23 illustrates a useful property of tail-recursion, namely that the recursive call is always the last instruction in the sequence so that on completion of a given invocation the sequence is terminated and control passed back to the next highest level (figure 3-23a). This process forms a cascade of 'returns' to the original invocation which can be optimised, as in figure 3-23b, by just retaining a record of the first invocation and throwing away the records for deeper invocations, thereby curing the problem of excessive storage use

when large numbers of 'iterations' (tail-recursions) are performed. In the conventional von Neumann machine Iteration permits a further optimisation whereby the storage and time costs incurred by the need to remember the first 'return address' and perform repeated invocations are reduced by storing, at compile time not run-time, the 'return address' as part of the 'jump' instruction which performs just that part of the general, recursive 'subroutine exit' instruction necessary for iteration to be achieved.

Thus Hierarchy, Integration and Induction may be implemented on an RVNC machine extended by the two following instructions:

1. execute a named subsequence (potentially optimised by forgetting second and succeeding 'return addresses' if tail-recursion);
2. execute one subsequence from an enumerated set of subsequences according to a given selection criterion.

Let an RVNC with such an extended instruction set be called a Generating von Neumann Computer, GVNC. A GVNC program is built exclusively from realisations of the Hierarchy, Integration and Induction abstractions. If each use of each abstraction allows the valid use of the appropriate proof technique (functional composition for Hierarchy, enumerative reasoning producing (higher level) function for Integration and invariant assertion for Induction) then the GVNC program has the following advantages compared to its family of RVNC counterparts:

1. its description is smaller;
2. its proof/understandability is simpler;
3. its capability is greater.

These are the advantages claimed in section 3.2. Such a GVNC program retains a property in common with its RVNC family:

4. it is a Structured Program.

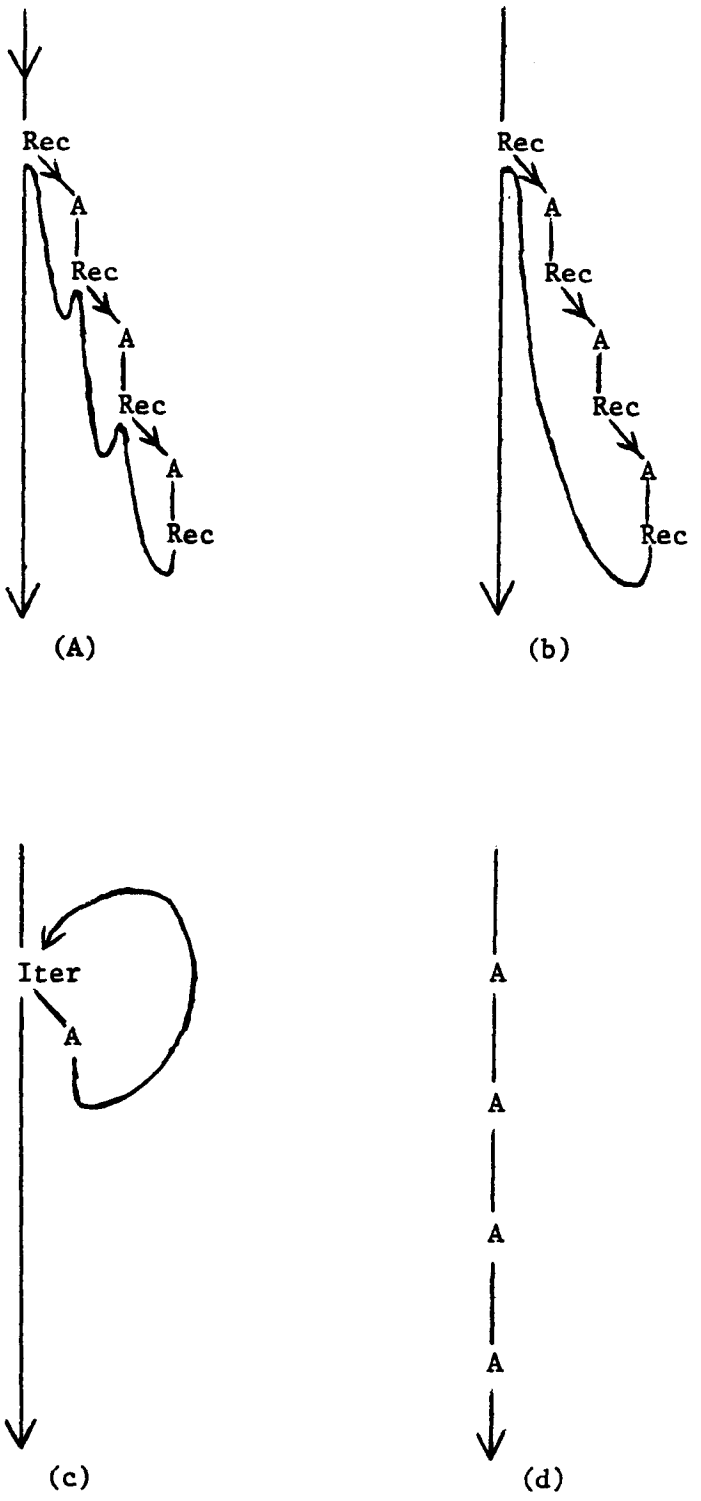


Figure 3-23. Optimisation of Tail-Recursion.

3.5.2. Storage

The above discussion related only to the generation of the instruction sequence at run-time. For Iteration to be considered as a special case of Recursion a second criterion must be met, namely that the data space of each invocation is identical ie no local variables. This condition is obviously fulfilled, as Generation only applies to the instruction sequence, because the whole of the data store is 'input' to each and every instruction and the whole of the data store is 'output' from each and every instruction. This has the effect of allowing communication between every pair of instructions making a program less nearly-decomposable. It would obviously be beneficial to restrict access to the data store thereby reducing enumerative reasoning because the system is nearer to being perfectly decomposable. A good way to do this is to allow each subsystem to have some private storage which it may, if it chooses, allow its constituent subsystems to access. Such a scheme would allow intra-subsystem local variables and inter-subsystem communication via parameters, the whole scheme forming a storage hierarchy linked to the functional hierarchy. It is interesting to compare this scheme with Algol 60's storage allocation in which, via inherited global variables, lower level subsystems can interfere with their brothers', parent's, grandparent's etc data spaces. Such a scheme clearly reduces near decomposability. Global variables are a bad feature because they go against the 'empty world' descriptive principle in that a low level subsystem has access to a global variable without having to specify anything. The high level system has no control over its own data space.

3.5.3. Other Hierarchies

Hierarchies may control the generation of executable instructions and the allocation and protection of data storage. Hierarchies may be used to control the allocation of resources in general, to implement protection schemes and

filestore dictionaries. Widespread use of hierarchies in software design brings many advantages, including the conceptual integrity that arises from utilising a common mechanism for dissimilar functions. In any useful program it is likely that several different hierarchies will coexist harmoniously. Each can be thought of as a Projection (see section 2.4) of one aspect of the program. Such hierarchies are more closely linked with the actual function of the program than are the instruction sequence and storage allocation hierarchies.

3.5.4. Binding Time

One aspect of Generation that has so far been overlooked is Binding Time. This is the point in the Generation process when either a selection is actually made or a formal parameter is replaced by an actual value. Obviously all binding affecting a given instruction must be completed before the execution of that instruction, but in general the later the binding the more general and powerful the program. If all binding is done, say, during compilation then the executable program will be the enumerated version ie large and inflexible, but fast because no generation or decisions are required. If all binding is left as late as possible then a small, flexible program results; the self-modifying interpreted program is an example in which much generation and selection must take place in addition to 'useful' work. Thus there are tradeoffs involved in delaying or accelerating the point at which binding takes place, and it is important that the binding time of any given decision is clearly defined.

3.5.5. Development

One aspect of a program usually resolved by early binding is which version is to be generated. Large software systems particularly tend to exist in many different versions and a hierarchical structure can help with system development because in a functional hierarchy a given subsystem can be simply

replaced by a different version if no change is made to the actual function performed. This allows the simple and reliable replacement of one version by another. If a change is to be made to a system the distance 'up and down' the hierarchy that the change penetrates indicates the seriousness of the change. Conversely hierarchical structuring tends to limit the effect of any one change as each subsystem has a limited effect on the total system.

A hierarchically structured system seems to be an advantageous structure if an active development programme is envisaged. This is true too in the natural world where Evolution favours hierarchies.[73]

3.5.6. Dynamics

Hierarchically structured executable programs share a second characteristic with more natural hierarchical systems in that their Dynamics are similar. In a perfectly decomposable system all activities would occur within the subsystems of a given level and no interactions would occur between them. In a nearly decomposable system, the more common type, most of the activity is within the subsystems but a small amount of inter-subsystem interaction occurs (side effects). The duration and frequency of activities tends to be proportional to Depth. Higher level subsystems act for longer periods but at lower frequencies (program runs for one hour, once per day) than lower level subsystems which act for shorter periods of time at higher frequencies (add instruction runs for one microsecond, 10^{10} times per day). This means that the short term behaviour of a subsystem is independent of the other subsystems (add unit acts independently) and that the long term behaviour of a system depends on the aggregate behaviour of its subsystems (filing system affected by all user jobs over long periods of time). These observation are supported by the evidence from performance measurement (see section 7.6.2).

It might be said that being able to relate the static Generative Description to the actual dynamic behaviour of an executing program is a major programming skill. Understanding run-time behaviour is the motivation behind the concept of Animation (see section 2.3.2 and 7.6.3). However the preceding sections should have convinced the reader that a logically correct program can be understood in its static form, for it is irrelevant how long each operation takes, so long as it is finite, and it is irrelevant how many times each loop is repeated, so long as it is repeated the correct number of times.

An understanding of the dynamic behaviour is only important at a secondary level, when a logically correct program is optimised to improve its efficiency. If a program is built as a hierarchy of Sets and Sequences, and Hierarchical Reduction, Integration and Induction are used to reduce its description, to provide an understanding and a proof, and to generate the executable program, then this conceptual integrity ensures that the Dynamic Behaviour will be strongly related to the Static Description and be consequently much easier to visualise, animate and understand.

3.6. Conclusion

This beautiful unification of the Static Description and the Dynamic Behaviour has been achieved by starting from the simplest possible program structure, the pure sequence of instructions, applying a set of Abstractions to reduce the sequence to a Generative Description, and then enhancing the RVNC so that the GVNC could use the Generative Description directly to reproduce the original sequence, so in the end the simplicity is regained. At the heart of the matter is the Generative Description (figure 3-24) which is the most concrete form of the program available to the programmer. This is the working form used for design, proof, generation, animation, manipulation and development. The technique for representing the Generative Description is a central

and vital programming tool. A new representational technique has been surreptitiously introduced during this chapter. Now it will be more fully developed as the second stage of Abstraction ie Representation is considered.

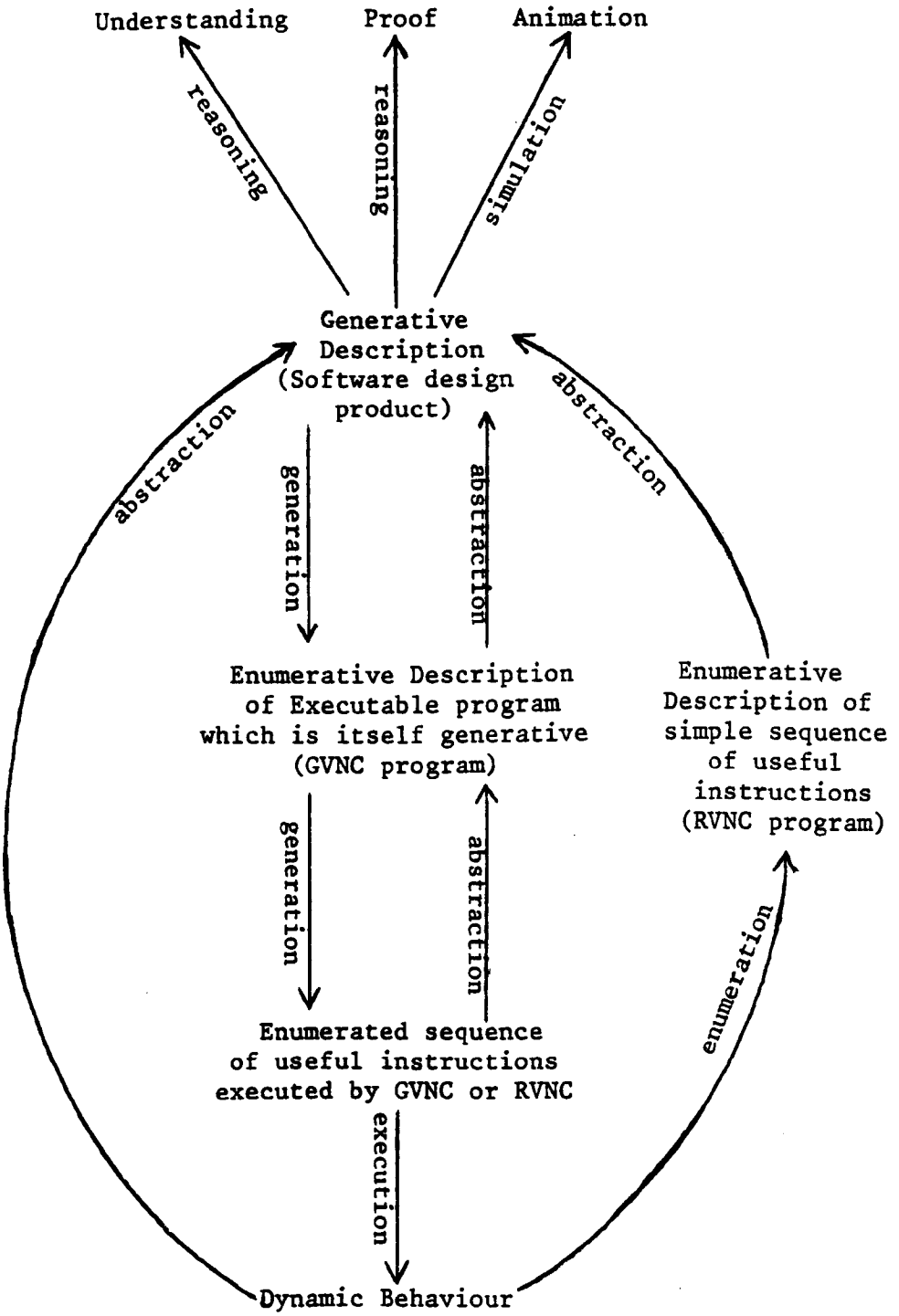


Figure 3-24. Enumeration and Generation.

CHAPTER 4. DIMENSIONAL DESIGN: REPRESENTATION

OUTLINE

The Software Crisis, Software Engineering and Software Life Cycle discussions lead to the identification of the small scale software engineering problem concerning the quality and tractability of the software design product. A set of programming abstractions were introduced as the basis for Dimensional Design, an attempt to solve the small scale problem.

This chapter, Chapter 4, informally introduces the Dimensional Design representational technique which uses an analogy with the three spatial dimensions to represent the Hierarchy, Set and Sequence abstractions. The Enumerative and Generative use of Hierarchical Reduction, Integration and Induction are explicitly represented by special symbols. Dimensional Design thus attempts to enhance *textual* descriptions of programs by these *perceptual* devices to make instances of the programming abstractions immediately visible.

The ability of Dimensional Design to portray the key features of many aspects of software design is brought out in a wide range of examples, including instruction sequences, data structures, parallelism, design versions, stepwise refinement and input/output files. Some interesting lapses of generality and conceptual integrity amongst conventional programming techniques are highlighted, in passing.

These examples lead up to the observation that a Dimensional Design is, in the general case, a tree of symbols whose edges perceptually encode the relationships between the nodal symbols. The chapter closes with the statement of several hypotheses about the advantages of this form of representation. The validity of these hypotheses is tested by the succeeding chapters which examine the manipulability, tractability, practicality, axiomatisation and originality of Dimensional Design.

4. DIMENSIONAL DESIGN: REPRESENTATION

A computer program is a complex structure with millions of components. Fortunately this does not necessarily mean that a program is difficult to understand. How hard or easy it is to understand a program depends critically on how it is described. A modern programmer works almost exclusively with the *description* of his program, for the real executable binary program is so large, complex and obscure that only a desperate or foolhardy programmer ever manipulates or studies it directly. The major activities of a programmer, and of a maintenance programmer in particular, are the understanding and manipulation of the description of the executable program. The way a program's description is represented critically affects the ease, reliability and accuracy with which these two activities may be performed. Dijkstra[19] illustrates this point with an analogy about numbers. "The ease of manipulation with numbers is greatly dependent on the nomenclature we have chosen for them. It is much harder to establish that

Twelve times a dozen = a gross
Eleven plus twelve = twenty-three
XLVII + IV = LI

than it is to establish that

12*12 = 144
11+12 = 23
47+ 4 = 51

because the latter three answers can be generated by a simple set of rules that any eight year old can master". Notice that the left hand sides of the latter three equations are Generative Descriptions and that each number is itself a Generative Description, a hierarchy based on powers of ten.

Before a program can be manipulated it must be understood. The previous chapter demonstrated that understanding is facilitated by recognising commonly occurring features in a program which may be abstracted. It would

therefore seem sensible to represent the description of a program so that the relationship between the program and the abstractions is clearly indicated. In the author's opinion, modern representations of programs do not adequately express this relationship (see Chapter 8), so the author has invented a new representational technique, Dimensional Design, which attempts to represent, simply and explicitly, the abstract concepts used in programming.

4.1. Informal Introduction to Dimensional Design

The first and fundamental premise on which Dimensional Design is based is that a program is best described as a hierarchy of abstractions, and that this hierarchy is best represented by an explicit picture showing the exact relationships between the hierarchy's subsystems.

The second premise is that, at any level of a program's descriptive hierarchy, every non-elementary subsystem can be described as either a Set or a Sequence of lower level subsystems.

The third premise is that every Set and Sequence can be described Enumeratively or Generatively and that Enumeration and Generation are abstractions which can be explicitly represented.

The following discussion will show how Dimensional Design represents each of the abstractions mentioned in Chapter 3 by presenting a wide range of examples.

4.2. Enumeration

4.2.1. Subsystem

For the purposes of this informal introduction only, a subsystem in a Dimensional Design is a linear string of characters. Characters such as 'newline' may appear in the character strings so a subsystem may be any piece of

text. Common forms of elementary subsystem are the 'useful work' instructions such as

```
A := B + C;

WRITE (1,100) A,B,C
```

Elementary subsystems will henceforth be called Elements or Atoms. The Null subsystem is represented by • .

4.2.2. Sequence

An ordered sequence of subsystems is represented as a vertical list of subsystems with the first subsystem at the top. Each subsystem is connected to its successor by a vertical, downward directed arrow. An Enumerated Sequence is terminated by the earth symbol ($\overline{\overline{-}}$) which is synonymous with enumeration. A sequence of subsystems enclosed by the symbols (• $\overline{\overline{-}}$) is itself a subsystem. [Note: formally this construction depends on relation precedence - see Chapter 8: Axiomatisation]. Figures 4-1 and 4-2 are examples of enumerated sequences.

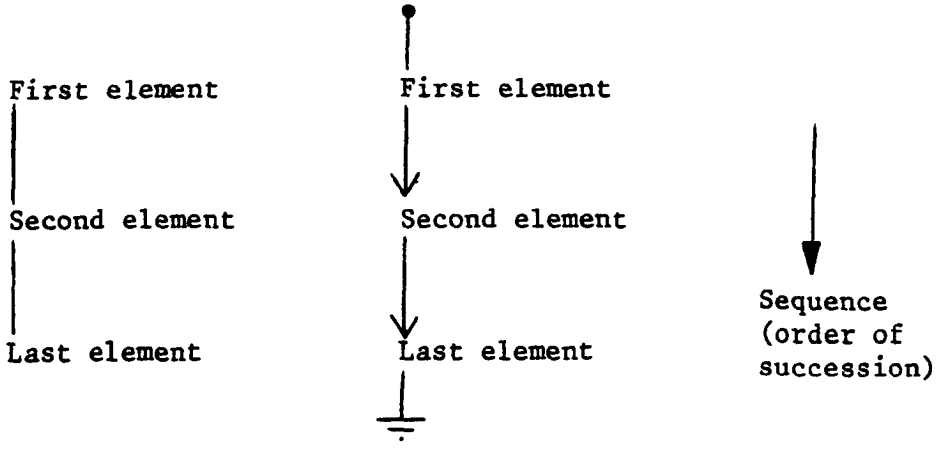


Figure 4-1. Enumerated Sequence.

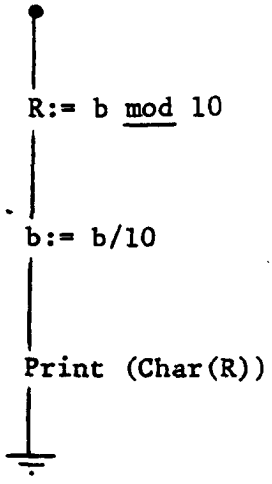


Figure 4-2. Enumerated Sequence of Instructions.

4.2.3. Set

An ordered set of subsystems is represented as a horizontal list of subsystems with the first subsystem on the left. Each subsystem is connected to its successor by a horizontal, rightward directed arrow. An enumerated set is terminated by an enumeration symbol ($\vert\vert\cdot$). A set of subsystems enclosed by the symbols • and $\vert\vert\cdot$ is itself a subsystem. Figures 4-3 and 4-4 are examples of enumerated sets.

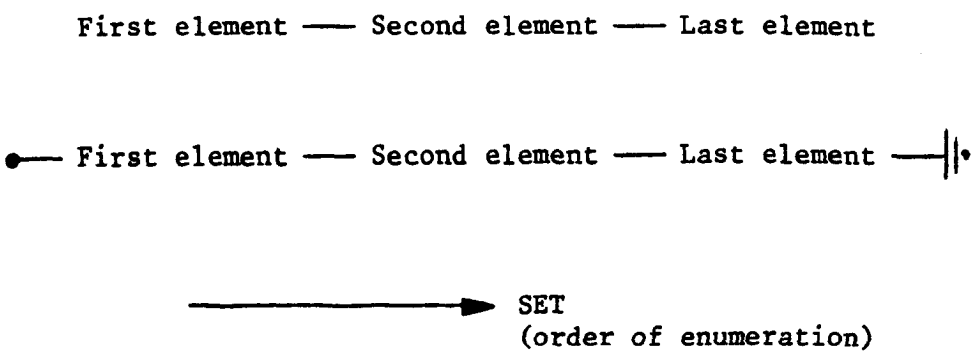


Figure 4-3. Enumerated Set.

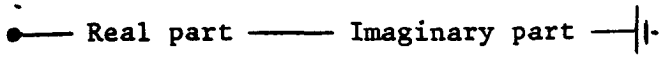


Figure 4-4. Complex Number represented as Enumerated Set.

4.2.4. Hierarchy

If subsystem B is hierarchically subservient to subsystem A then this relationship is represented by joining A to B by an arrow directed at B and orthogonal to both the Set and Sequence directions. Such a representation clearly requires a three-dimensional medium. Fortunately three dimensions can be projected onto two so the hierarchical relationship is expressed by a line at 45° to both the Set and Sequence lines (figure 4-5). If B is the lowest level of the hierarchy then the symbol /. is used to indicate that B is an elementary subsystem ie the hierarchy has been enumerated down to its lowest level. The /. symbol is often omitted when a subsystem is obviously elementary.

In any Dimensional Design the relation R governing the hierarchy must be specified. R must be constant, of course, throughout the hierarchy. The examples in this section use R='What-How', the double relation, to create functional hierarchies.

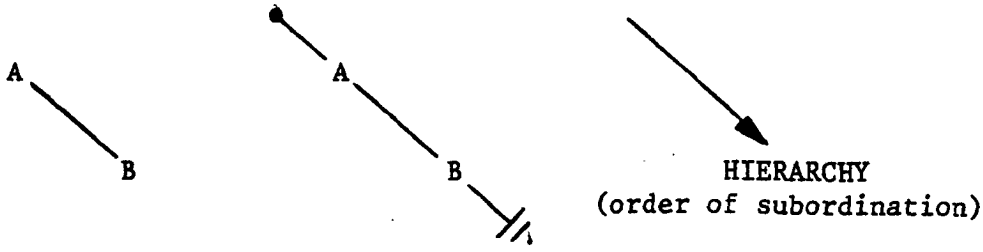


Figure 4-5. Hierarchy: B is subordinate to A.

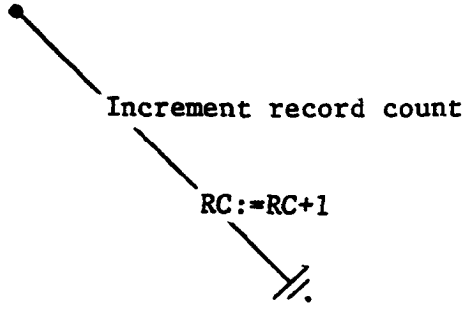


Figure 4-6. 'What-How' Double Hierarchy.

4.2.5. The Three Dimensions

The name Dimensional Design is derived from the technique of using the three orthogonal spatial dimensions to represent each of the three independent abstractions, Set, Sequence and Hierarchy. A Dimensional Design is a directed graph. Each node in the graph is a subsystem. Nodes are connected by edges representing the three abstraction relationships; because the edges representing these abstractions are orthogonally aligned, the orientation of each edge relative to the specified axes (figure 4-7) uniquely identifies which relationship exists between any two subsystems. All edges are directed edges. For each edge its direction must be parallel to one of the three axes and in the direction specified on that axis. No two edges may be directed at the same node, thus every Dimensional Design is a directed tree, a T(h)ree Dimensional Design. As the direction of any edge is implied by its orientation relative to the axes the arrow heads are usually omitted.

The three concepts of Hierarchy, Set and Sequence may thus be used in combination to express complex ideas such as a multi-tasking program having a set of two tasks, running in parallel, where each task is a sequence of instructions (figure 4-8). In figure 4-8a the arrow heads have been included but they will usually be omitted in all following examples, eg figure 4-8b. Every directed edge must join two subsystems in a completely Enumerated Description. An edge which does not join two subsystems implies that the program's description is not completely enumerated; this is the key to representing Generation.

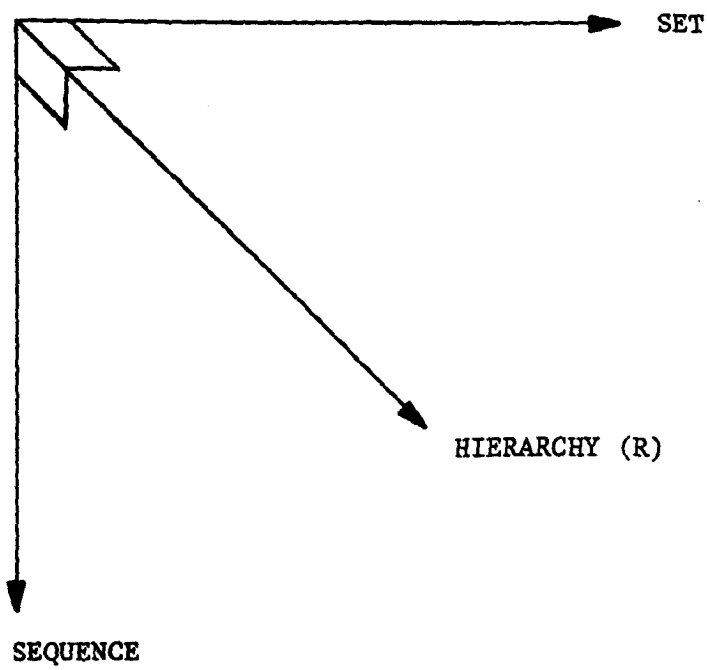


Figure 4-7. The Three Dimensions.

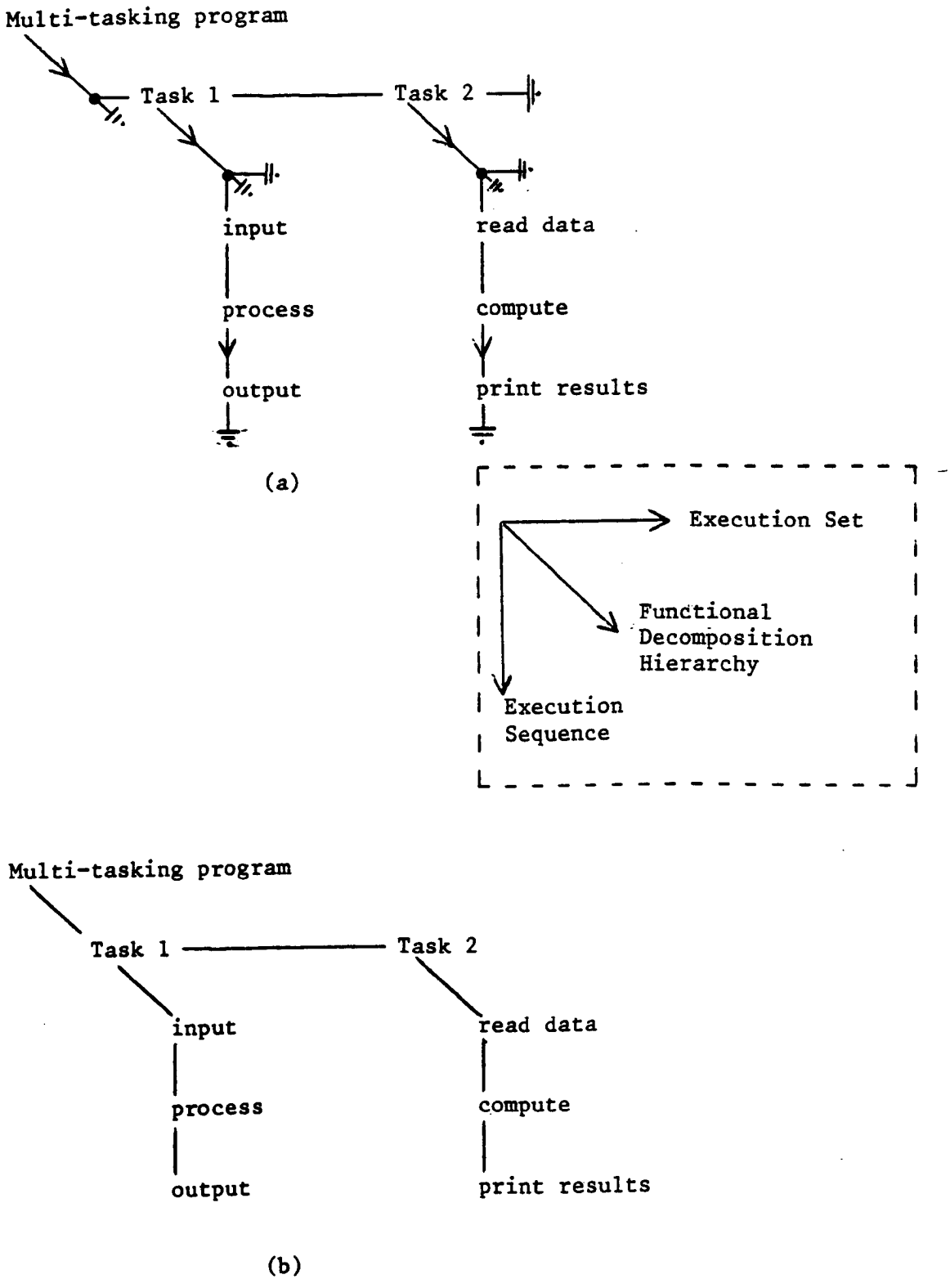


Figure 4-8. 3-D Combination.

4.3. Generation

4.3.1. Hierarchical Reduction

The size of a description may be reduced by eliminating all but one of the detailed specifications of a common subsystem. This process is called Hierarchical Reduction, often shortened to H-Reduction. (See section 3.3.1.) In figure 4-9 the second occurrence of subsystem B has had its lower levels eliminated. This is shown by the directed edge leaving B not being connected to another subsystem, implying that the lower level subsystem must be generated to produce the Enumerated Description in figure 4-10. [Note: sometimes it is convenient to represent a missing subsystem explicitly so as to differentiate between a finite subsystem (□) and an infinite subsystem (▣).

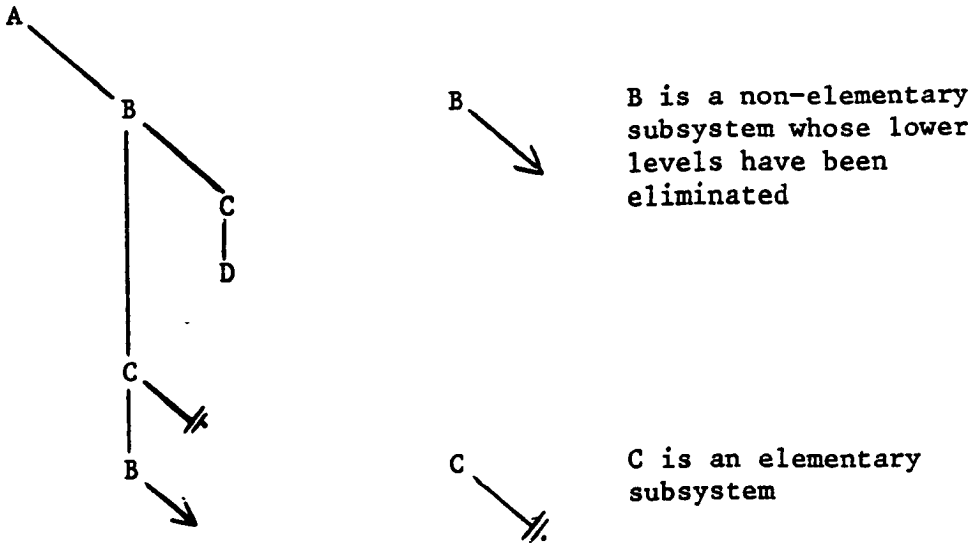


Figure 4-9. Generative Hierarchy.

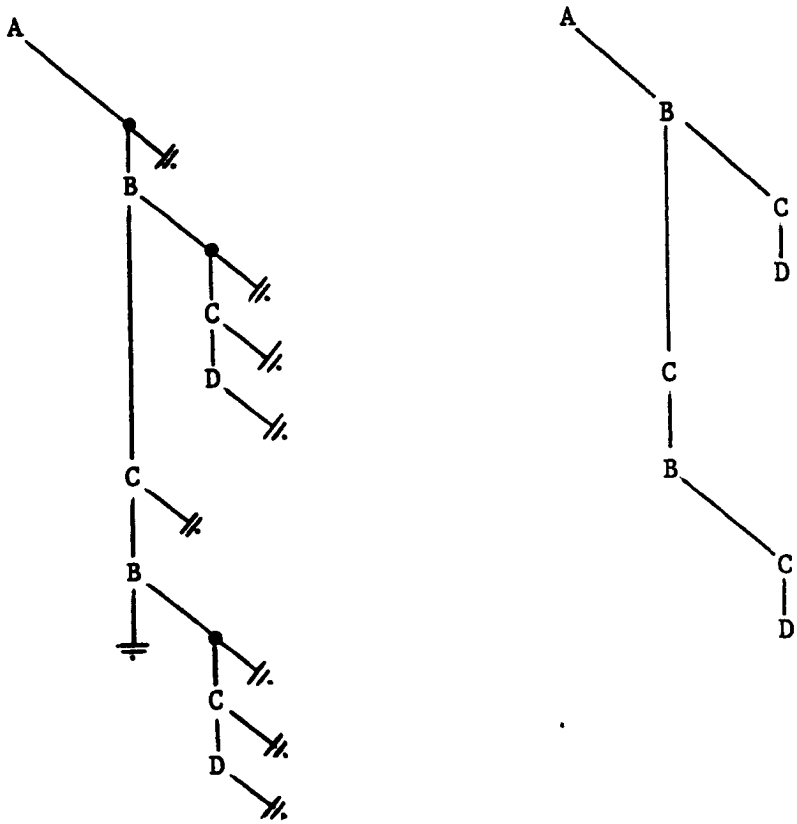


Figure 4-10. Two Enumerated Versions of Figure 4-9.

4.3.2. Integration

Integration depends on a selection mechanism to choose, for example, one subsequence from a set of possible subsequences according to some criterion. Dimensional Design has a binary selection device which either makes or breaks a connection according to the value of a Boolean variable, see figure 4-11. If the Boolean is *false* then the possible connection is broken and a part of the Dimensional Design tree is 'pruned' ie is removed from the tree, thereby eliminating one possibility.

The selection device may be used in any dimension and its Boolean may be computed by a subsystem (see section 4.8). The exact point in the generation process when the selection is made is a function of Binding Time (see section 3.5.4). The selection device must be used in conjunction with other concepts to implement Integration. For example, in figure 4-12 a set of programs is integrated and then one of them is regenerated assuming Trans=amend.

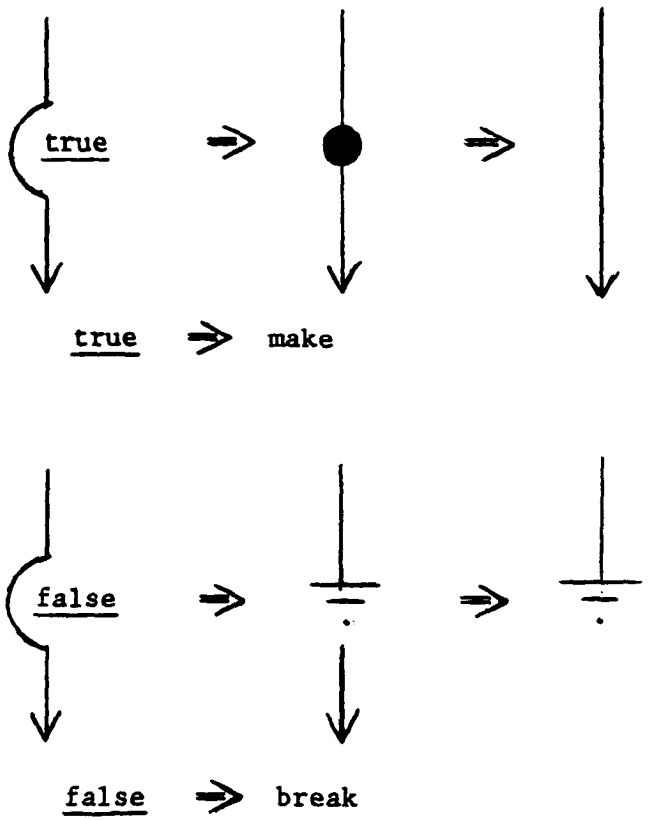
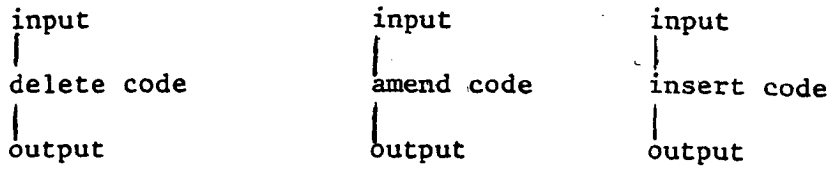
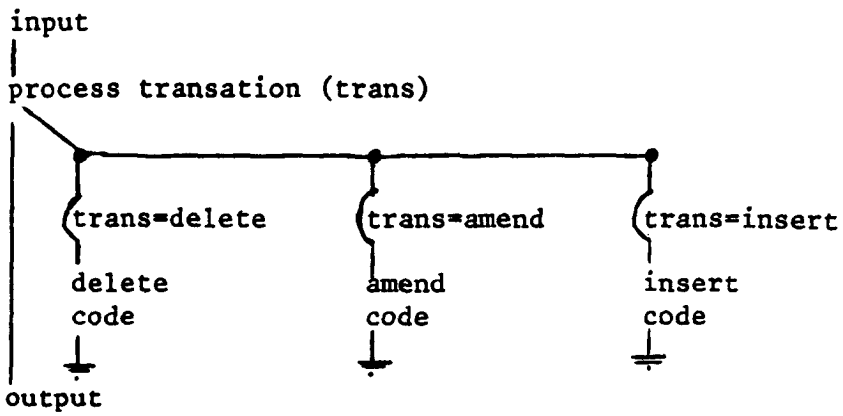


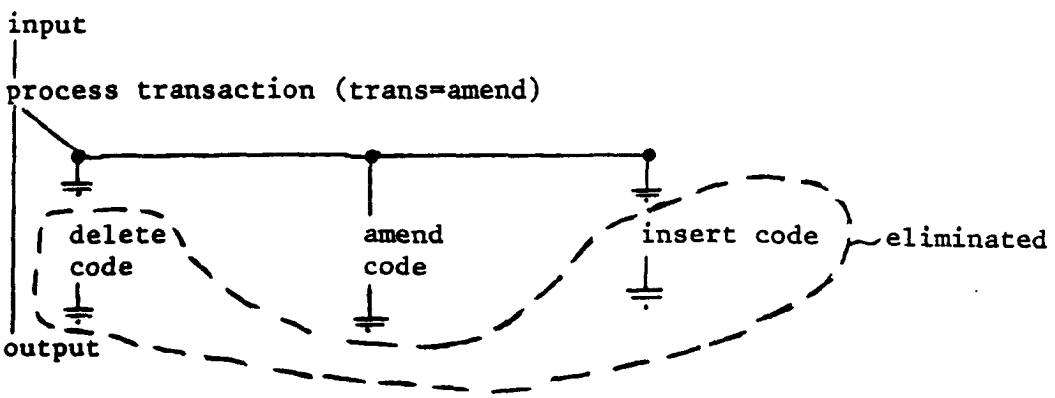
Figure 4-11. Sequential Conditional Selection Mechanism.



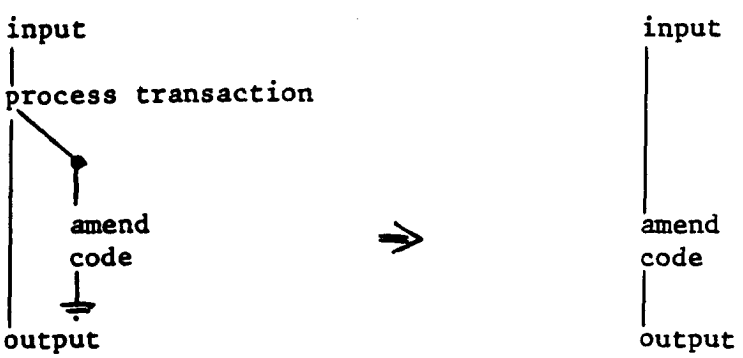
(a) Three separate but related programs.



(b) Integrated description - set of possible subsequences.



(c) After Selection.



(d) Completion of Generation.

Figure 4-12. Integration & Selection Animated.

4.3.3. Induction

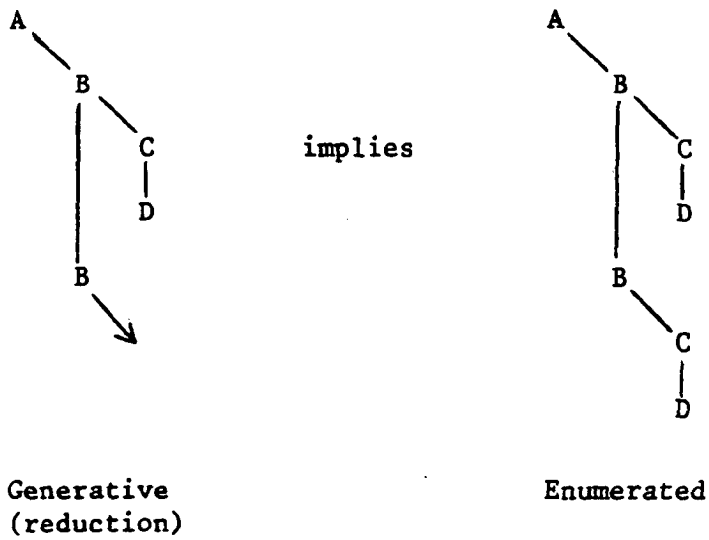
4.3.3.1. Recursion

Induction is the technique used to compress the description of common subsystems which occur in a regular order. The most powerful inductive technique is Recursion which involves the self-referential definition of a subsystem and a generation mechanism similar to H-Reduction. (See sections 3.3.1 and 4.3.1.)

H-Reduction and Recursion both imply a constant relationship between a system and its constituent subsystems so that the constituent subsystems can always be deduced from the unexpanded occurrence of the higher level subsystem (figure 4-13). In figure 4-13 the hierarchy relation, R, is R='is *always* expanded into', so B is always expanded into [C; D]. A sequential relation, Q, containing such a *constant* factor might be Q='is *always* followed by'. This would allow H-Reduction to occur in the sequential dimension too. For example, figure 4-14 shows how an infinite sequence of [A; B]s might be expressed. Such strong sequential and set relations usually preclude the representation of useful sequence and sets for programming purposes; the axes in figure 4-15 show the normal programming relations. This is why, so far, all sets and sequences have had to be fully enumerated; that is because the Set and Sequence programming relations are weak ie Enumerative whereas the Hierarchy programming relation is strong ie Generative. Conventionally, H-Reduction and Recursion are usually exploited only in the Hierarchy dimension, so sequences and sets containing redundancy must exploit the Hierarchy dimension if their descriptions are to be compressed. This is a pleasing result because it explains one reason why Hierarchy is such an important abstraction.

Although H-Reduction and Recursion may both be represented by an edge which leads from one subsystem but not to another, it is reasonable to distinguish between them as Recursion implies multiple generation (ie induction) and H-Reduction does not ie it implies enumeration. A H-Reduction edge can be represented as an edge with an open arrow head and Recursion by an edge having a solid arrow head. Although usually omitted the arrow head on an edge connecting a subsystem whose composition must be used in either H-Reductive or Recursive generation can be used to differentiate between the two classes of expansion (figure 4-16).

Similarly it is unnecessary but often useful to emphasise that a subsystem has been removed by H-Reduction by explicitly replacing it by a symbol indicating that the missing subsystem is finite, the □ symbol, or infinite, the ▣ symbol. (Arrow heads and 'dangling edges' or 'missing subsystem' symbols are stylistic alternatives. The former is more indicative of removal by H-Reduction, the latter enables a Dimensional Design to remain a tree at all times which is useful when software tools are involved.)



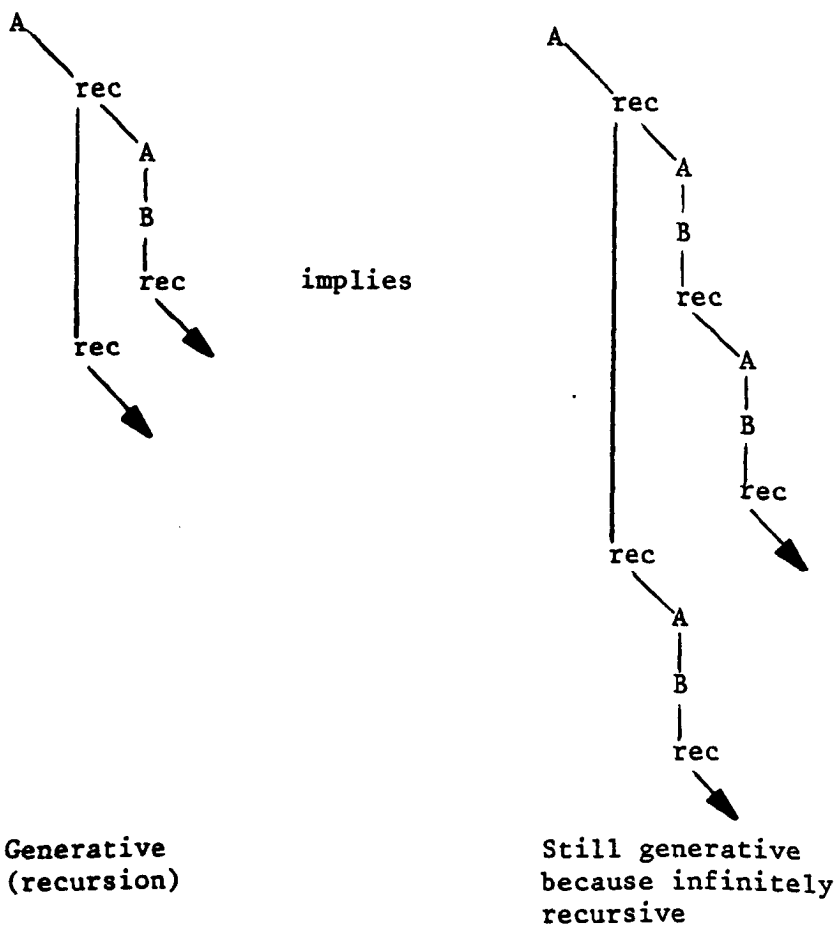
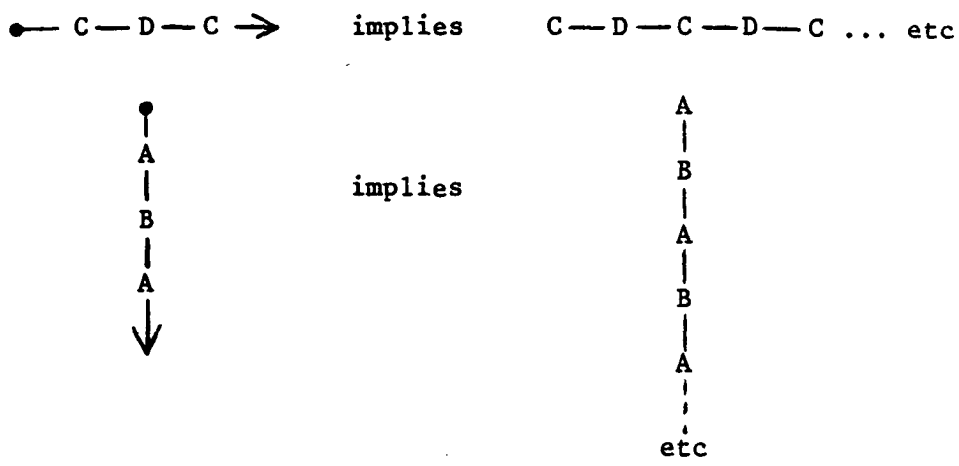
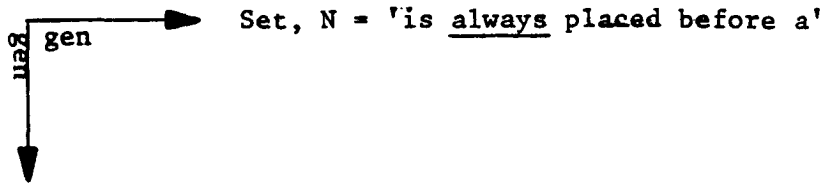


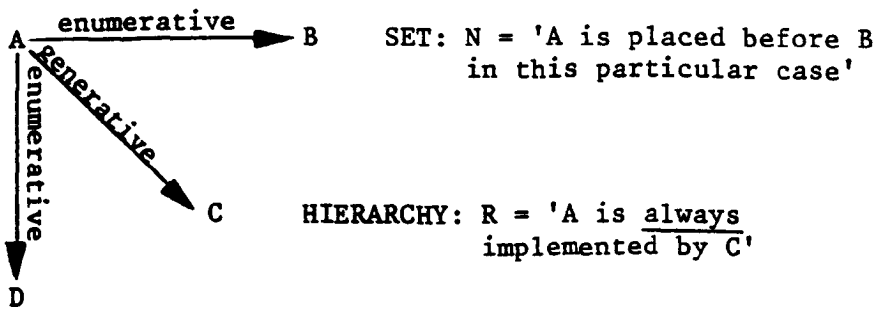
Figure 4-13. Reduction & Recursion.





Sequence, Q = 'is always followed by a'

Figure 4-14. Sequential & Set Reduction.



SEQUENCE: Q = 'A is followed by D
in this particular case'

Figure 4-15. Programming Axes.

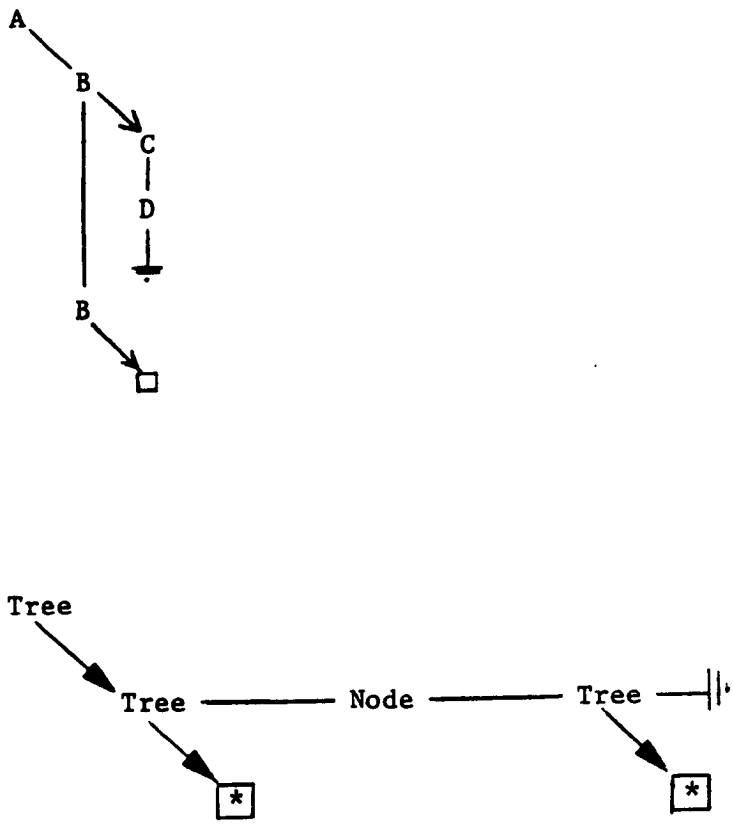


Figure 4-16. Reduction & Recursion Edges.

4.3.3.2. Iteration

Figure 4-17b shows the recursive description of an infinite number of sequential copies of the subsequence [A; B]. The simple formulation of figure 4-17a cannot be used because the Sequence relation is not strong enough ie not Generative (see figures 4-14 and 4-15). Tail recursions may be expressed as Iterations. This can be thought of, at the descriptive level, as the notational shorthand used to simplify figure 4-17c into 4-17d. At the generative level figure 4-17d can be thought of as a repeated subsequence enclosed by the Iterative delimiters • * . (The enumerated sequence delimiters • $\overline{-}$ were introduced to heighten the contrast between enumeration and inductive generation.) At the implementation level figure 4-17d can be thought of as representing the optimised form of implementing induction, stressing the choice of iteration over tail recursion.

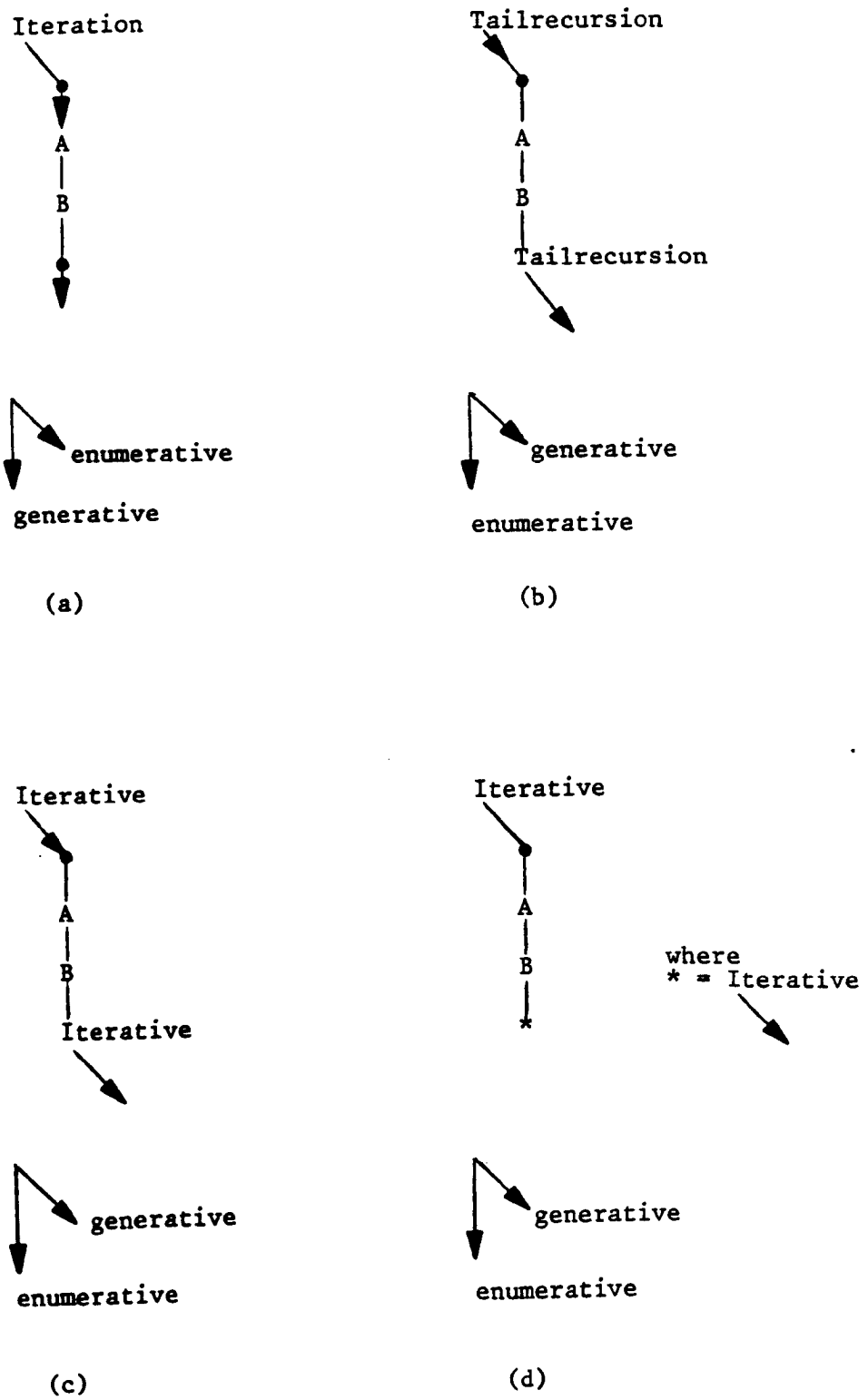


Figure 4-17. Iteration & Recursion (infinite).

4.3.3.3. Finite Induction

So far only infinite induction has been represented. Induction may be controlled by using the selection mechanism to generate, say, a sequence of the required length. Figure 4-17 and 4-18 contrast infinite and finite induction. Figure 4-19 shows an informal use of the selection mechanism amidst a comparison of the representations of the abstractions discussed in this chapter. Further examples will now be presented to show how Dimensional Design clarifies some more complex programming concepts.

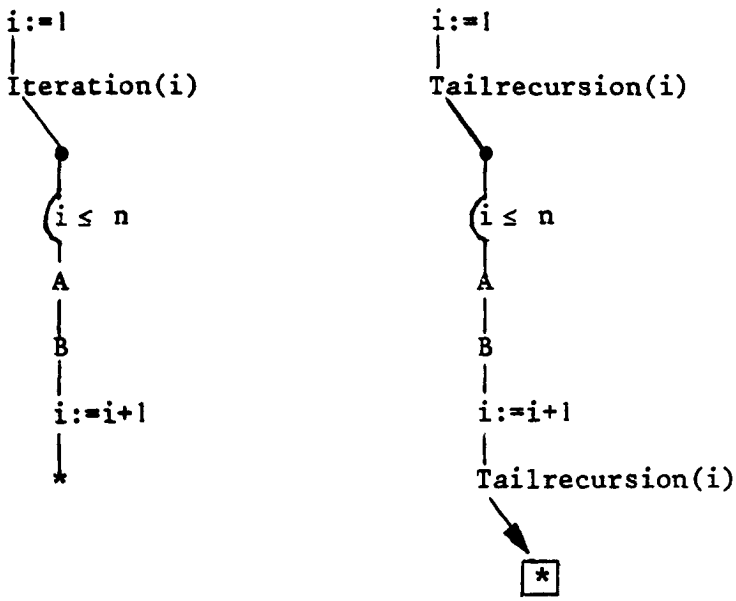


Figure 4-18. Iteration & Recursion (finite).

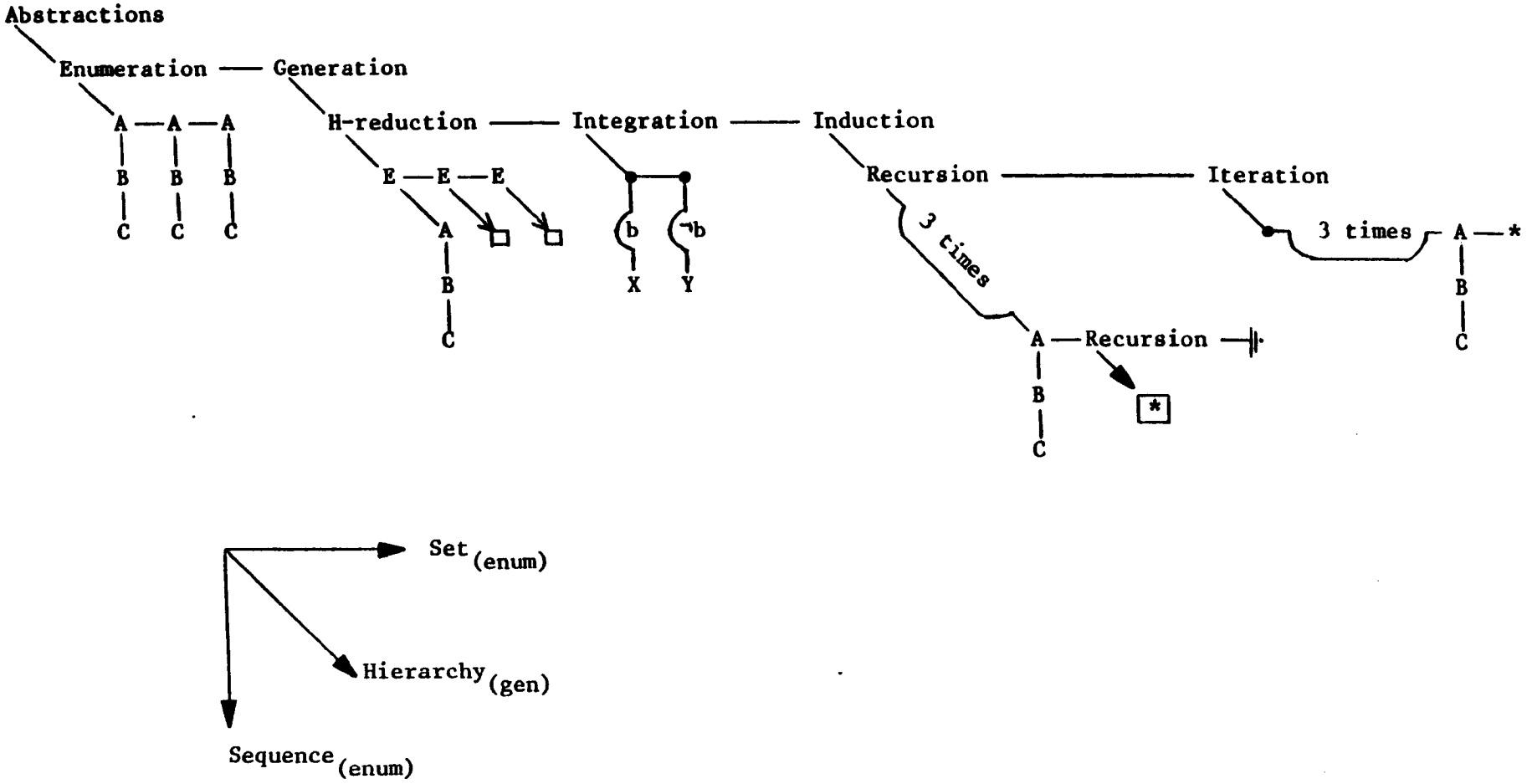


Figure 4-19. Set, Sequence, Hierarchy, Itegration, Induction, Enumeration, Generation.

4.4. Further Examples of Instruction Constructs

4.4.1. Integration

4.4.1.1. Enumerated Set

Integration is the combination of the descriptions of several, related programs. Dimensional Design represents Integration as a set of alternatives. As only one of the programs is actually executed, Generation involves the resolution of various selection criteria. There are different ways to resolve selection criteria; sequential and parallel resolution are the primary operational methods and their representations are dependent on the semantics of their generative mechanisms.

Figure 4-20 shows a set of alternatives. B_i and S_i are the i^{th} Boolean expression and associated sequence of instructions respectively. Suppose all the selection criteria are resolved instantaneously (ie before the execution of any non-selection statement, perhaps in parallel). Assume B_2 and B_4 are true, all other B_i are false. Then the result of resolving the integration shown in figures 4-21 and 4-22, is that S_2 and S_4 will be executed (in parallel). This method of resolving integration requires the horizontal axis of the Dimensional Design to be labelled 'Set, to be executed in parallel'. The more familiar sequential testing of alternatives to produce only one executable sequence can be represented, using parallel execution as in figure 4-23. As this is the most prevalent Integration mechanism on conventional von Neumann machines, its representation can usefully be simplified by relabelling the horizontal axis as 'Set of alternatives, search, left to right for first non-null alternative'. Thus figure 4-24 animates this other resolution mechanism. Contrast figures 4-23 and 4-24.

Use of a set of alternatives representation gives a far clearer picture of

the available options at any given point in a program. Figure 4-25a gives the more normal, operational view of sequential resolution. It is much more difficult to understand than figure 4-25b.

The sequential method of handling sets is based on a treewalking method of 'executing' a Dimensional Design, in which the subtrees are visited in the order

1. Refinement
2. Sequence
3. Set

(see precedence axioms in Chapter 6). This treewalking method gives the normal sequential von Neumann machine semantics from a Dimensional Design and is discussed further in Chapter 5: Manipulation.

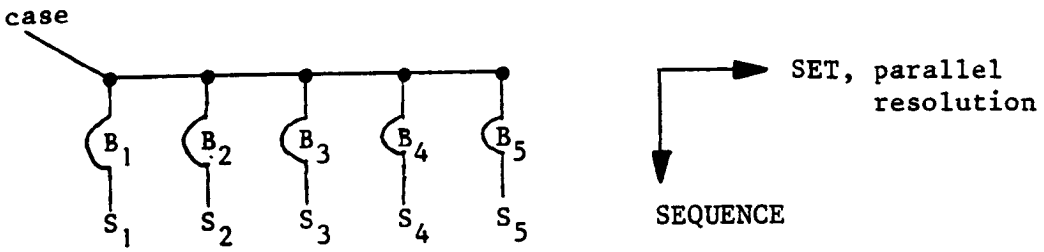


Figure 4-20. Parallel Case Statement.

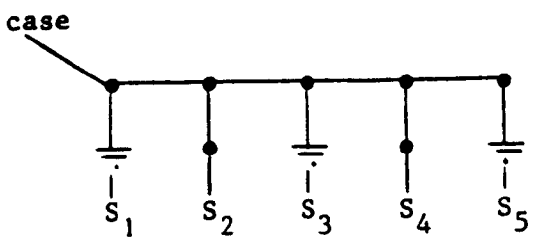


Figure 4-21. B_2 and B_4 true.

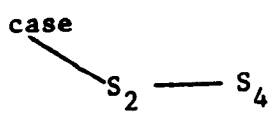
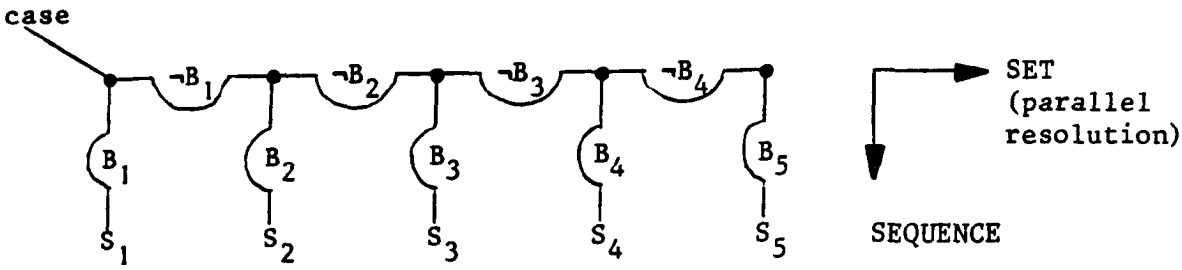
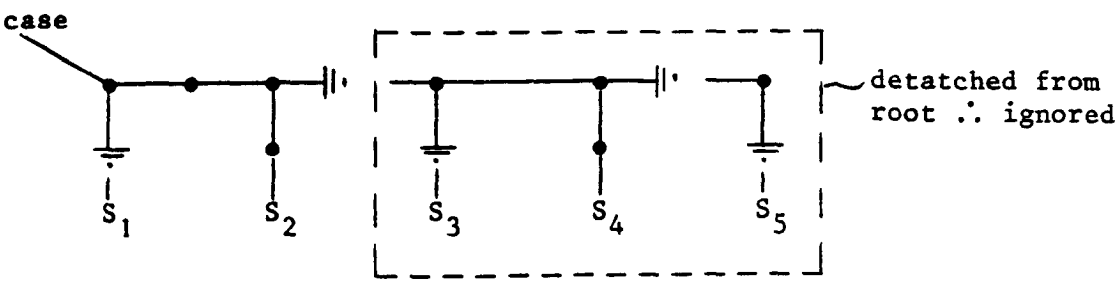


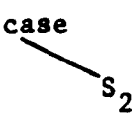
Figure 4-22. S_2 and S_4 executed.



(a) Find First Alternative

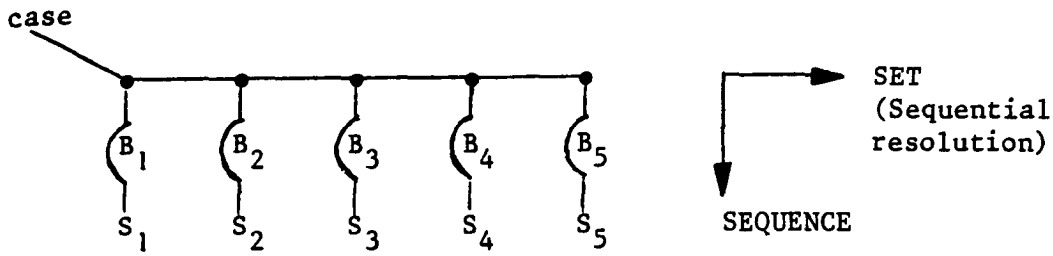


(b) B_2 and B_4 true

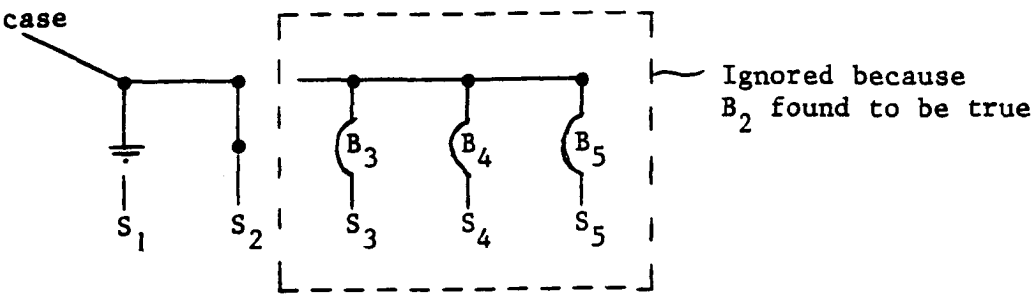


(c) Only S_2 executed

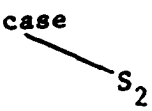
Figure 4-23. Sequential Testing by Parallel Resolution.



(a) Find First Alternative

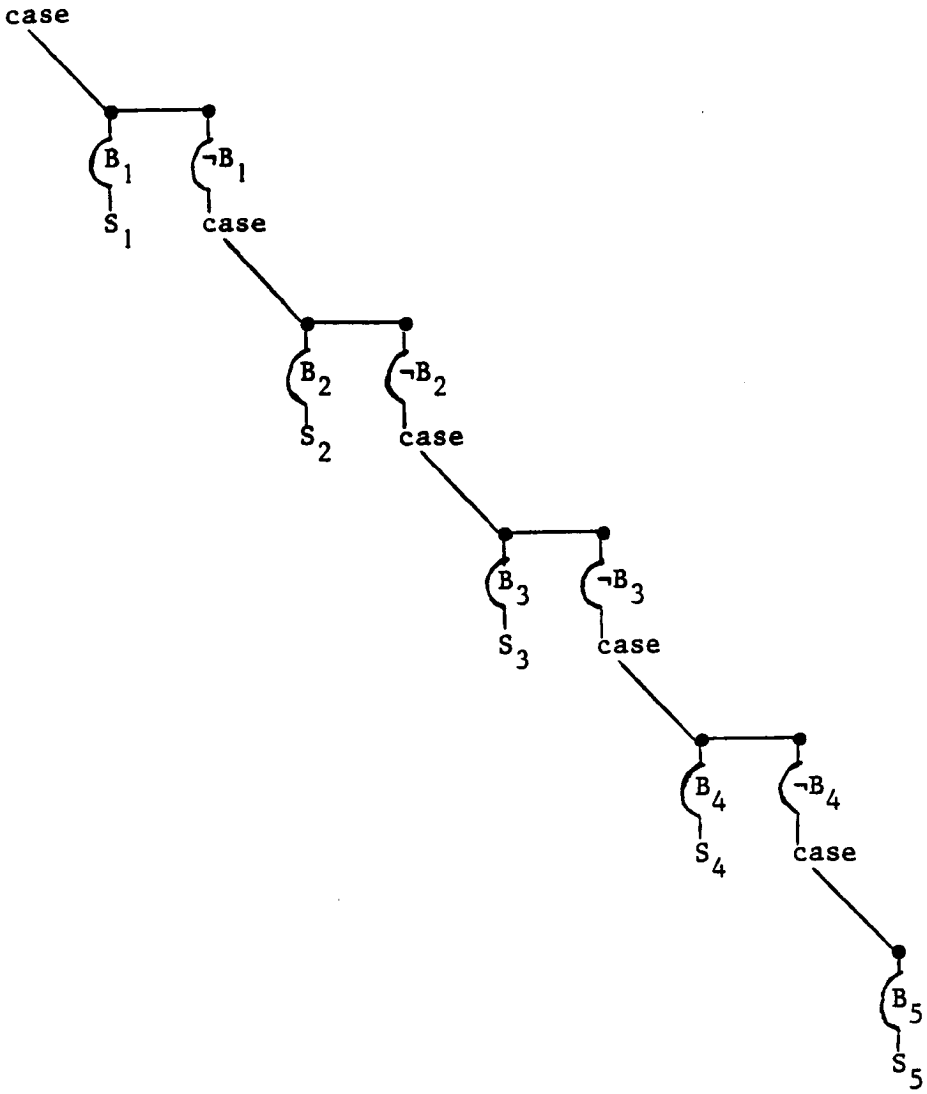


(b) B_2 and B_4 true

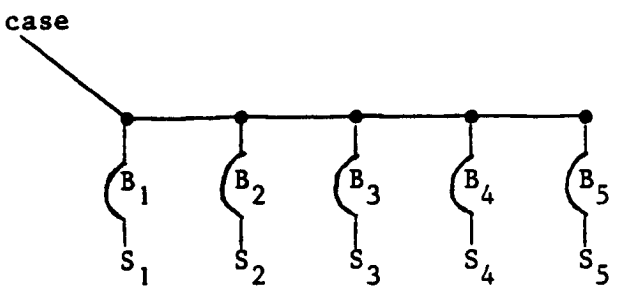


(c) Only S_2 executed

Figure 4-24. Sequential Testing by Sequential Resolution.



(a) Operational

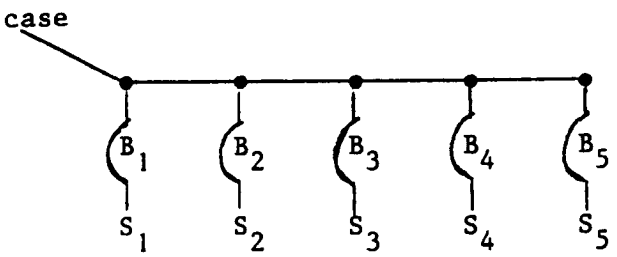


(b) Logical

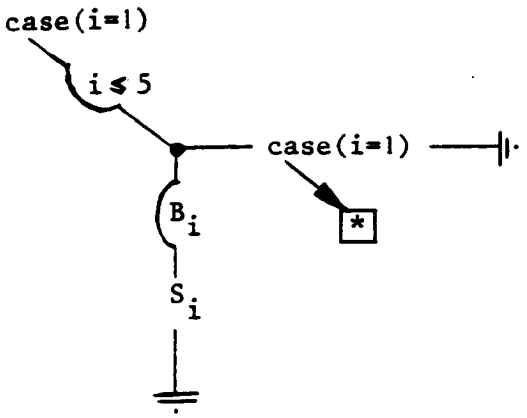
Figure 4-25. Integration: Operational v Logical View.

4.4.1.2. Generated Set

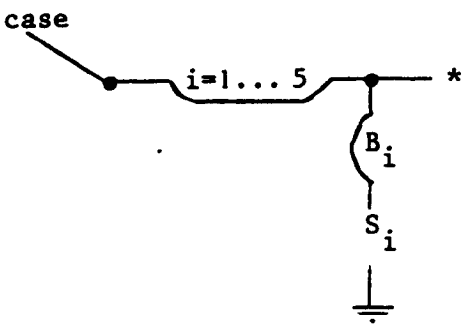
Figure 4-26a is an example of Integration showing an *enumerated* set of sequences. Such a set may be expressed inductively (figures 4-26 and 4-27). Old programming languages such as Fortran cannot represent the integration of an enumerated set of sequences, neither can sequences of Fortran instructions be generated by recursion; the most powerful programming abstraction is not available. Modern programming languages such as Pascal can represent Integration as an enumerated set of sequences and they allow both the iterative and recursive description of those sequences. It is reasonable, therefore, to ask why do modern programming languages not allow the inductive description of Integration? Such generality, such symmetry between the Set and Sequence dimensions, such conceptual integrity is highly desirable as the exploitation of Set Induction allows programs to be made smaller, simpler and easier to prove. For example, figure 4-28 is the inductive version of figure 4-29 which is taken from the routine CHAR (see section 10.2.2), a function to convert the binary version of a decimal digit to its external character code. Look at figure 4-29. Can you spot the error it contains? This error highlights the difficulty of enumerative reasoning and the advantage of the inductive description.



(a) Enumerated Set

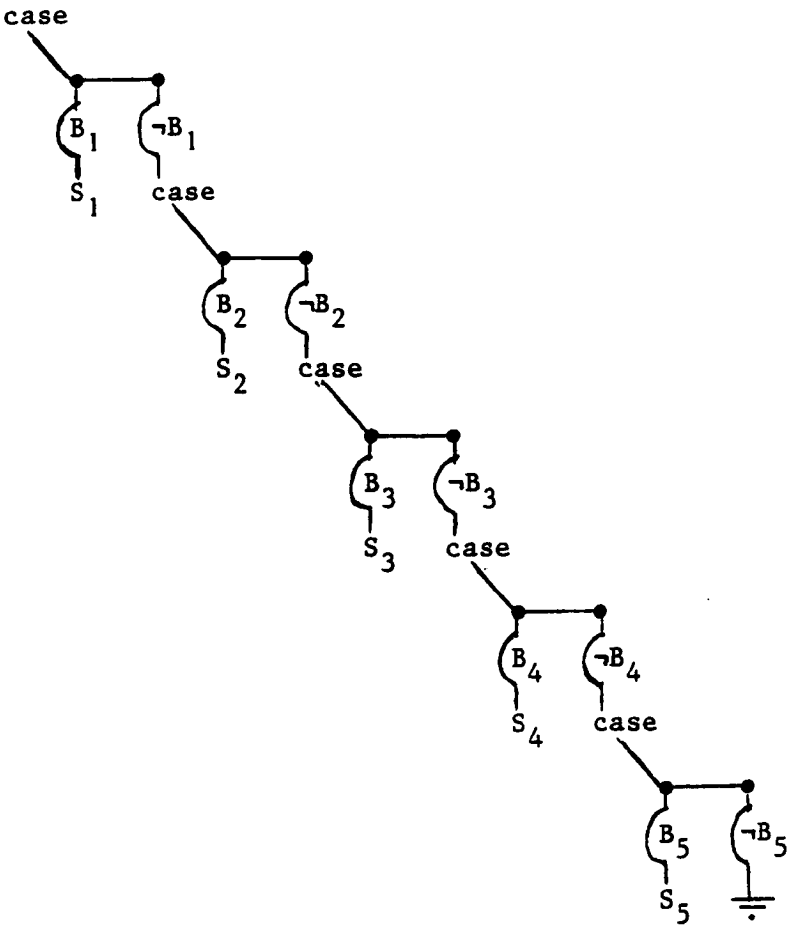


(b) Recursively Defined Set

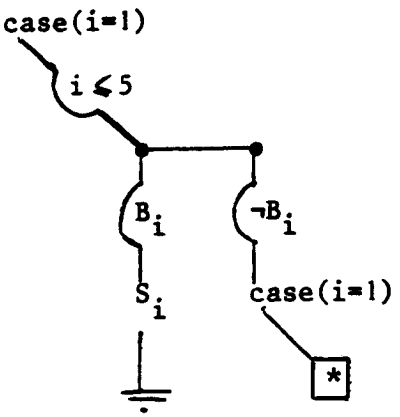


(c) Iteratively Defined Set

Figure 4-26. Integration : Inductive Set.



(a) Enumerated Set



(b) Recursively Defined Set

Figure 4-27. Integration: Inductive Set.

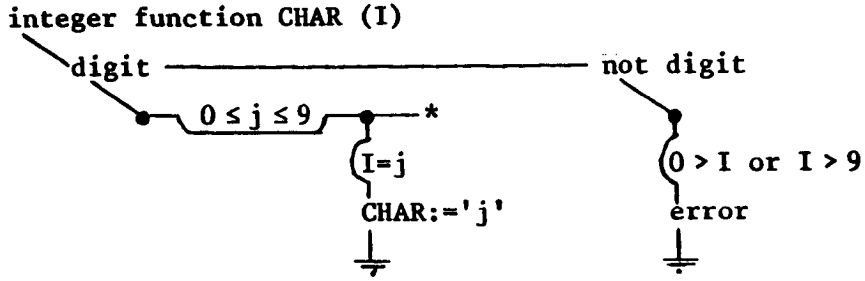


Figure 4-28. Inductive Integration.

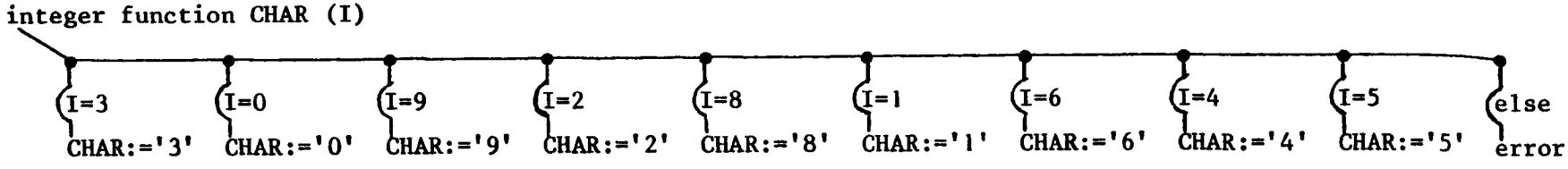


Figure 4-29. Enumerated Integration.

4.4.2. Parallelism

A special case of Integration is the non-selective combination of several subsystems which may be executed in parallel. Some modern programming languages can express such parallelism at either the statement or the process level. Hoare's Communicating Sequential Processes[35] can express both. Figure 4-30 shows a set of Communicating Sequential Processes whose parallel execution will form a linear pipeline of filters, the Sieve of Eratosthenes. As in the previous examples, induction greatly reduces the description. Induction is used to describe 100 (SIEVE(1)..SIEVE(100)) out of the 102 (SIEVE(0) to SIEVE(101)) processes which can be executed in parallel. These processes communicate values between themselves. The notation A!B means the value of B is sent to process A. The notation A?B means a value is to be received from process A and is to be referred to by the local name B. Communication is unbuffered and synchronised ie A:B!C must be matched by B:A?D.

Figure 4-31 shows a set of Communicating Sequential Processed simulating the parallel execution of mutually independent recursive invocations of a factorial function. This version of Gurd's factorial algorithm[86] shows both process level and statement level parallelism. The processes executed in parallel are Fac(0) to Fac($2^{max}-1$). The statements executed in parallel are the refinement of 'Split into 2 subcomputations', shown as a non-selective integration. Figure 4-32 is a more conventional, truly recursive version of the doubly recursive splitting algorithm, showing the inductive specification of a binary tree of parallel subcomputations which is dynamically generated and destroyed. Dimensional Design can explicitly represent this hierarchically structured form of parallelism, easing the identification of the ancestral relationships between processes, statements and invocations.

PAGE

NUMBERING

AS ORIGINAL

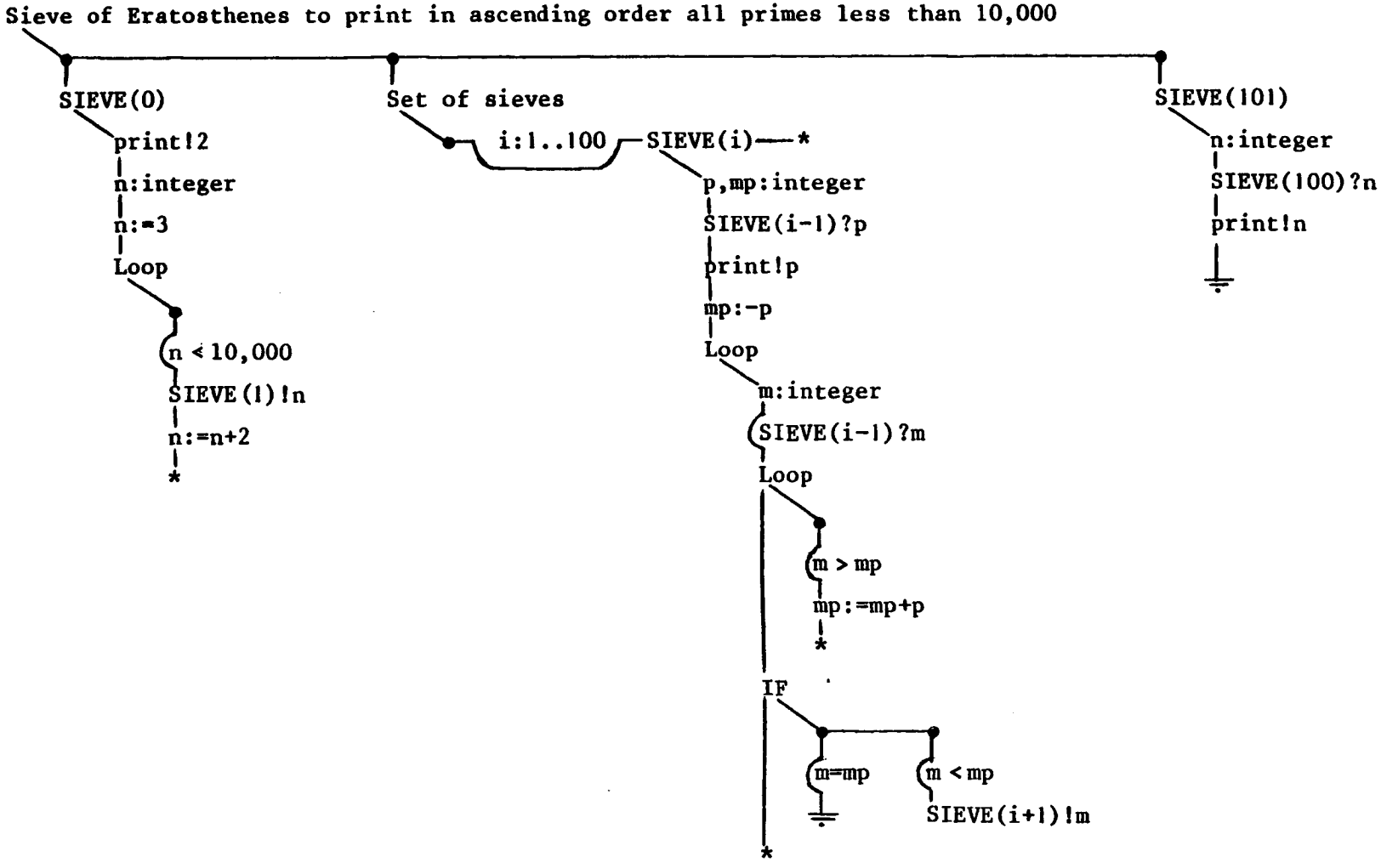


Figure 4-30. Set of Parallel Processes.

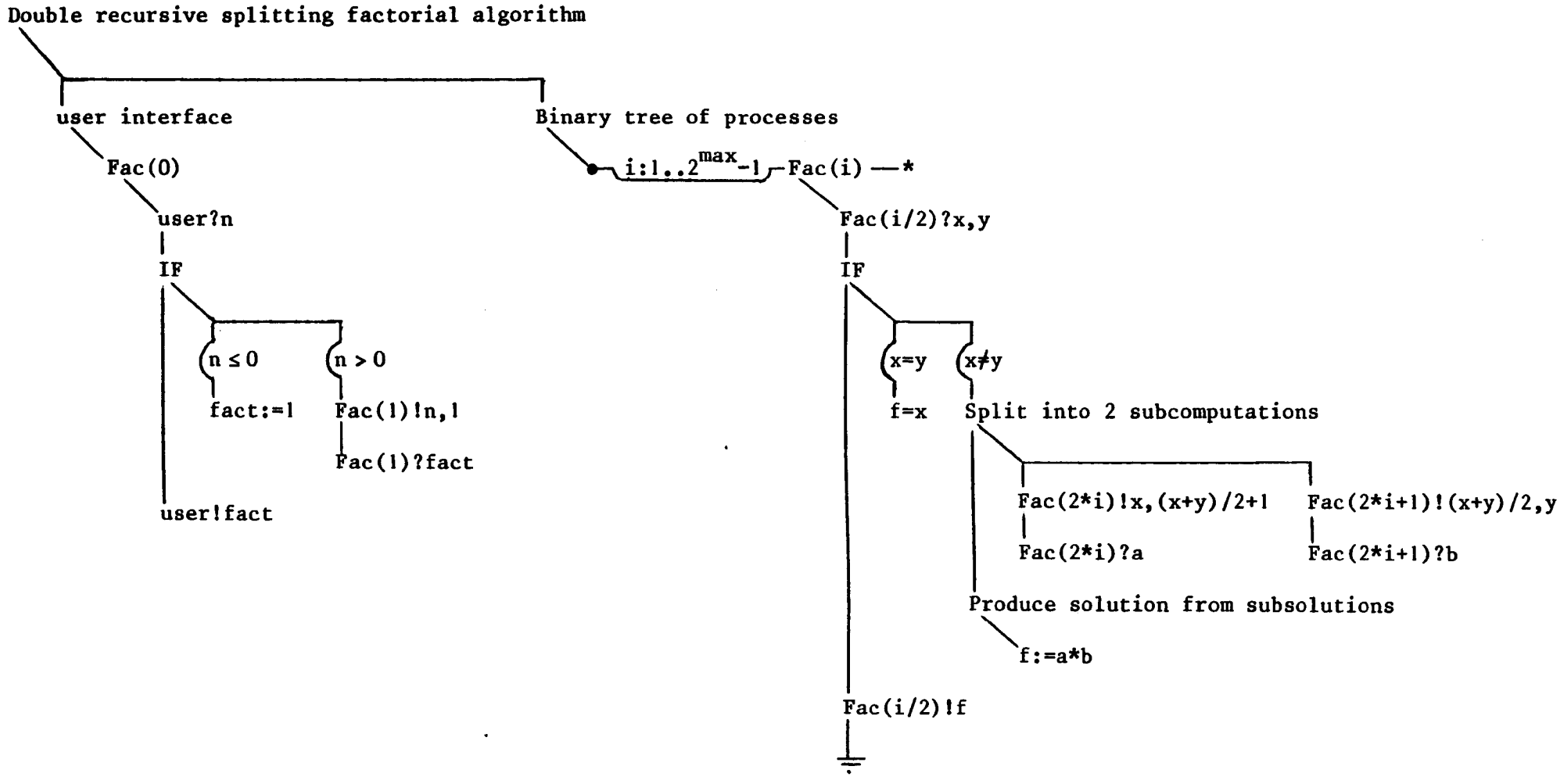


Figure 4-31. Process & Statement Level Parallelism.

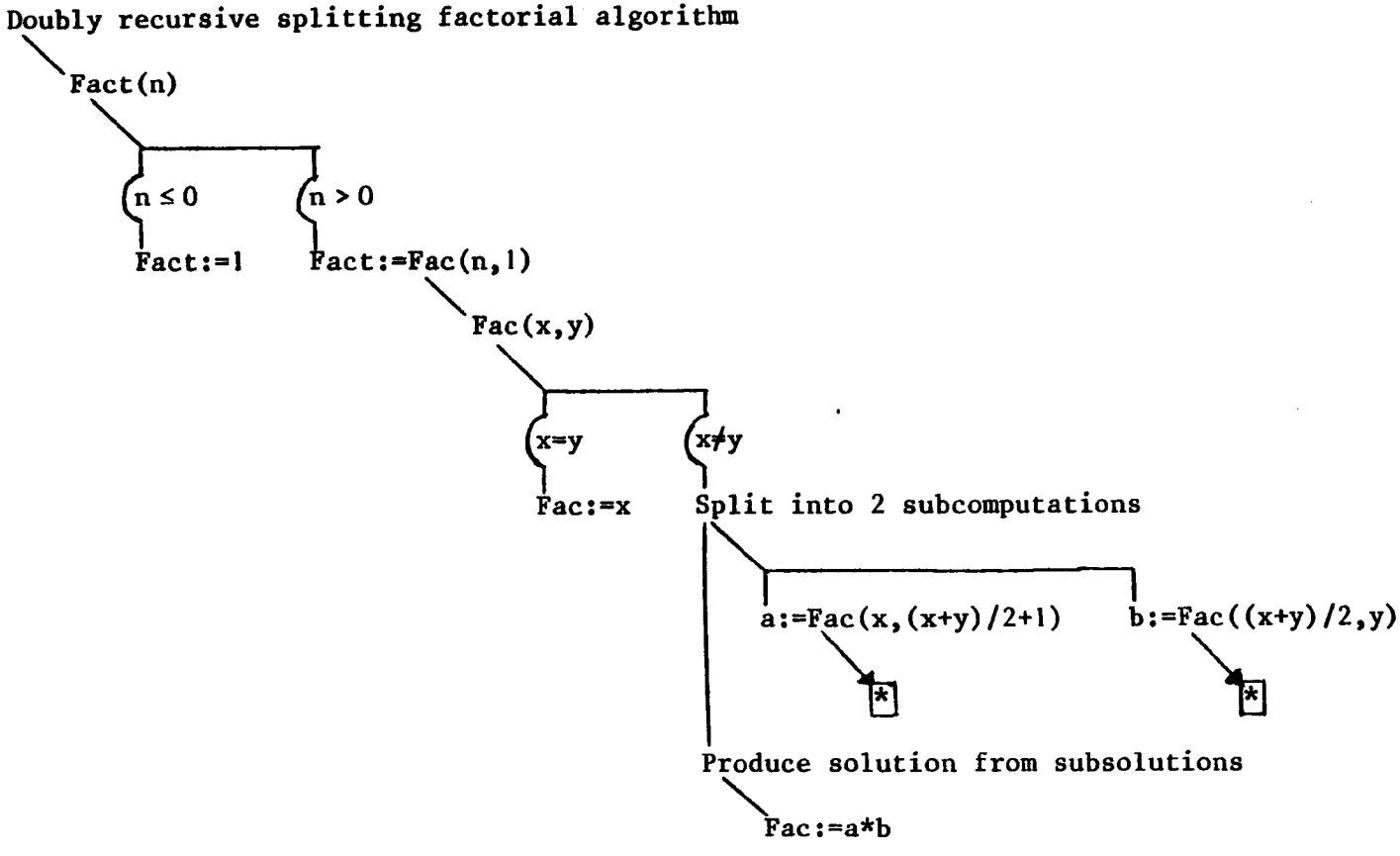


Figure 4-32. Binary Tree of Recursive Invocations.

4.4.3. Design Alternatives & Versions

A kind of inverse Integration occurs when a designer is faced with different methods of implementing a subsystem and chooses one of them for use in the final design. Selecting a sorting technique is a typical example (figure 4-33). Such a design, though complete, fails to record the vital information as to which techniques were considered and why one was selected. During the development of a system the questions "Why is this subsystem implemented in this particular way and did the designer ever think of doing it this other way?" are often the hardest ones to answer and many man-hours are wasted trying to work out what was in the designer's mind at the time he made a particular decision.

Organising the design as in figure 4-34 helps the designer to check that he has considered all the options and allows him to document his assumptions and decisions for the benefit of the development staff and himself at the point in the design document *where such information is most relevant.* Such recording of design decisions does not imply a run-time overhead as the selection can be made prior to compilation.

During the development of a system the design of the input sorting subsystem may have to be changed. Suppose SORT-MERGE was chosen because of the large volume of input data. Suppose that during its life cycle the use of the system changes so that the volume of input data reduces and the SORT-MERGE solution becomes so unacceptably inefficient that the system is changed to use the in-core Quicksort. A new version of the system is produced. It is important to record this development history and to distinguish between versions. The complete "versions problem"[31] is, as yet, unsolved but Dimensional Design can help the developers to document and generate different versions (figure 4-35). Planned variants can also be described (figure 4-36).

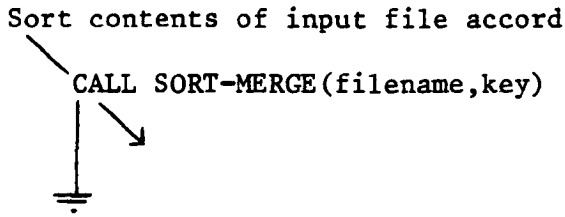


Figure 4-33. Design Decision.

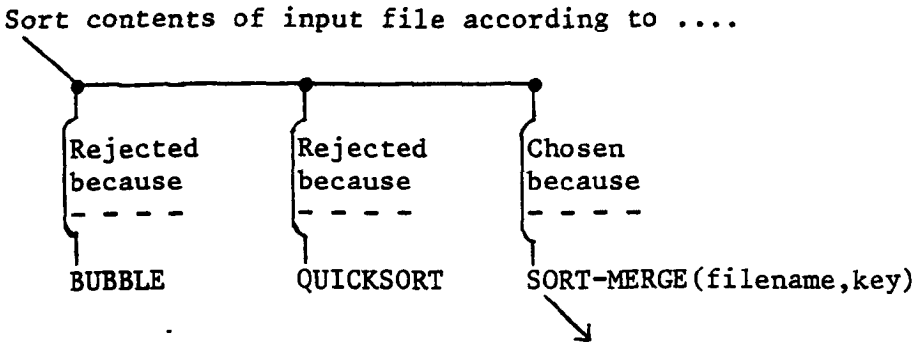


Figure 4-34. Design Alternatives & Decision.

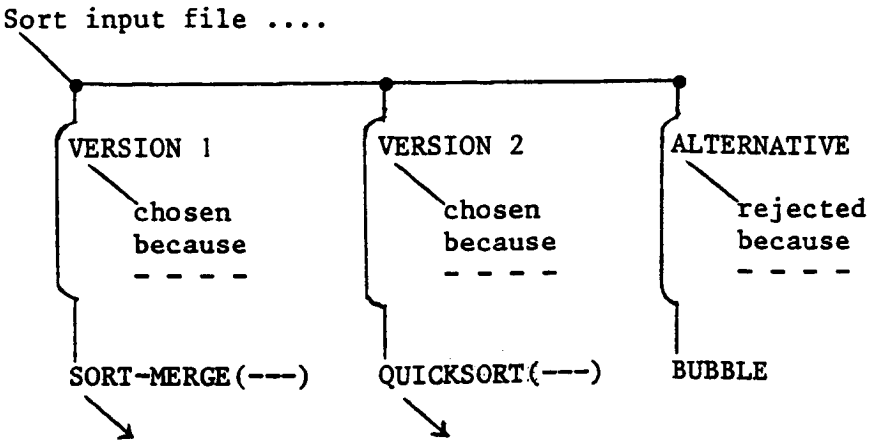


Figure 4-35. Versions.

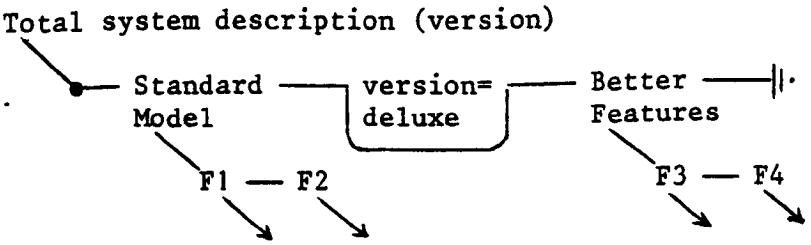


Figure 4-36. Planned Variants.

4.5. Multiple Hierarchies

4.5.1. Step-wise Refinement

The representation of a design created by the Step-wise Refinement (SWR) method[80] can be regarded as a special case of the Design Alternatives problem. During the process of SWR a series of designs is produced in which each individual design is a successively more detailed version of its predecessor. This process continues until an executable program is produced.

Conventionally only the final version of the program is kept - an example of the source code and the design product being one and the same document. During the development of such a program the maintenance programmer typically wastes a great deal of effort in 'learning' the program by a process akin to recreating, in reverse order, the missing, more abstract versions of the program.

By keeping the original series of design versions the program's design is more fully documented and the program is easier to learn and understand as the overall goals of the designer are now available to the maintenance programmer.

Figure 4-37 shows a rather simple example of a program produced by a three stage process of SWR represented in the same way as the design alternatives in the previous section. Such a representation is less than perfect for two reasons:

1. the strong, parental relationship between the versions is not made clear;
2. a genuine design alternative cannot be easily distinguished from an intermediate SWR version.

Due to the Top-down element in the SWR method the strong, parental relationship between versions is a hierarchical relationship. Any given SWR

version can be regarded as the functional specification of its successor and as the implementation of its predecessor ie the SWR inter-version relation is a familiar 'What-How' double relationship. This connection between the versions may be explicitly represented via the 'What-How' dimension (figure 4-38) which solves the two problems inherent in figure 4-37.

Figure 4-38 demonstrates two facets of design not previously encountered, namely:

1. multiple 'how's for a single 'what';
2. complex abstractions treated as single entities.

In all previous discussions a single function has always been either elementary or expressible in terms of lower functions. The expression of a design produced by SWR introduces the concept of a single function (Solve) being expressed by a whole hierarchy of descriptions, each of which is a complete Dimensional Design in itself. Up till now a set or sequence of elementary functions has been abstractly expressed as a single higher level function. Now, in SWR, such sets and sequences are expressed in terms of other more abstract sets and sequences which implement a common higher level function. This means that, in general, level (n+1) of a SWR Dimensional Design cannot be Enumerated by Generating it from level n. This is because the representation of a SWR design is representing the actual design *process* rather than reducing the description of final design *product* itself. The production of level (n+1) from level n can only be achieved by genuine creative effort (understanding) on the part of the designer (maintainer). This is illustrated by observing that level 3 of the Solve design cannot be Generated from level 2. Level 3 is an equivalent program to level 2 but it includes more detail and a reorganisation of the computation to eliminate the redundant calculation of common subexpressions.

The above discussion indicates that figure 4-38 is an inadequate Dimensional Design because:

1. the relationship between SWR versions is not the strict functional 'What-How' relation. The 'diagonal' dimension is being used ambiguously;
2. the diagram is required to show one sequence being the abstract representation of another but contains no mechanism to implement this new requirement

and so figure 4-38 is actually *incorrect*.

These problems are solved by introducing a further generalisation of Dimensional Design - the concept of a Dimensional Design being expressed as a hierarchy of Dimensional Designs, each of which may define its own independent set of axial relations. An individual Dimensional Design may be thought of as a cuboid with its spatial axes representing its own relations which are valid only within the volume of the cuboid. Analogous to the block(!) structured language constructs, a compound Dimensional Design may consist of a tree of cuboids (nodes) connected by edges representing the three relations relevant to the compound Dimensional Design (figure 4-39).

Figure 4-40 shows this generalisation being used to, correctly, represent a design which records both the design *process* (three SWR versions) and the design *product* (level 3).

The excursion into Design Alternatives and the Step-wise Refinement method has been used to introduce the concept of a Dimensional Design expressable, recursively, as a nested hierarchy of Dimensional Designs. Treating a Dimensional Design as an elementary subsystem or an abstraction now requires the designer to enclose the Dimensional Design in an explicit cuboid. It is clear that all elementary subsystems and abstractions may be so enclosed. Indeed, the lack of an enclosing cuboid is now seen to be merely a useful

shorthand. The cuboids are omitted unless they are really necessary, thereby removing from the Dimensional Design an enormous clutter of lines which impart no real extra information. The cuboids are only mandatory when defining a sub-Dimensional Design eg when changing the axial relations or when step-wise refining a Dimensional Design, set or sequence rather than a simple functional specification.

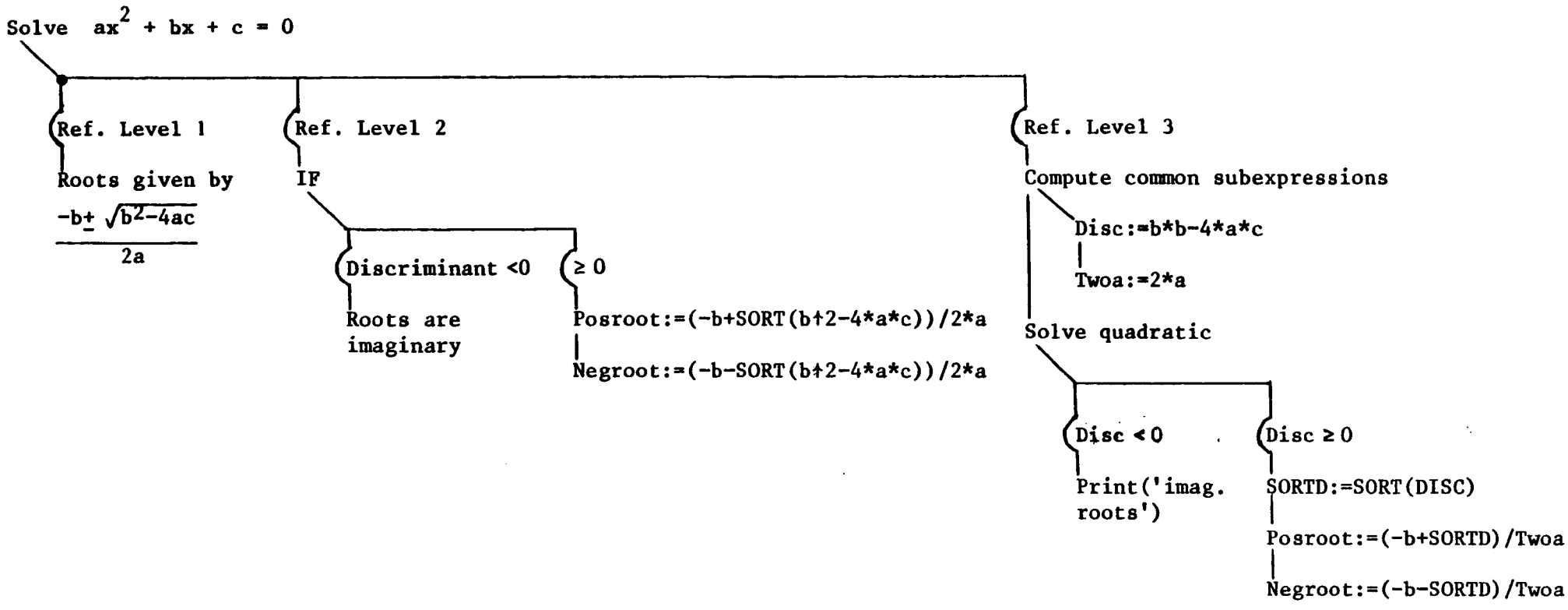


Figure 4-37. Step-wise Refinement of 'Solve Quadratic Equation'.

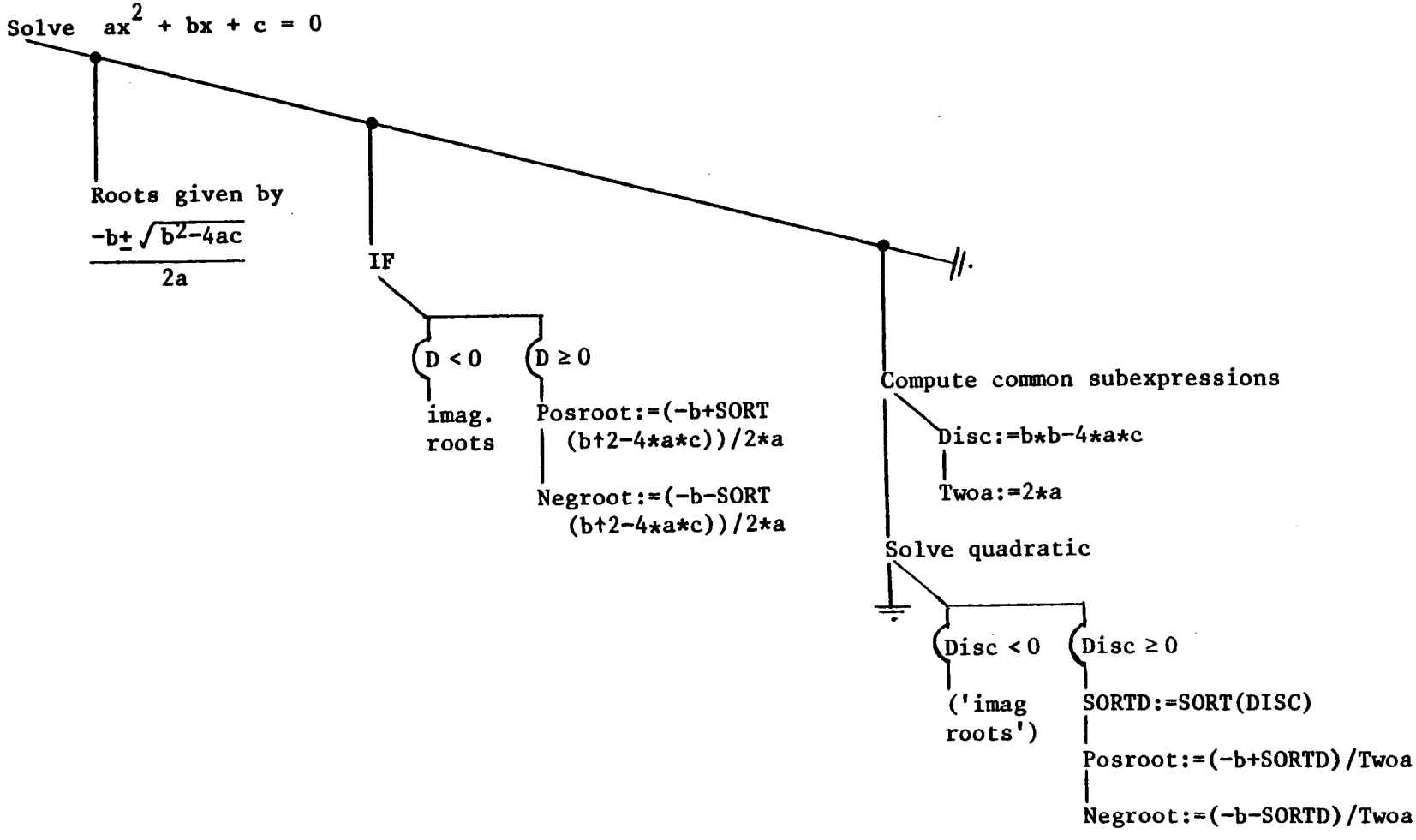


Figure 4-38. Step-wise Refinement showing Hierarchical Relationships between Versions.

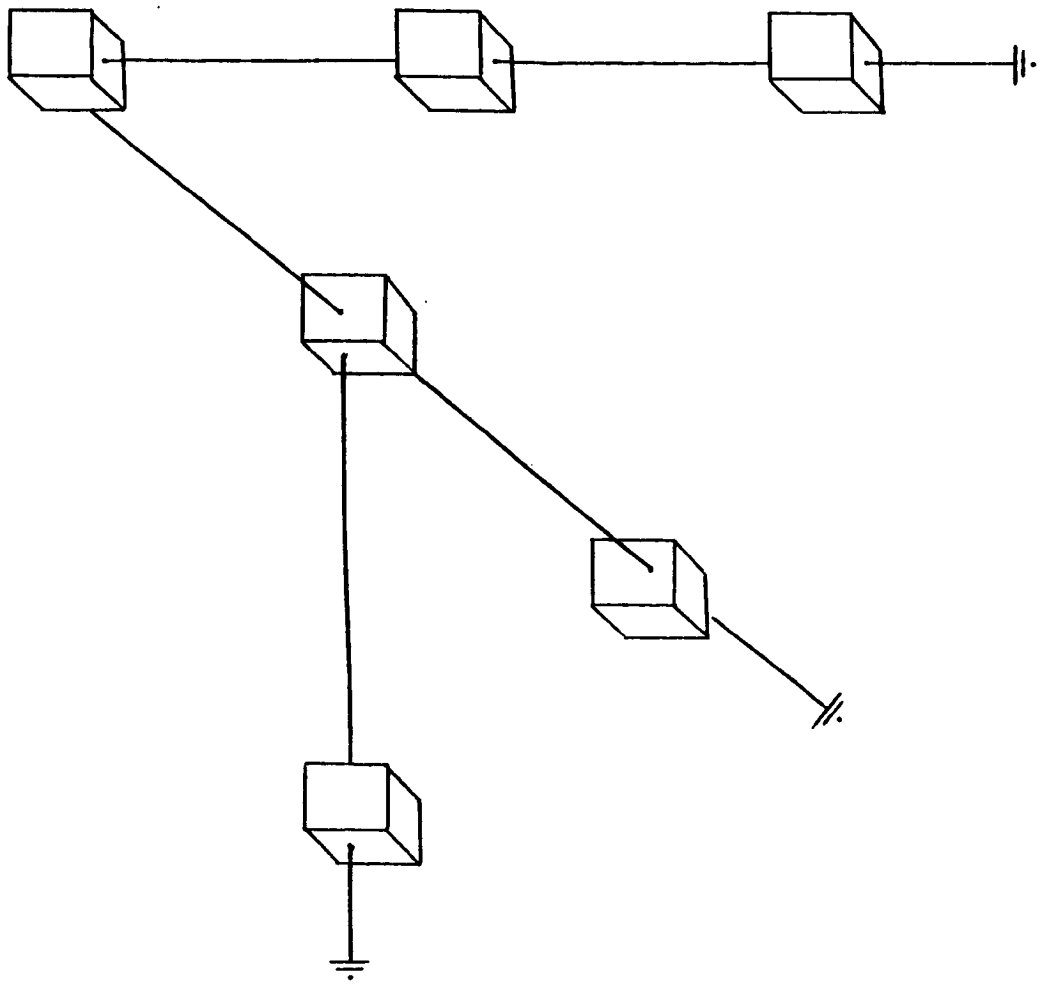


Figure 4-39. Dimensional Design: a hierarchy (tree) of Cuboid Dimensional Designs.

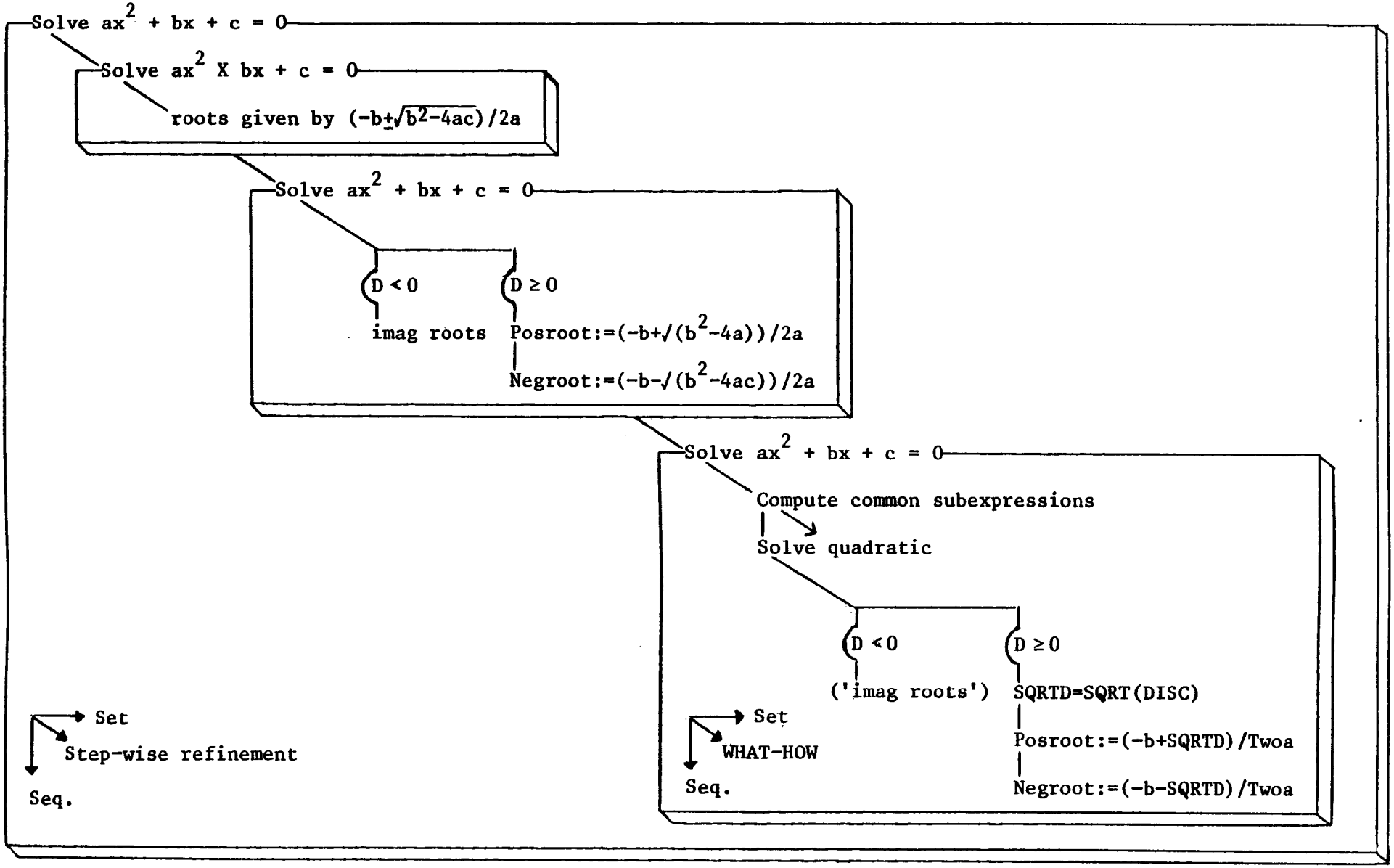


Figure 4-40. Nested Dimensional Designs with Axial Relationships Changing.

4.5.2. Abstract Machines

Refinement is roughly the inverse of Abstraction. Abstraction seeks to reduce an executable program to a compact, manageable description whilst Refinement tries to expand a skeletal description into an executable program. Abstraction uses the occurrence of multiple instances of identical subcomponents to reduce the size of a program's description. These are not a bountiful natural phenomenon but are the product of human design effort in identifying, creating and employing useful subsystems.

In traditional programming, the subroutine is the most common way of specifying a useful subsystem. It is often advantageous to study a system purely in terms of its subroutine interactions. A 'subroutine call graph' is the traditional descriptive technique to aid this study (figure 4-41). A graph such as figure 4-41 immediately tells the maintenance programmer that he is in for a tough job working on program A. Such complex interdependencies increase the effort needed to comprehend program A when contrasted with a more hierarchical, nearly decomposable alternative. The complexity of programs can be reduced by restricting the freedom of a given subroutine to call any other subroutine. Dijkstra[17] used a restrictive technique in the T.H.E. operating system which was designed as a set of layered abstract machines.

When using the abstract machine design approach a designer partitions his subroutines into sets. Each set is considered to be the instruction repertoire of an abstract machine. An abstract machine may be implemented in terms of other lower abstract machine instructions. The lowest level abstract machine is the real machine. In this way a hierarchy of subroutine sets is created in which each subroutine can only call lower level subroutines or hardware instructions. Each abstract machine instruction is implemented independently of whichever higher routine might call it and without any

knowledge about how its lower level routines are implemented, thereby achieving a 'What-How' separation and simplification. Figure 4-42 gives a model for a design expressed as layers of abstract machines and section 10.2.2 gives an actual example of such a program. Note that each abstract machine is a cuboid because the intra-cuboid axial relations are the conventional programming set used to describe the internal structure, in detail, of the individual subroutines whereas the overall relations reflect the layering of the abstract machine hierarchy.

From the design viewpoint layering into abstract machines constrains the designer to produce a simpler structure. This constraint reduces the effort needed to understand and reason about the layered program. The nature and scope of the constraint and the contents of the individual layers are clearly shown in the Dimensional Designs of figures 4-42 and section 10.2.2. The description of a layered program is Generative. It differs from previous descriptions in that whenever a common code component is made into a subroutine and the description reduced, the details of the subsystem are not recorded by leaving unreduced some arbitrary instance of the subsystem, but all instances are reduced and the details are placed in an abstract machine allowing the second layered hierarchy to be created. The RVNC executable program is produced or enumerated by expanding, generatively, the top level program to include the next lower abstract machine's implementation and then removing that machine from the hierarchy (figure 4-43). Contrast figures 4-42 and 4-43 with the unconstrained description of Example Program 1 used in figures 3-3, 3-4 and 3-5 to introduce hierarchical reduction.

It is important to realise that the organisation of a program into layers of abstract machines is an artificial descriptive organisation, not a physical one, derived from a design constraint. The executable program (if for an RVNC) is

still a linear sequence of instructions which contains no evidence of layering restrictions.

It is interesting to consider Recursion, used so successfully in reducing descriptions, from the standpoint of subroutine calling hierarchies and abstract machine principles. Notice that the 'B2D' program in section 10.2.2 contains recursion. Figure 4-44 is a simple example of the call graph of a recursive program. Figure 4-44 is not a hierarchy because of the loop in the graph. Parnas[62] says "For those who argue that the hierarchical structuring proposed in this section prevents the use of recursive programming techniques, we remind them of the freedom available in choosing a decomposition into subprograms. If there exists a subset of the programs, which call each other recursively, we can view the group as a single program for this analysis and then consider the remaining structure to see if it is hierarchical. In looking for possible subsets of a system, we must either include or exclude this group of programs as a single program". This seems to imply that recursion is some odd programming technique rather that the most powerful of the formal reasoning tools. Figure 4-44 is probably, for most programmers, a model of the way control passes between routines (in a non-hierarchical fashion here). Figure 4-44 is a Generative description and a poor description if recursion is not implemented by reentrancy but by, say, a set of Communicating Sequential Processes running in parallel. If the program implied by figure 4-44 is finite (infinite) then it can be remodelled as an RVNC (GVNC) program description which is a true, but redundant, hierarchy (figure 4-45) capable of being reduced in a similar manner to a Dimensional Design (figure 4-46). Figure 4-46 is a Generative description of the call graph which makes the underlying enumerable hierarchy easier to grasp than figure 4-44. A similar argument can be constructed for abstract machine designs which, by employing recursion to

reduce their descriptions, seem to invalidate the basic design rule.

The above small digression has been taken so as to pose the unanswered question as to why recursion, call graphs and abstract machines fail to blend although recursion is such a natural tool in formal reasoning and descriptive reduction?

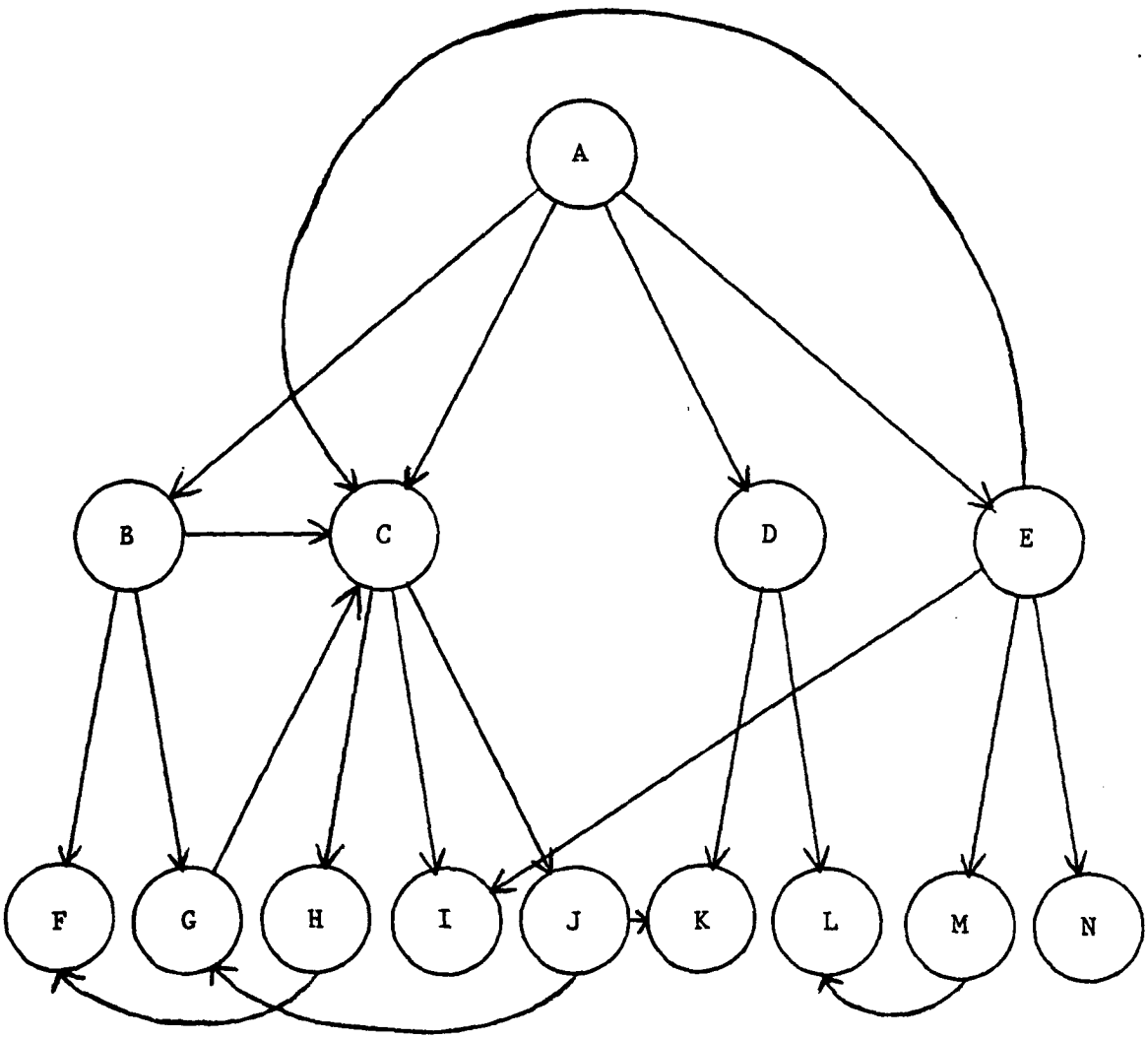


Figure 4-41. Call Graph.

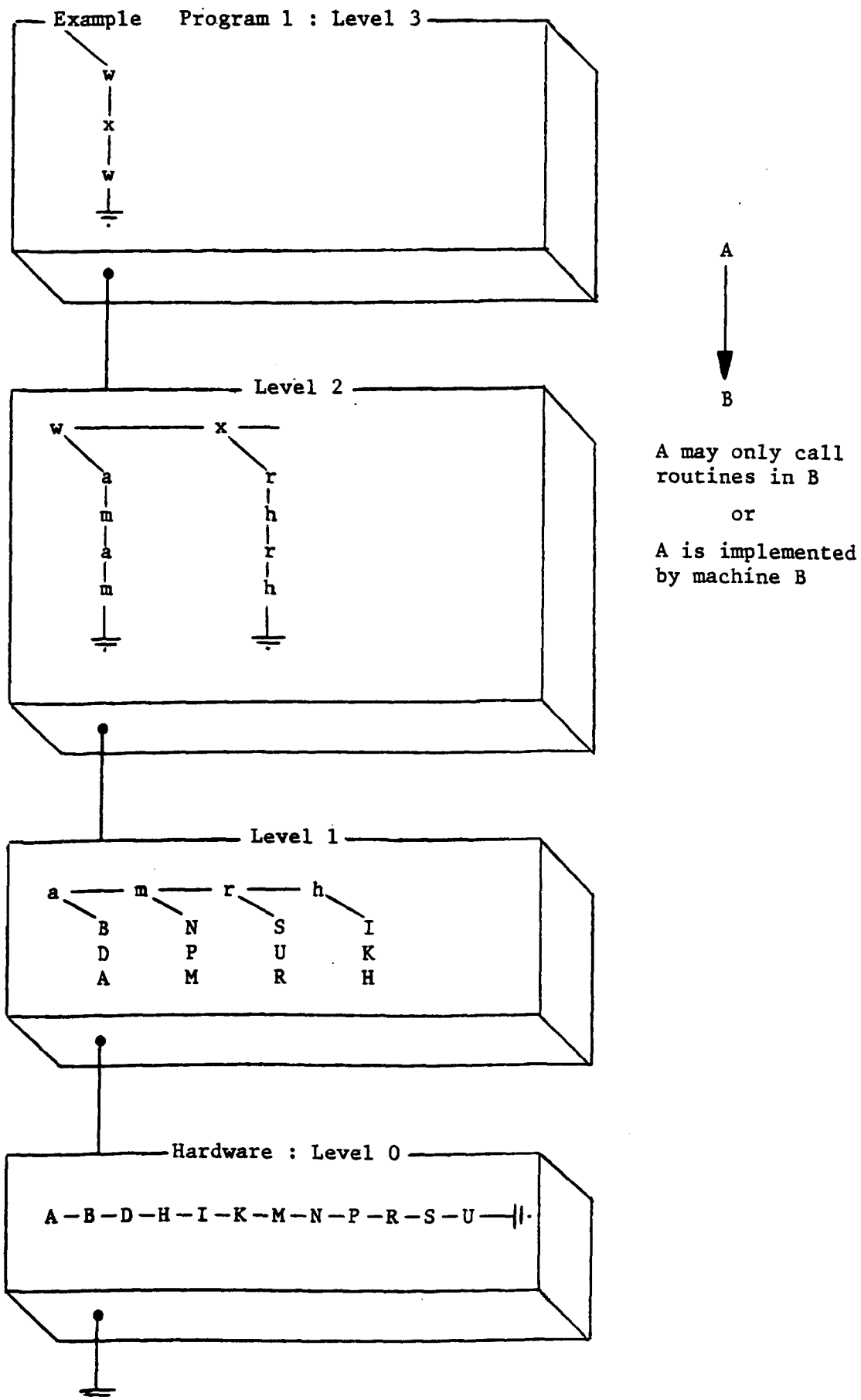


Figure 4-42. Four Layers of Abstract Machines.

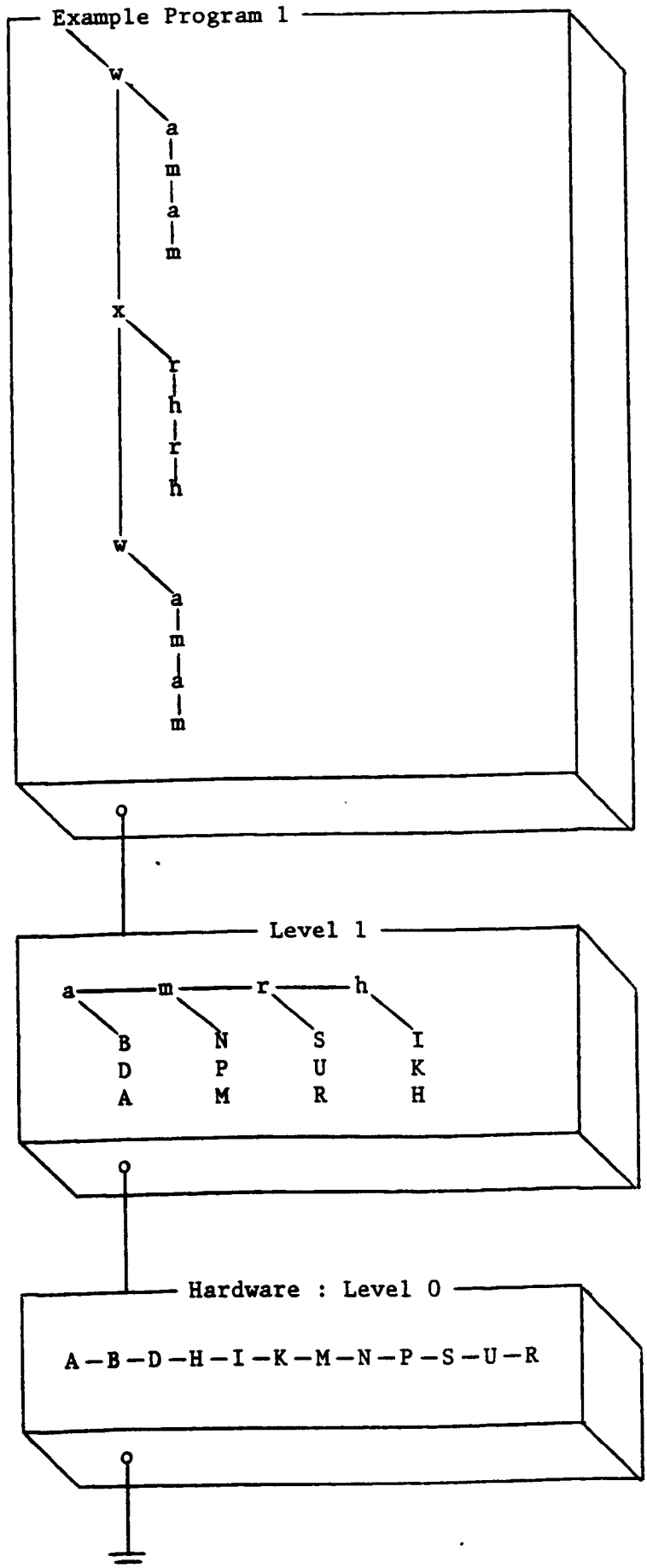


Figure 4-43. Partially Enumerated Figure 4-42.

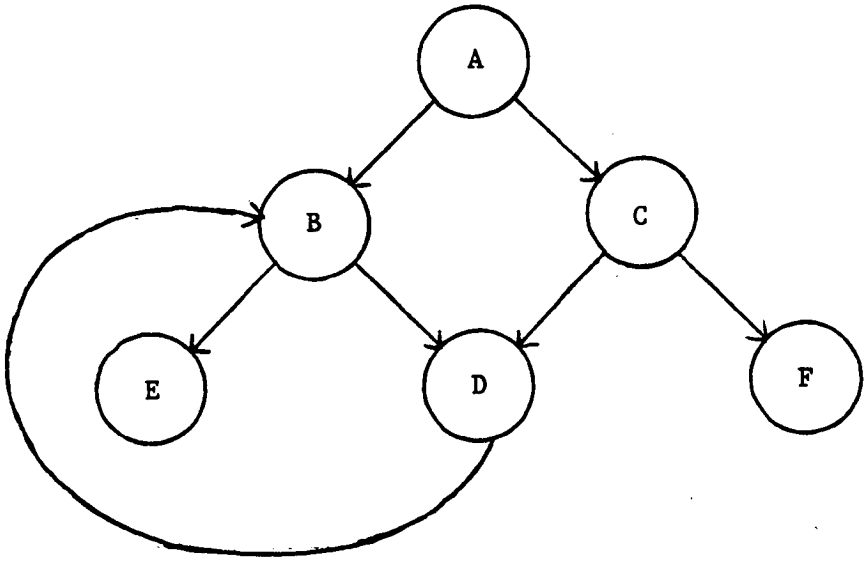


Figure 4-44. Call Graph Showing Recursion (generative).

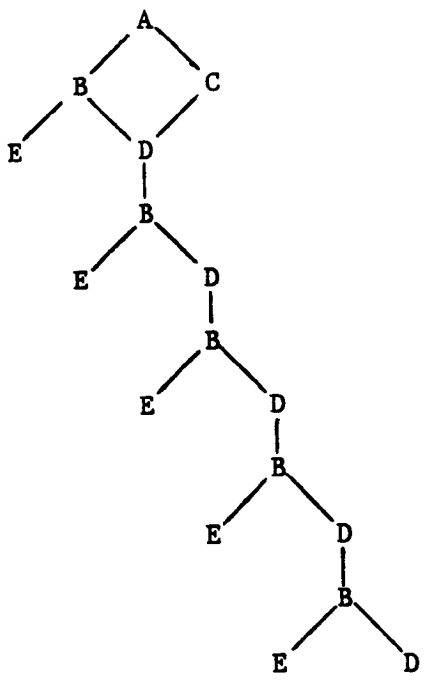


Figure 4-45. Enumerated Call Graph.

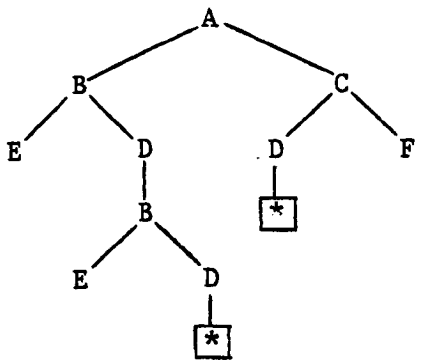


Figure 4-46. Generative Call Graph (hierarchical).

4.6. Postscript to Instruction Examples

The reader has now been informally introduced to all the Dimensional Design techniques needed to represent the major programming abstractions of Set, Sequence and Hierarchy, together with the tools, such as H-Reduction, Integration and Induction, to simplify the description and understanding of programs. These concepts have been exclusively illustrated with examples describing sets and sequences of program instructions so as to concentrate on the representational techniques rather than the objects being described. In reality it is impossible and undesirable to consider a program's instructions without simultaneously considering the data operated on by the instructions. Fortunately the abstractions and their associated tools developed for instruction representation are equally applicable to data descriptions.

4.7. State

4.7.1. Introduction

A program for the RVNC consists of an ordered sequence of instructions which, during execution, operates on the data store, an ordered set of data, to perform a computation. The RVNC program is immutable. The RVNC data store is changed by the execution of each individual program instruction. The state of the computation at any given time describes the contents of the data store and the remaining instructions to be executed. The State of a computation and, in particular its data store, is therefore more difficult to describe than the program because the State varies with 'computational time'.

Being a deterministic finite state machine the RVNC changes an initial state into a final state. This ability is of little real value unless the RVNC is connected to the 'outside world' in some way. Thus the State of an RVNC computation is more accurately described as the contents of the internal data store

plus the (relevant) contents of the 'outside world' plus the remaining instructions to be executed. Modern programming languages are able to fully (if not easily) describe the internal store and remaining instructions. They are weak at describing the connection to the 'outside world'. For most programming languages the 'outside world' is everything other than the internal data and instruction stores. Knowledge about the 'outside world' of files, peripherals, users etc has to be implicitly contained in the instruction sequence. Indeed the essence of programming is the mapping of 'outside world' concepts into the internal store and instruction sequence of the RVNC. This is not easy because the elements of the internal store are very primitive and the instruction repertoires of most machines allow any single element to be treated ambiguously ie its type may vary with computational time.

4.7.2. Data Structures

The concept of data type is very important. The elements of the RVNC's store are an ordered set of uniform data type. The programmer structures this store implicitly, exactly the same as with the instruction sequence, by constructing an *artificial* hierarchy to reduce the description of the data store to meaningful and manageable proportions. The concept of Type allows the programmer to describe the *varying* contents of the store in more abstract, *fixed* terms. For example, to calculate the sum of two numbers the programmer Integrates vast numbers of programs of the form given in figure 4-47 to produce figure 4-48. The actual values involved are a function of both the initial state and computational time, yet the data store organisation in terms of 'sum of two numbers' and 'integer' remains fixed. Instructions can be treated as functions whose ranges and domains are fixed. This facilitates formal reasoning by greatly reducing enumerative reasoning because individual values are not treated separately but as members of a class of values or Type. This is

Abstraction at work again. Fortunately all the abstraction techniques used to describe instruction sequences can be employed in data descriptions.

Hoare,[34] arguing in favour of improved data structuring techniques, sums up the similarity between data structuring and instruction structuring as follows "In the case of a well-structured program, the overall pattern of the class of possible computations is so clearly revealed in the pattern of the program itself that it is possible to understand and prove the correctness of such a program without even thinking of the large class of potential computations involved. A type declaration should use similar clear patterns to define the structural properties of all possible data instances of that type, independent of their size or the component values involved".

Sadly no widely used programming language has fully general type definition facilities exploiting the structuring abstractions. Even Pascal cannot offer a fully inductive type definition let alone a logical treatment of files.

Figure 4-49 illustrates the use of the abstractions Hierarchy, Set and Sequence to represent a data structure. Note the Hierarchy relation in figure 4-49. It is of the familiar What-How variety but now applied to the mapping of real world concepts into the internal data store of the RVNC. Figures 4-50 to 4-52 show the use of Reduction, Integration and Induction for improving the descriptions of data structure types. Notice the similarity between types and grammars. "One remarkable feature of the structuring methods introduced here is that they make no mention of the reference or pointer, which are traditionally regarded as the prime means of structuring data. In this respect, references seem similar to *goto* statements".[34]

Figure 4-52 shows how easily files can be represented. A sequential file (figure 4-52c) is, naturally, a sequence of items and is clearly different in structure to the random access file in figure 4-52d. Note that there is no

difference in representation between a random access file and a data structure held in main memory because, in this instance, only their logical structures are considered, not their physical storage media and organisation.

A data type is not actual data, but is a form of artificial descriptive hierarchy. In programming language terminology a datum is an instance of a type. In Dimensional Design terms, a datum is a type description which is completely refined so that the elementary subsystems are actual values rather than abstractions (figure 4-53).

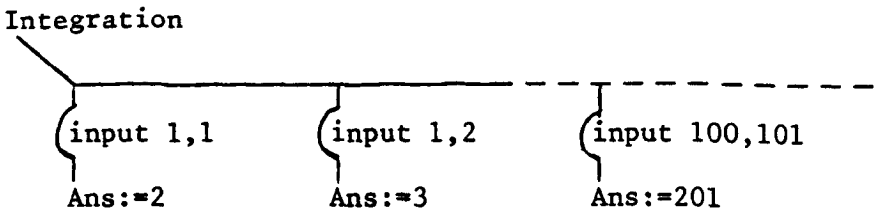


Figure 4-47.

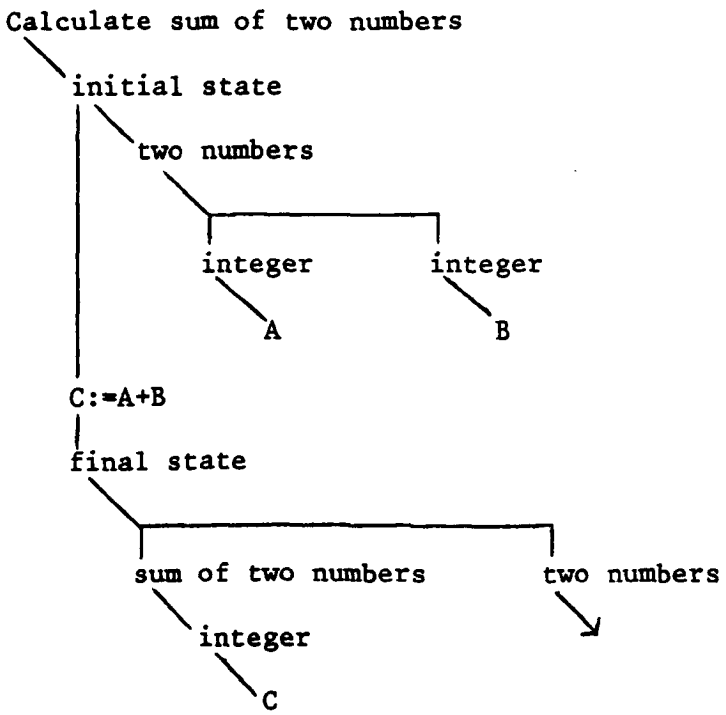
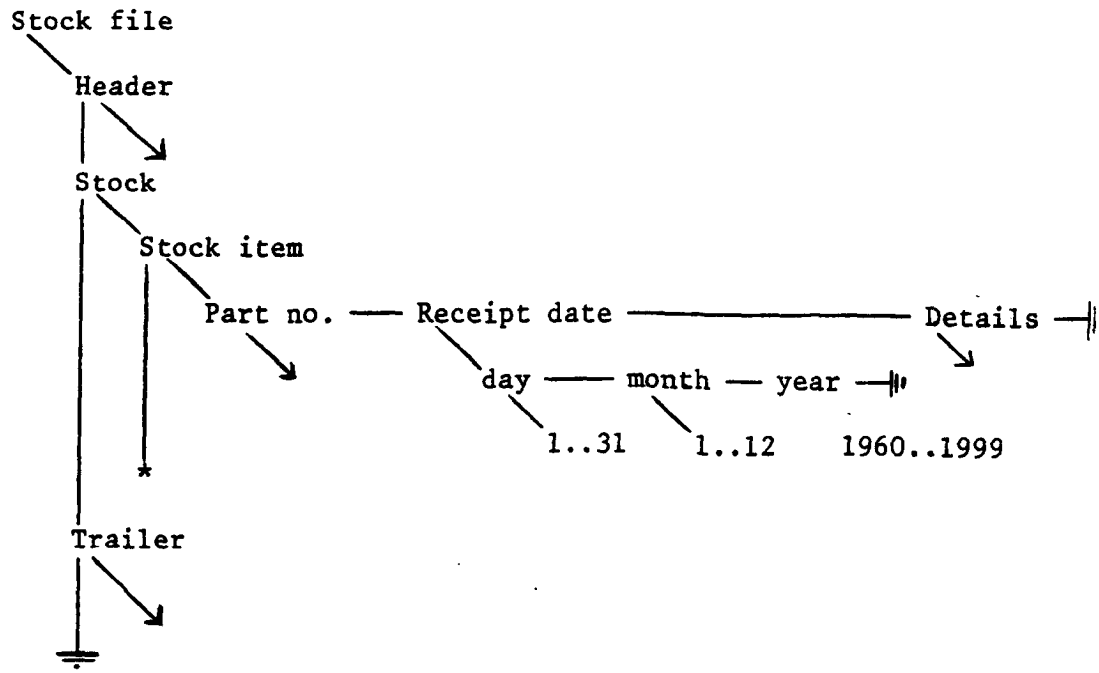


Figure 4-48.



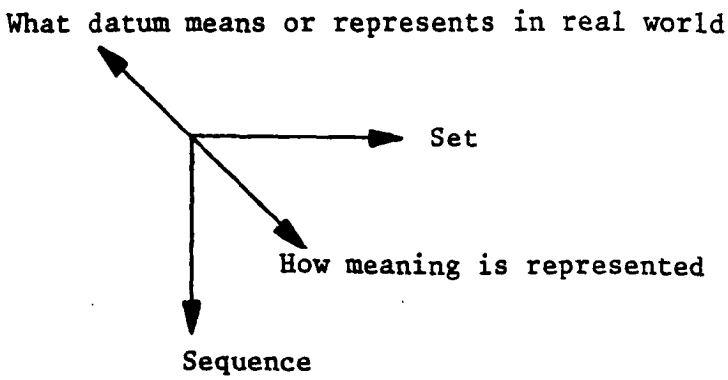
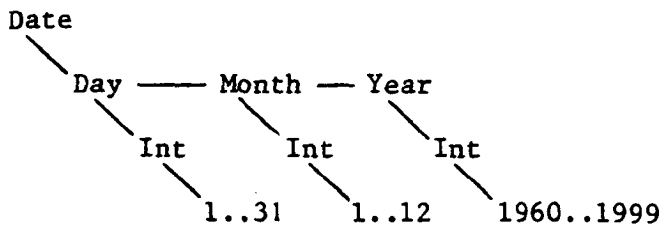


Figure 4-49. Simple Data Structure.



Cartesian Product: Date = day x month x year

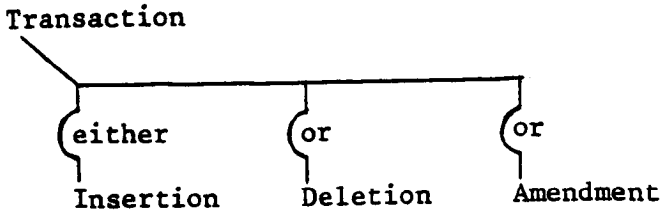
Pascal:

```
Type Today,Receipt:Date

with Today do begin
    Receipt.day:=day;
    Receipt.month:=month;
    Receipt.year:=year;
    end;
{or} Receipt:=Today;
```

Cobol:

```
MOVE TODAY TO RECEIPT.
```

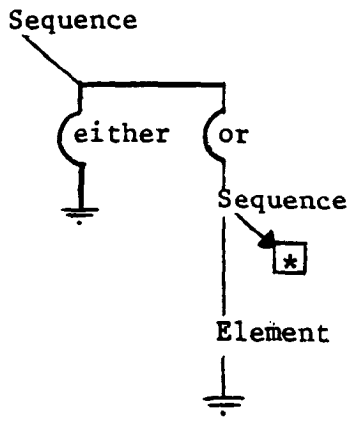
Figure 4-50. Reduction. Contrast Pascal & Cobol.



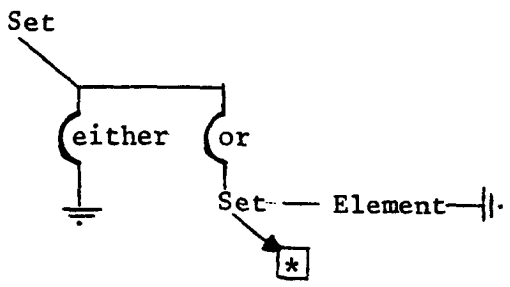
Discriminated Union:

Transaction = insertion | deletion | amendment

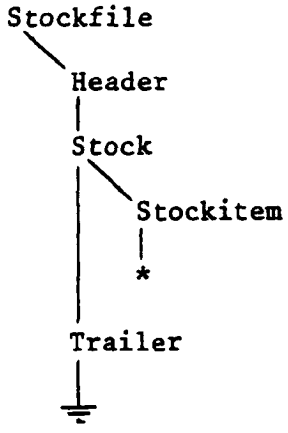
Figure 4-51. Integration.



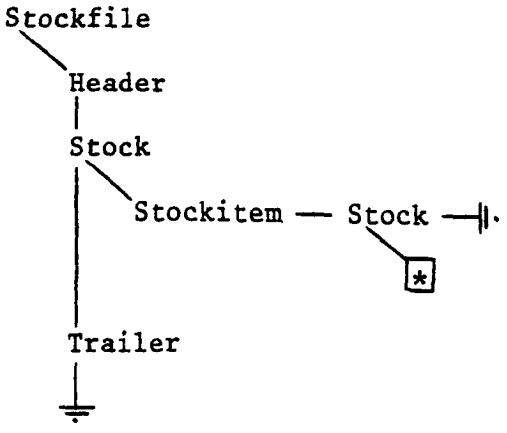
(a)



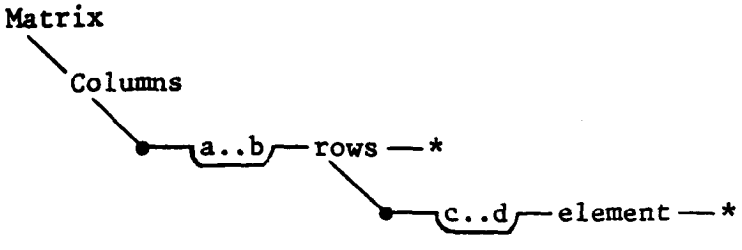
(b)



(c) Sequential File

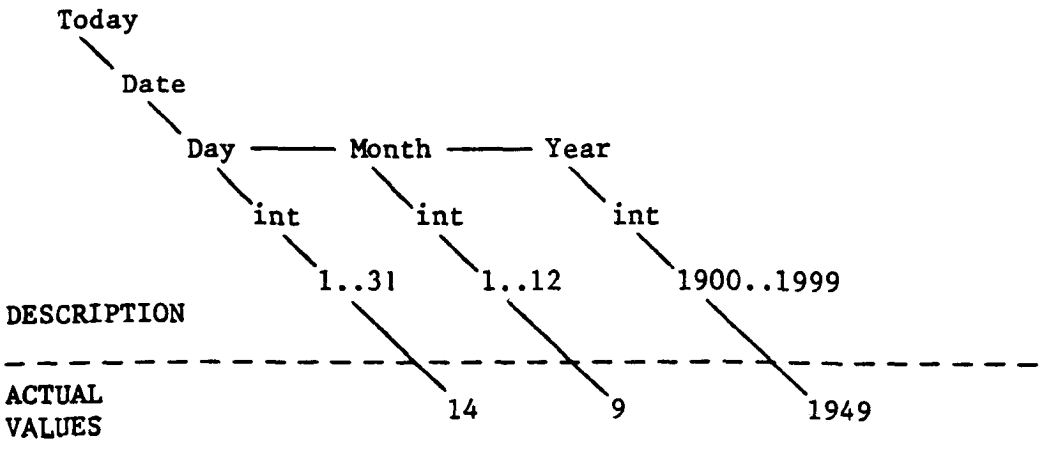


(d) Random Access File



(e)

Figure 4-52. Induction.



'Today' is of type Date and has value (14,9,1949)

Figure 4-53. Instance of Type.

4.7.3. State Descriptions & Computational Histories

A datum is an instance or refinement of a type. In a complementary fashion a type may be more abstractly described (figure 4-54). The various types, their actual instances and abstractions collectively form the description of the state of an RVNC's data store. As Dimensional Design can represent files as easily as any other data structure the state description can be widened to include those aspects of the state such as files which conventional descriptive techniques omit. The state of a computation is comprised of a set of parts including the contents of main memory and the associated files of input and output data, together with the instructions to operate on this data (figure 4-55).

A computational history, the sequence of states produced during an execution of a program, may be represented by a Dimensional Design. Figure 4-56 crudely illustrates the action of a program which produces a serial output listing from a serial input file.

If the actual data are unknown, but the state components are formally described then an 'abstract' computational history may be developed which resembles the 'execution' of a Predicate Transformer[19] (figure 4-57). Note how, in figure 4-57, the direction of the sequential relationship has been reversed - a simple device which neatly emphasises the duality between the derivation of the weakest precondition from the transformation of the postcondition by the instruction and the transformation of an initial state into the final state by the instruction.

The above discussion has briefly illustrated how the same techniques used to describe instruction sequences can be used to describe data, data types, files and states. Combination of the descriptions of instruction sequences and the states on which they operate enabled computational histories and proofs

to be similarly expressed.

So far the discussion has tended to concentrate on the relationships between individual instructions (or data) and their more abstract descriptions. The individual elementary subsystems have been simply described by blocks of text. However these blocks of text are themselves a structured *microcosm*. This echoes Simon's observation[73] (see section 3.2) that "it is somewhat arbitrary...what subsystems we take as elementary. Physics makes much use of the concept of the 'elementary particle' although particles have a disconcerting tendency not to remain elementary very long". Can one split the textual, elementary subsystem?

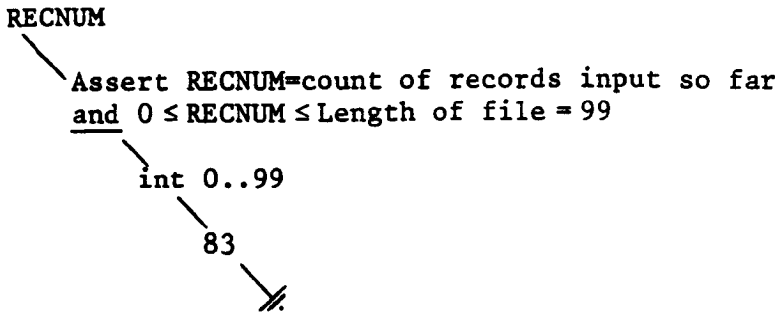


Figure 4-54. Abstraction & Refinement of the type 'int'.

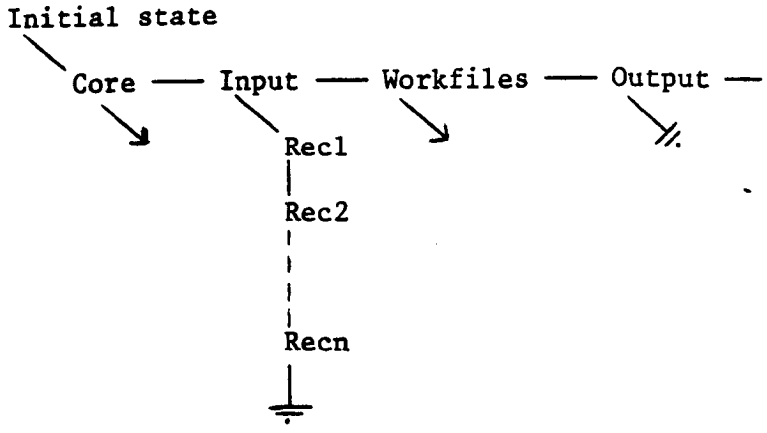


Figure 4-55. State Description.

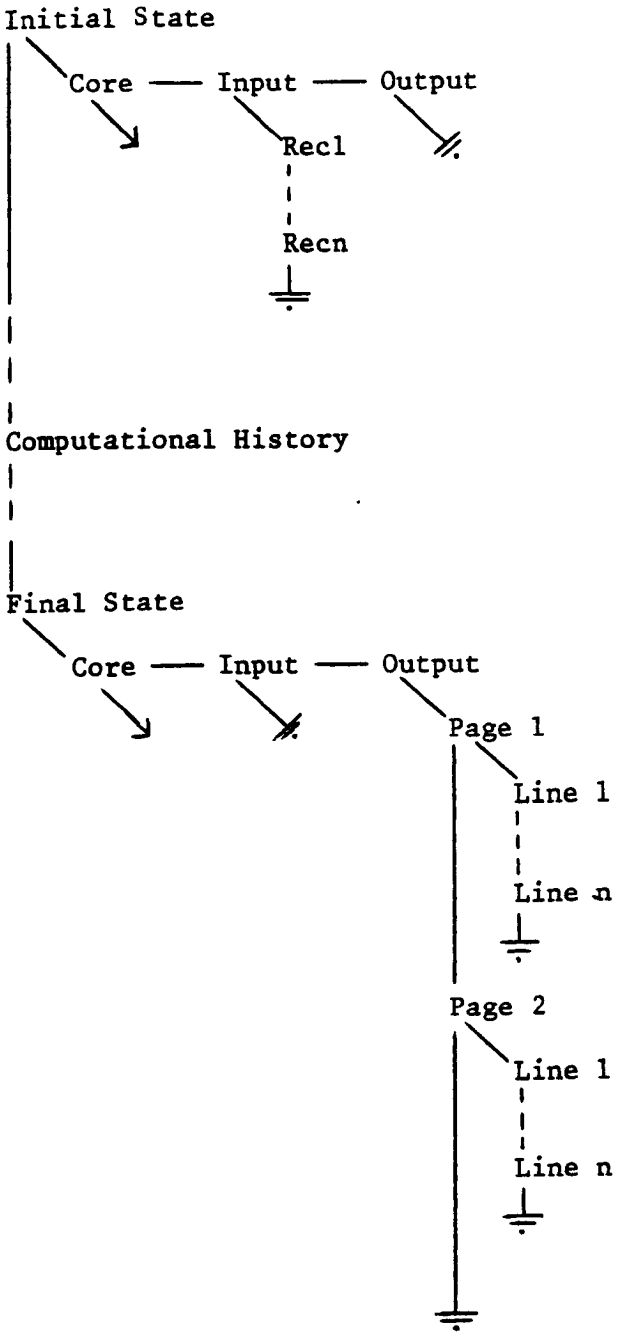


Figure 4-56. Computational History.

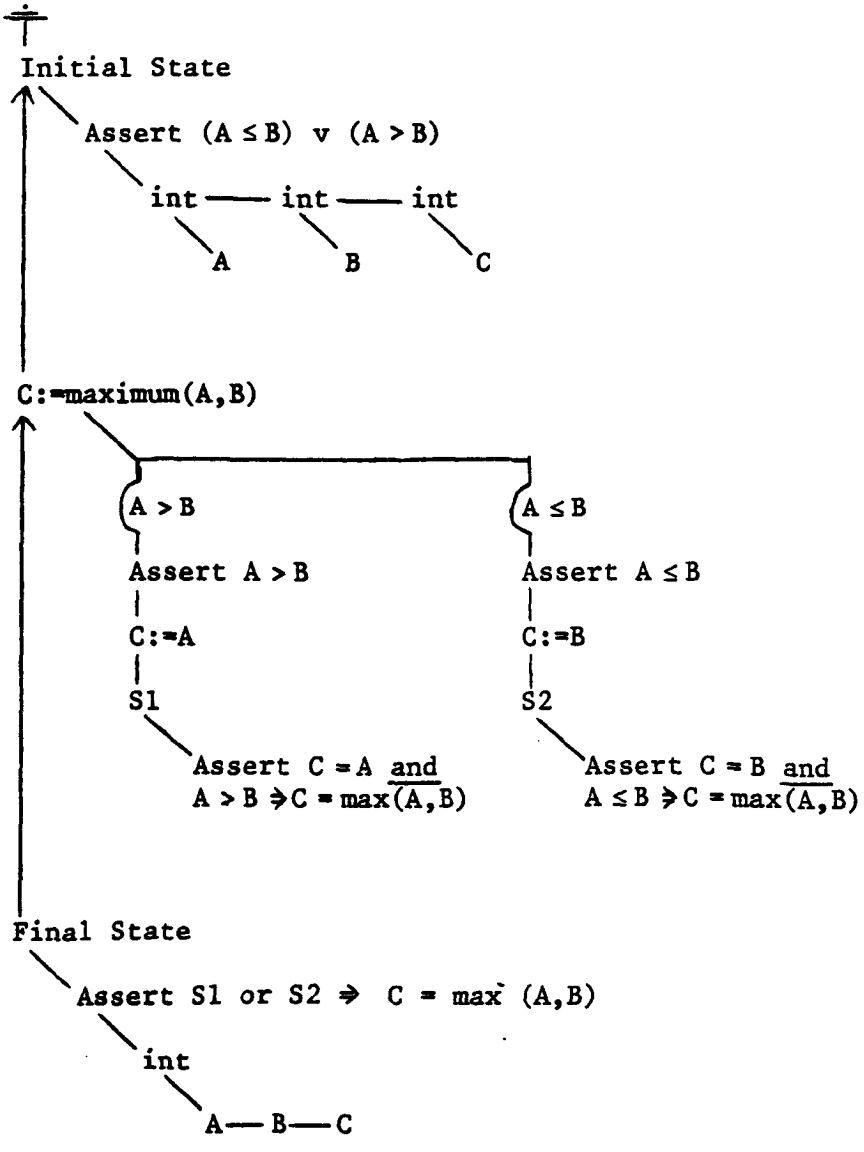


Figure 4-57. Predicate Transformer.

4.8. Subsystems

If a Dimensional Design uses statements from a high level programming language as its elementary subsystems then it is obvious that such statements could, if required, be split up into sets and sequences of components such as operators and operands. It will be no suprise to the reader to learn that Dimensional Design can express such a finer grain of detail and that Hierarchical Reduction, Integration and Induction may be employed to reduce and simplify the resultant description.

Figure 4-58 illustrates three of the many possible ways of splitting a programming language statement to reveal further levels of detail. Figure 4-58a shows an assignment statement to be a sequence of operators and operands and really just highlights the textual layout. Figure 4-58b is the same assignment statement but set in the context of its formal, syntactic, parse tree; it is clear that a syntax tree is an additional, artificial hierarchy. In contrast, figure 4-58c shows the statement's semantic or evaluation tree; note how the assignment statement has been completely reorganised to bring out, very explicitly, the order of evaluation. Contrast this to the ordering implicitly recorded in the textual version which requires interpretation of bracketing and operator precedence rules by the reader.

Figure 4-59 illustrates Hierarchical Reduction at work eliminating common sub-expressions. The distinction between function names and definitions, formal and actual parameters is demonstrated in figure 4-60, which also shows that function definitions may be usefully enumerated within expressions. The Conditional Expression is a familiar statement-level use of Integration (figure 4-61).

Figure 4-62a is an example of a popular programming technique; in this case it is used to sum a vector. A little reflection will convince the reader that

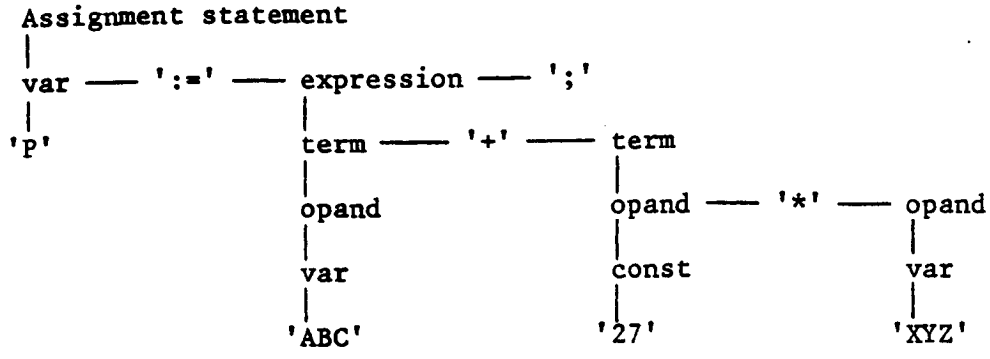
the creator of figure 4-62a often only resorts to a sequential inductive construction because he wishes to avoid the tedium of enumerating figure 4-62b completely as he is denied, usually, the facility shown in figure 4-62c and 4-62d which allows a more natural expression of the computation. The use of induction to produce a description of the computation rather than an operational procedure also facilitates optimisation as the computation would benefit from parallel evaluation, a technique proscribed by the sequential operations specified in figure 4-62a. Figure 4-63 indicates that even straightforward English text has a wealth of hidden structure.

Sentences, words, operators, operands, variables and constants are all fundamentally built up from sets and sequences of characters which are linked by many different relationships, hence the rich variation in the labellings on the axes in figures 4-58 to 4-63. In general, characters are examples of symbols whilst subsystems in Dimensional Design are really complex objects built up from inter-related symbols. It is the individual symbols which form the real elementary subsystems in Dimensional Design hierarchies.

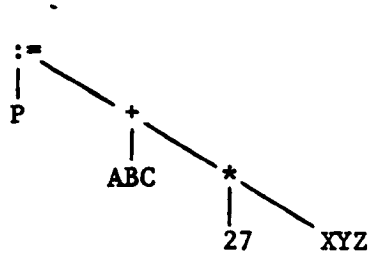
Recognition of the elementary nature of individual symbols concludes the above informal introduction to Dimensional Design and provides the foundation stone on which to build the generalisation and formalisation of the various concepts encapsulated by Dimensional Design.

```
P ── := ── ABC ── + ── 27 ── * ── XYZ ── ;
```

(a) Text



(b) Syntactic or Parse Tree

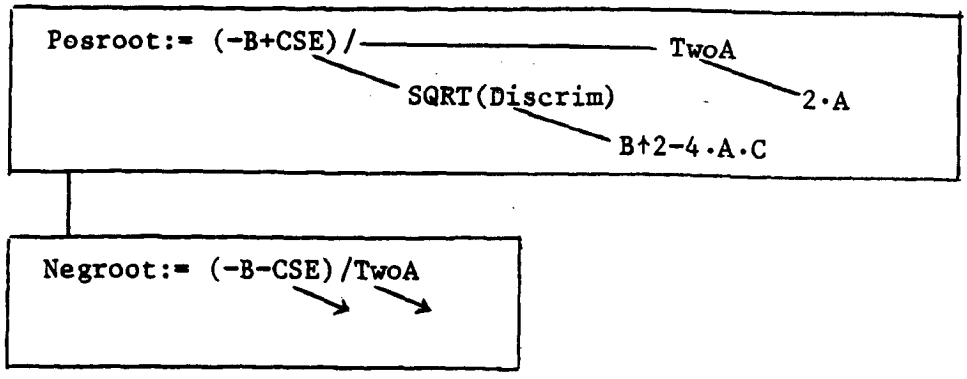


(c) Semantic or Evaluation Tree

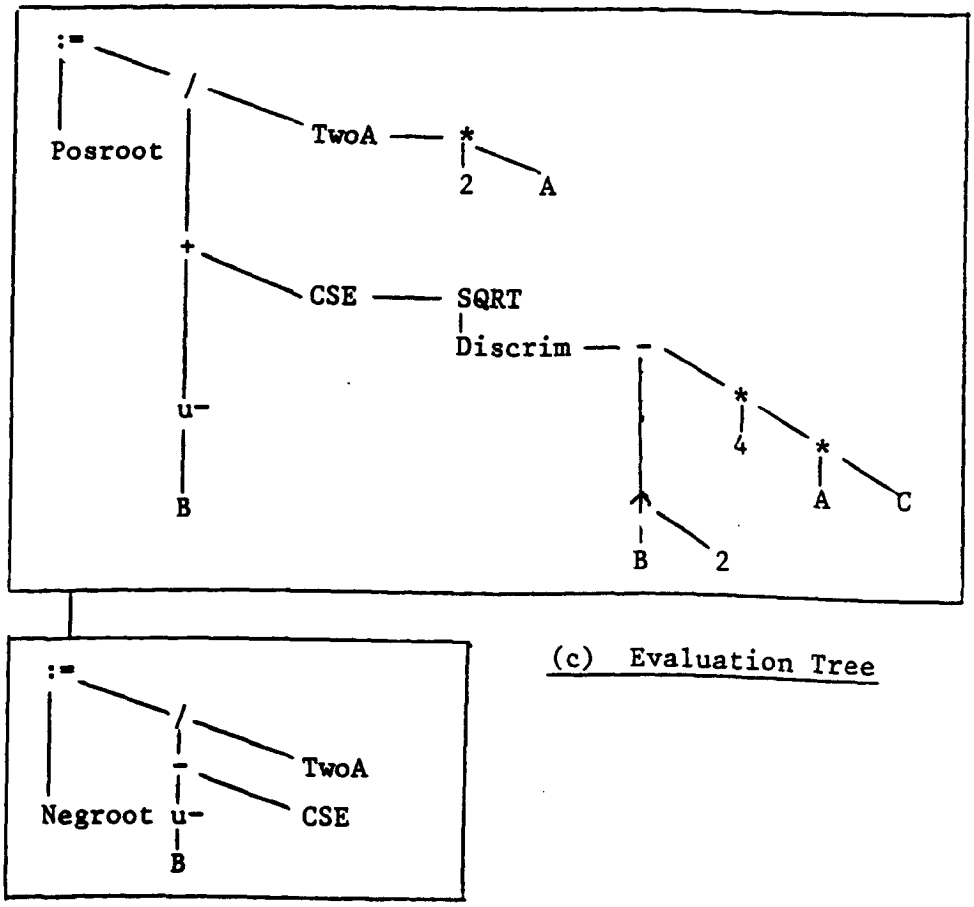
Figure 4-58. Three Ways to Split a Programming Language Statement.

```
Posroot:= (-B+SQRT(B↑2-4·A·C))/2·A
Negroot:= (-B-SQRT(B↑2-4·A·C))/2·A
```

(a) Text



(b) 'Structured' Text



(c) Evaluation Tree

Figure 4-59. Hierarchical Reduction.

```
X := A + fact(max(B,C·D+E))·F

X := A + fact[x](max[x,y](B,C·D+E))·F

X := A + fact[x]                    (  max[x,y]                    (B,C·D+E))·F
        ( x ≤ 1   x > 1 )             ( x ≥ y   x ≤ y )
          1       x*fact(x-1)           x         y
                  *
```

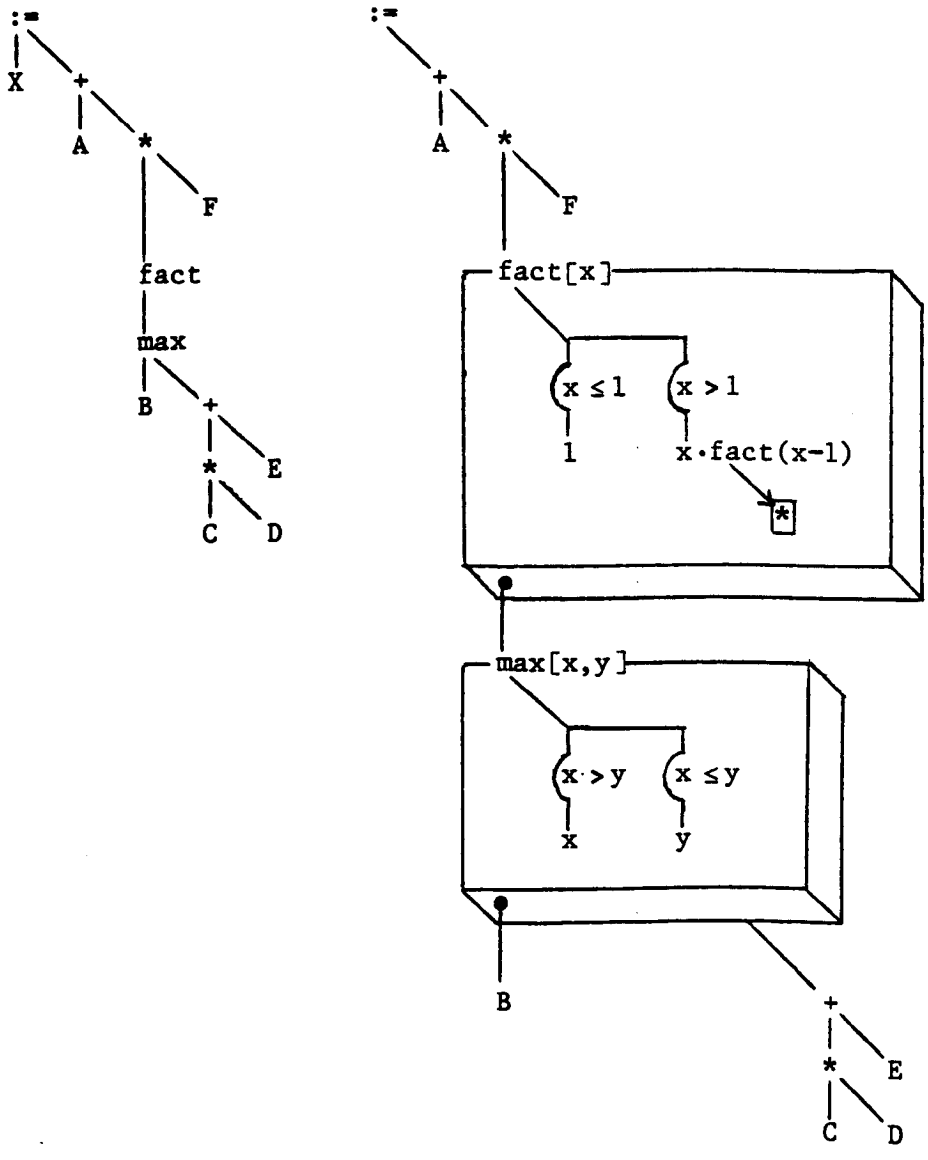
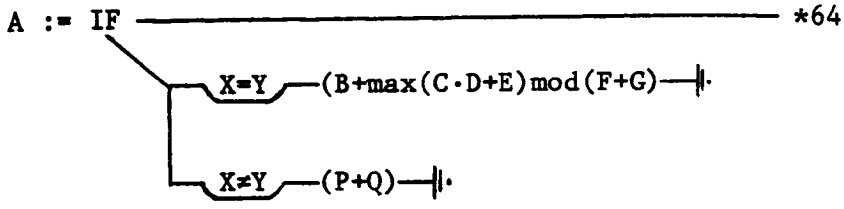


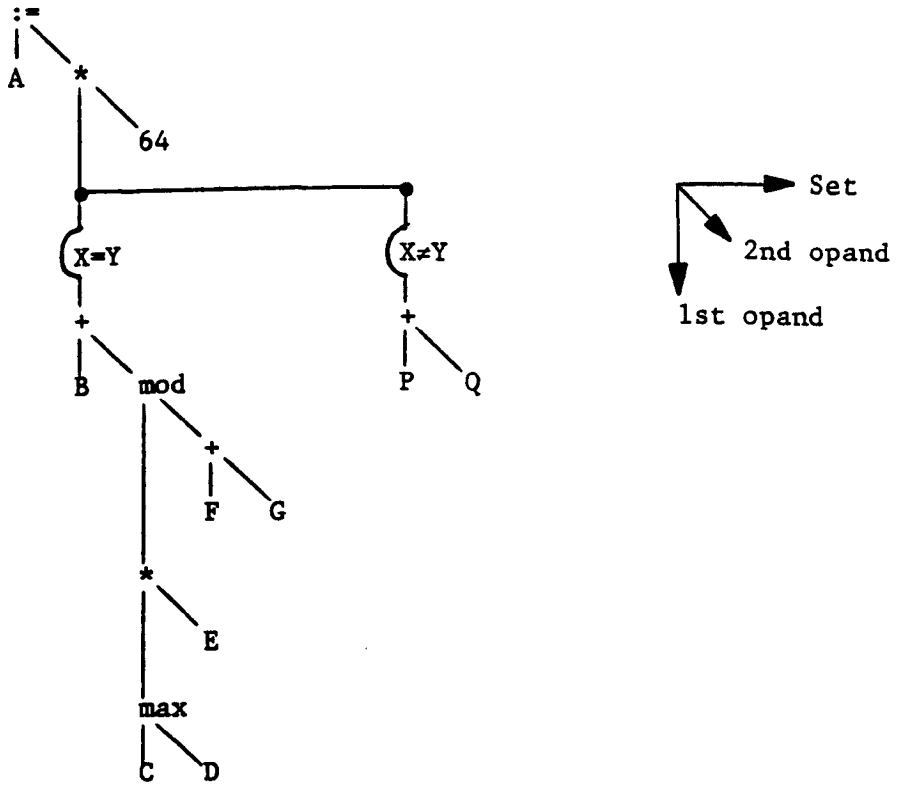
Figure 4-60. Function Name, Definition, Formal & Actual Parameters.

```
A := (if X=Y then (B+max(C·D+E)mod(F+G)else(P+Q))*64
```

(a) Text

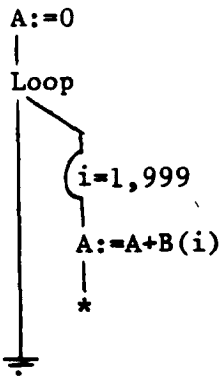


(b) 'Structured' Text



(c) Evaluation Tree

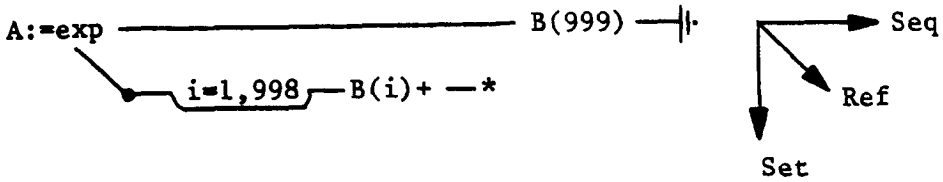
Figure 4-61. Integration.



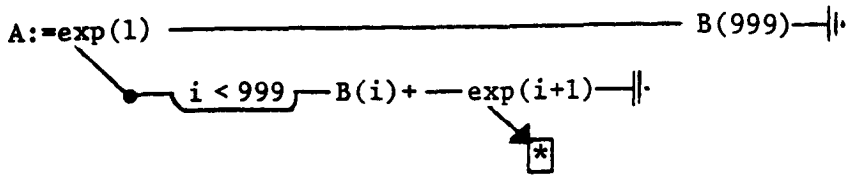
(a) 'Normal'

A:=B(1)+B(2)+ .... +B(999)

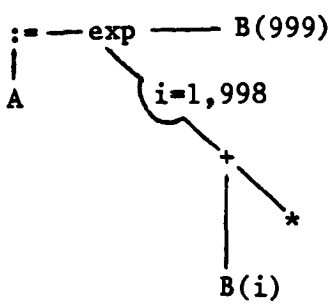
(b) Text



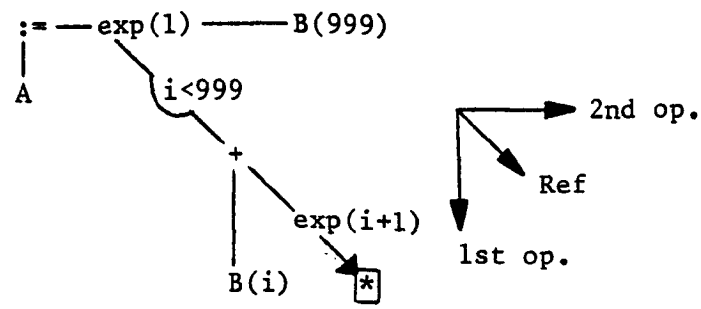
(c) Structured Text: Iteration



(d) Structured Text: Recursion



(e) Evaluation Tree: Iteration



(f) Evaluation Tree: Recursion

Figure 4-62. Induction.

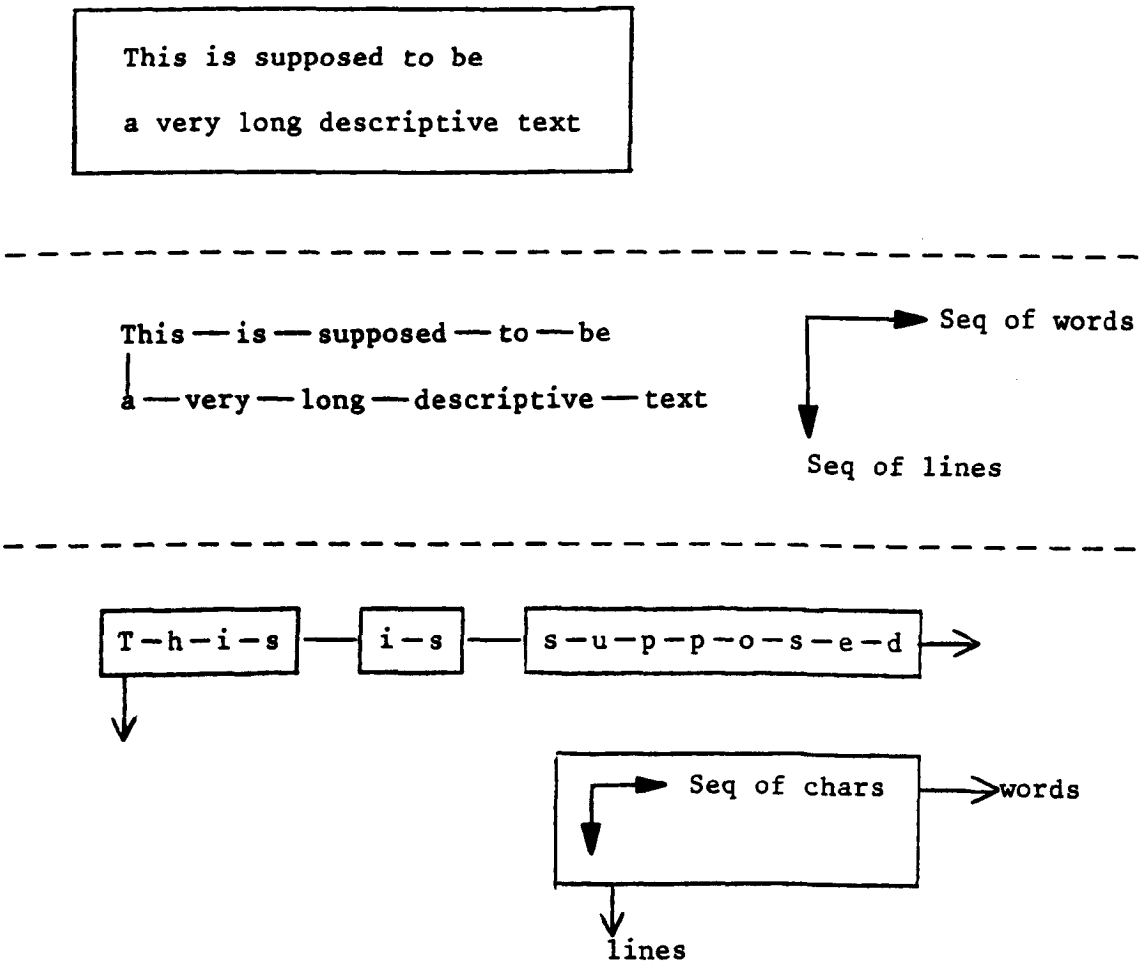


Figure 4-63. Symbolic Description.

4.9. Generalisation

From the binary digit, up through the predicate calculus, to today's programming languages, the symbolic description, restricted to *textual* forms, dominates the computing world. Little is seen or heard of purely *perceptual* descriptive techniques such as scale models, pictures or sounds. Sometimes, usually for creative or explanatory purposes (ie human(e) use), these two descriptive techniques are combined in a mutually reinforcing way. Von Neumann's[59] use of the flowchart for design and exposition is one famous computing example of this symbiosis. Dimensional Design also attempts to enhance textual descriptions by perceptual devices. Whether or not this attempt has been successful is discussed in Chapter 7: Practical Experience but first it will be useful to state the assumptions and aspirations which motivated the attempt and then to follow on with a harder look at the manipulations and underlying axioms on which this particular symbiosis is based.

4.9.1. Assumptions & Hypotheses

Assumption-1: A Description is an organised collection of inter-related symbols.

Implication-1: In a Description containing N symbols if every symbol is related to every other there are N^2 relationships. This figure of N^2 will be much higher if pairs of symbols are inter-related in more than one way. For example, a 1000 line computer program contains approximately 10^5 symbols(characters) and could contain $O(10^{10})$ relationships!

Assumption-2: A Description can be represented as a graph in which the vertices represent individual symbols and the edges represent the relationships.

Definition-1: A Relationship Graph is the graph of a Description.

Implication-2: A Description can always be represented as a 3-D embedding of its Relationship Graph in which no two edges cross. A Relationship Graph cannot, in all cases, be embedded in a 2-D plane.

Assumption-3: In general, the 3-D embedding of a Relationship Graph will be so cluttered, complex and obscured that it will be useless as a practical technique for representing a Description.

Hypothesis-1: It is possible to produce 'projections' (ie formal transformations) of the Relationship Graph so that certain aspects (relationships) are explicit (ie immediately perceivable) whilst other aspects are rendered implicit (ie must be deduced by human interpretation).

Hypothesis-2: It is easier to understand some particular aspect(s) of a system or object from a Description in which the relationships associated with the aspect(s) are explicitly (perceptually) represented rather than implicitly (textually) represented.

Hypothesis-3: A good way to 'project' a Relationship Graph is to systematically replace certain classes of edge by textually encoded equivalents so that it is still possible to deduce the, now, implicit relationships.

Hypothesis-4: It is practical, during 'projection', to completely delete certain classes of edge without textual replacement so that information is lost from the Description. This technique is most valuable when it removes irrelevant (to the task in hand) detail. Conversely, this technique is valid only if a restricted aspect of the Description is required which is roughly orthogonal to the

deleted aspect. The user of the Description may overcome such a deficiency by either learning about the missing aspect from another projection which does contain the requisite information, or by making assumptions about the missing aspect (hence the importance of conceptual integrity - see sections 2.3.3, 2.4) or by returning to a study of the complete Description armed with a clearer understanding of a particular aspect.

Example-1: A good example of the above hypothesis is the conventional flowchart which 'projects' control flow relationships by retaining the control relationships as explicit edges of a graph whilst holding most other information in textual form in the vertices (boxes). Some aspects are often completely missing from such projections eg timing, paging behaviour etc. Note in passing that such projections are graphs drawn in 2-D planes and usually contain cycles. Edges often cross.

Implication-3: In any projection of a Relationship Graph there are three classes of relationship:

1. those which are explicit (perceptually represented);
2. those which are implicit (textually represented);
3. those which are missing (but were in original Description).

Definition-2: For any connected graph G, a Spanning Tree of G is a tree formed by repeatedly removing an edge from a circuit until there are no circuits left. The graph so formed will still be connected.

Hypothesis-5: A very good way to project a Relationship Graph is to remove or replace edges so that the resultant graph is a Spanning Tree.

Implication-4: A Spanning Tree projection contains all the symbols present in the original Description. Any two symbols directly connected (ie by 1 edge) in the original Description are still connected in the Spanning Tree but not necessarily directly (ie by $\geq$ 1 edges).

Implication-5: A Spanning Tree projection can be drawn as an embedding in a plane ie a 2-D drawing with no lines/edges crossing. (from Kuratowski's theorem, T12B[79]).

Example-2: English text and programming languages are Spanning Trees in which almost all the relationships are implicit ie textually encoded. The only explicit, perceptually encoded relationships are the predecessor and successor relationships (by 'juxtaposition'). Such linear strings of symbols are the simplest form of Spanning Tree in which the user must perform a considerable amount of work to deduce all the relationships for himself. In an N symbol description contrast this one linear explicit relationship with the $O(N^2)$ possible relationships to see how much deductive work could be necessary.

Definition-3: A Relationship Tree is a Spanning Tree projection of a Relationship Graph.

Definition-4: An RT-diagram is a 2-D physical representation (drawing) of a Relationship Tree.

Central Hypothesis A:

Relationship Trees are an intellectually useful tool to describe systems and objects especially computer programs. They facilitate creative thinking, understanding and cognitive manipulation of such systems. (The Quality aspect of the small scale software engineering problem).

Central Hypothesis B:

Relationship Trees, when drawn as Dimensional Designs, are a practical means of describing systems and objects especially computer programs. They allow practical, simple and easy manipulation by both humans and machines (The Tractability aspect of the small scale software engineering problem).

To substantiate the two Central Hypotheses will involve demonstrating that Relationship Trees can indeed:

1. be successfully represented and manipulated physically ie drawn, stored etc (see Chapter 5: Manipulation);
2. facilitate human understanding and creativity (see Chapter 3: Abstraction and Chapter 7: Practical Experience);
3. be formally defined (see Chapter 6: Axiomatisation);
4. be manipulated according to well formulated rules (see Chapter 6: Axiomatisation).

4.10. Concluding Remarks on Representation

Of the four stages to developing an abstraction (Abstraction, Representation, Manipulation and Axiomatisation) Hoare[15] sees Representation as "the choice of a set of symbols to stand for the abstraction; this may be used as a means of communication" The programming abstractions of Set, Sequence, Hierarchy, Enumeration, Generation, H-Reduction, Integration and Induction

have been identified as relationships between symbols and Dimensional Design has been developed to allow perceptual reinforcement of these abstractions within a textual description. The technique has been generalised to enable any given relationships to be highlighted. The motivation behind this attempt to increase the visibility and hence the understandability of these key abstractions is to improve the communication of a program's design between the people working on it, especially between the designer and the maintainer/developer for whom the design document may well be the only channel of communication - and a one way channel at that.

It has been shown in this Chapter and [85] that Dimensional Designs are a practical communication technique. Do these abstractions and their representational techniques fulfil the requirement of the next phase in the development of an abstraction ie do they allow the manipulation of the representation to predict the effects of similar manipulations in the real world? This is the essence of what the designer wishes to be able to communicate to the machine, the maintainer and himself.

CHAPTER 5. DIMENSIONAL DESIGN: MANIPULATION

OUTLINE

The broad discussion of Software Engineering concluded by focusing on the small scale software engineering problem of quality and tractability which lead to the introduction of Dimensional Design as a way of representing eight programming abstractions. Following Hoare's strategy of abstraction, representation, manipulation and axiomatisation, Dimensional Design represented the programming abstractions as Relationship Trees which were perceptual enhancements of textual descriptions.

This chapter discusses how Dimensional Designs may be manipulated, both physically and logically. The discussion concentrates on the physical and practical process of actually drawing Dimensional Designs. Other manipulations are briefly considered - many having been discussed in other chapters.

Succeeding chapters discuss manipulations more formally (axiomatisation) and their use in actual practical software production. Finally, Dimensional Design is assessed relative to existing techniques.

5. DIMENSIONAL DESIGN: MANIPULATION

Hoare[15] sees Manipulation as "the rules for transformation of the symbolic representations as a means of predicting the effect of similar manipulation of the real world".

The manipulations which may be performed on Dimensional Designs fall into two categories. The first is the physical transformation category containing those manipulations which may be performed independently of the 'meaning' of the Dimensional Design in the 'real world'. Such transformations include the reduction in size of a description by Hierarchical Reduction, Integration and Induction as well as the practical manipulations of drawing, formatting, copying, storing etc.

The second category of manipulation contains the logical transformations which require of the manipulator a knowledge of the 'meaning' of the description and its 'real world' counterpart. For example, the program fragment in figure 5-1a can only be validly manipulated (simplified) to become figure 5-1b if the symbols represent a conventional sequential computer program.

In the 'real world' of computer programs the manipulator is interested in two things, the generation of instruction sequences and the execution of instruction sequences. The power of the von Neumann machine lies in its superb integration of these two manipulations. Dimensional Design can represent instruction sequences and Generation, Selection and Induction, the manipulations which create instruction sequences. As was indicated in section 3.5.1.3, Generation, Selection and Induction are directly built into the GVNC and so manipulation of these descriptive techniques reflects directly the 'real world' version of creating instruction sequences. It is worth noting that restricting the design process to allow only these manipulations automatically ensures that the programs so produced are 'Structured Programs' and thus

enjoy the benefits described in section 3.2.

The execution of instruction sequences can also be represented by Dimensional Designs (see computational histories and animation in Chapter 7). It is interesting to note that execution can be viewed as a technique which can be used to further reduce the size of a description (figure 5-2). This is akin to Reduction Semantics,[4] a thought not pursued further.

Most of the rules for manipulating the representations of computer programs such that the transformations of the symbolic representations (source code) predict the effects of similar manipulations of the real world (binary programs) are discussed in Chapter 7: Practical Experience when Dimensional Design, allied to modern programming's logical manipulations, is put to the practical test.

Thus, to summarise, Dimensional Design manipulations can be categorised into:

Physical Manipulations

1. drawing, formatting, copying, storing etc;
2. reducing size of description (meaning independent).

Logical Manipulations

3. simulating & predicting the real world;
4. reducing size of description (meaning dependent).

The first of these, the physical process of actually drawing a Dimensional Design will now be discussed.

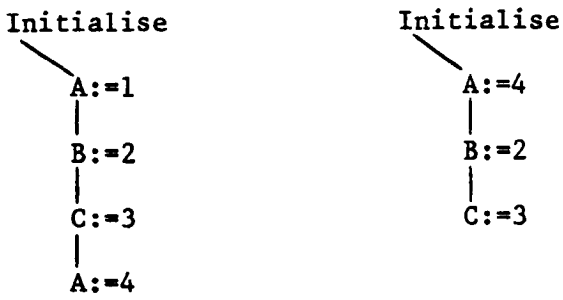
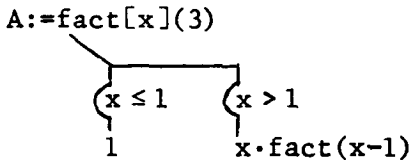
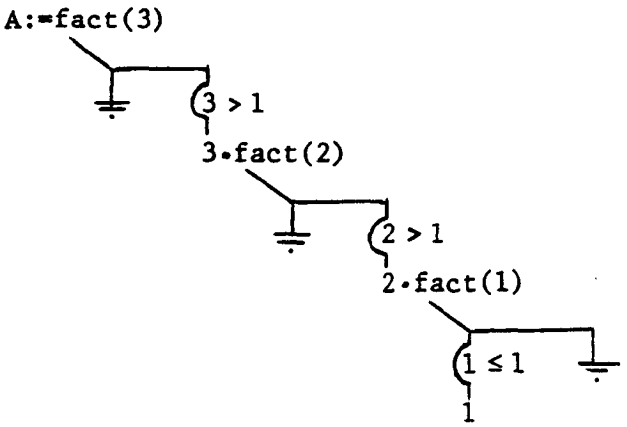


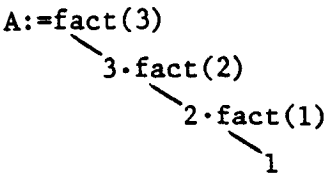
Figure 5-1. Simplification is a logical transformation.



(a) Description with Actual Parameter



(b) Description Parameter Substituted and Inductive Generation Halted by Selection



(c) Pruning

A:=3·2·1

(d) Removal of Artificial Hierarchy

A:=6

(e) Evaluation

Figure 5-2. Reduction by 'execution'.

5.1. Drawing

If the Central Hypotheses (section 4.9.1) are to be substantiated then one of the first questions to be answered must be 'Can the abstract concept of a tree of inter-related symbols be practically represented on paper and drawn easily by both man and machine?'

In section 3.1:Informal Introduction, the simple Dimensional Designs introduced were 3-ary Relationship Trees. This admitted the use of the elegant layout technique of assigning each relationship type to one of three orthogonal spatial axes, allowing instances of the same type of relationship to be represented unambiguously by edges lying parallel to their associated axis. The 2-D projection of this 3-D layout technique was used extensively throughout the Informal Introduction. It depended on the unambiguous angular differentiation of edges lying in the same plane (0°=Set, 45°=Refinement, 90°=Sequence).

Unfortunately the 3-D representation is not directly capable of practical generalisation to an n-D space representing a Relationship Tree with n types of distinct relationships. However an alternative approach is possible. Implication 5 indicates that a Relationship Tree can indeed be drawn on paper (a 2-D plane) in a readable fashion in which no two edges cross and edges are all simply straight lines. In general, therefore, an n-ary Relationship Tree can be drawn as figure 5-3 which is an RT-diagram.

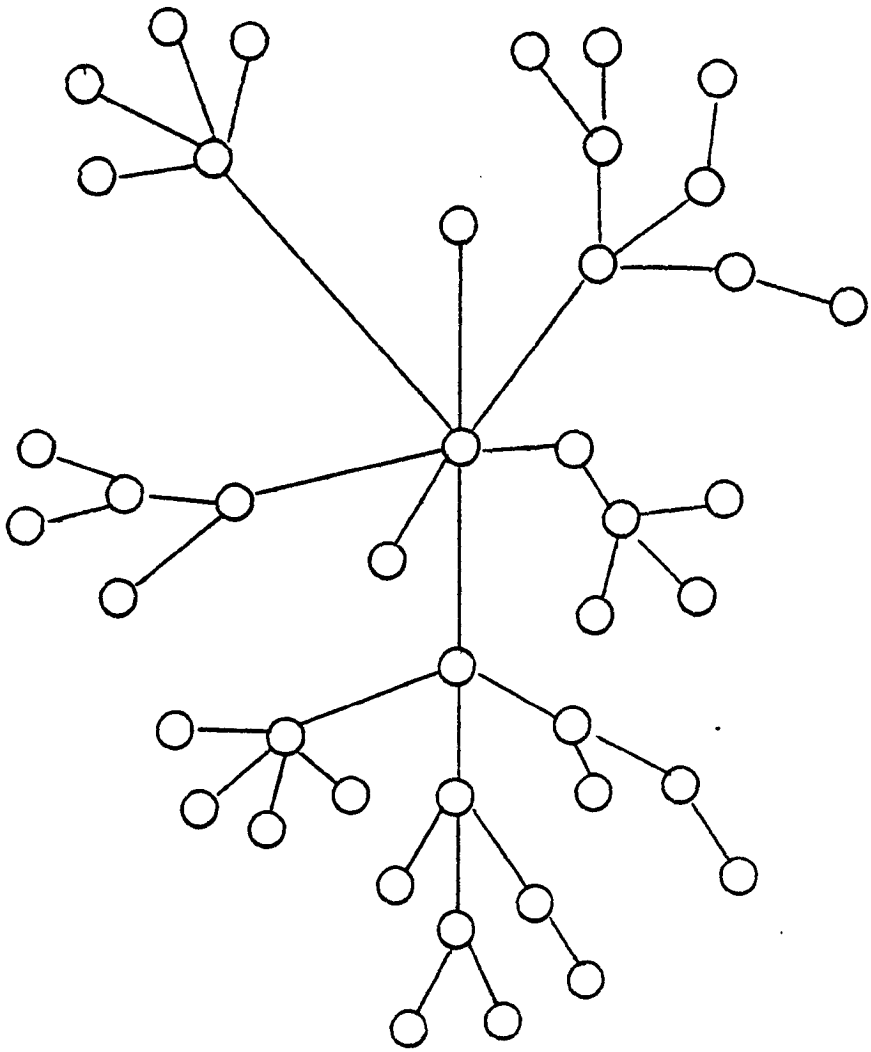


Figure 5-3. N-ary Relationship Tree-diagram.

5.2. Four Quadrant Diagrams

Consider figure 5-4 which is a Relationship Tree-diagram with 12 distinct relationship types laid out radially so that every edge lies in one of four quadrants with its origin at the root of the tree, the symbol 'A'. As any finite subtree can be enclosed in a rectangle whose sides are parallel to the quadrant axes, let the subtrees B-Q be so enclosed thereby representing non-elementary subtrees of A. Figure 5-5 illustrates two possible algorithms for drawing an RT-diagram of figure 5-4. Figure 5-5a shows the 'constant angle' technique in which all edges representing the same relationship type are drawn parallel to each other at a constant angle to the axes. Although this algorithm has the advantage that 'fraternal' subtrees may be drawn independently of each other because they lie in independent sectors it has the major disadvantage that any single vertex and the edge connecting it to a 'filial' subtree cannot be drawn until *after* the size of the rectangle enclosing the filial subtree has been calculated so that the subtree's position within the sector can be determined. This amounts to a *postorder* tree drawing algorithm in which subtrees are drawn before roots. Such an algorithm is expensive to automate and is impossibly complicated for manual use.

It should also be noted that this 'radial' technique of tree drawing requires that the diagram 'grow' in all four quadrants and that, given a finite paper size, the positions of the roots of trees (and subtrees) vary with respect to the boundaries of the paper.

One great advantage of the constant angle technique is that it is a simple matter to identify the angular orientation of an edge with a specific relationship type. This is a simple and effective technique compared to those of, say, explicit labelling, colour coding or varying line styles. However all of these techniques are only really effective in representing a small number of relation-

ship types, for if 360 types were to be handled one could not distinguish 1° differences in angular orientation. The success of the 3-D spatial layout projected onto a 2-D plane is a special case of the successful use of the constant angle technique to represent a small 'vocabulary' of relationships perceptually (see Chapter 8:Assessment).

The constant angle technique can be wasteful of paper in that large areas of 'white space' are required in the diagram. A saving can be achieved by using the 'variable angle' technique, see figure 5-5b. This technique draws 'fraternal' subtrees sequentially so that the i^{th} one to be drawn is aware of the amount of space occupied by its (i-1) brothers. This leads to more efficient use of the paper but only at the expense of losing the elegant, constant angle, edge marking perceptual encoding.

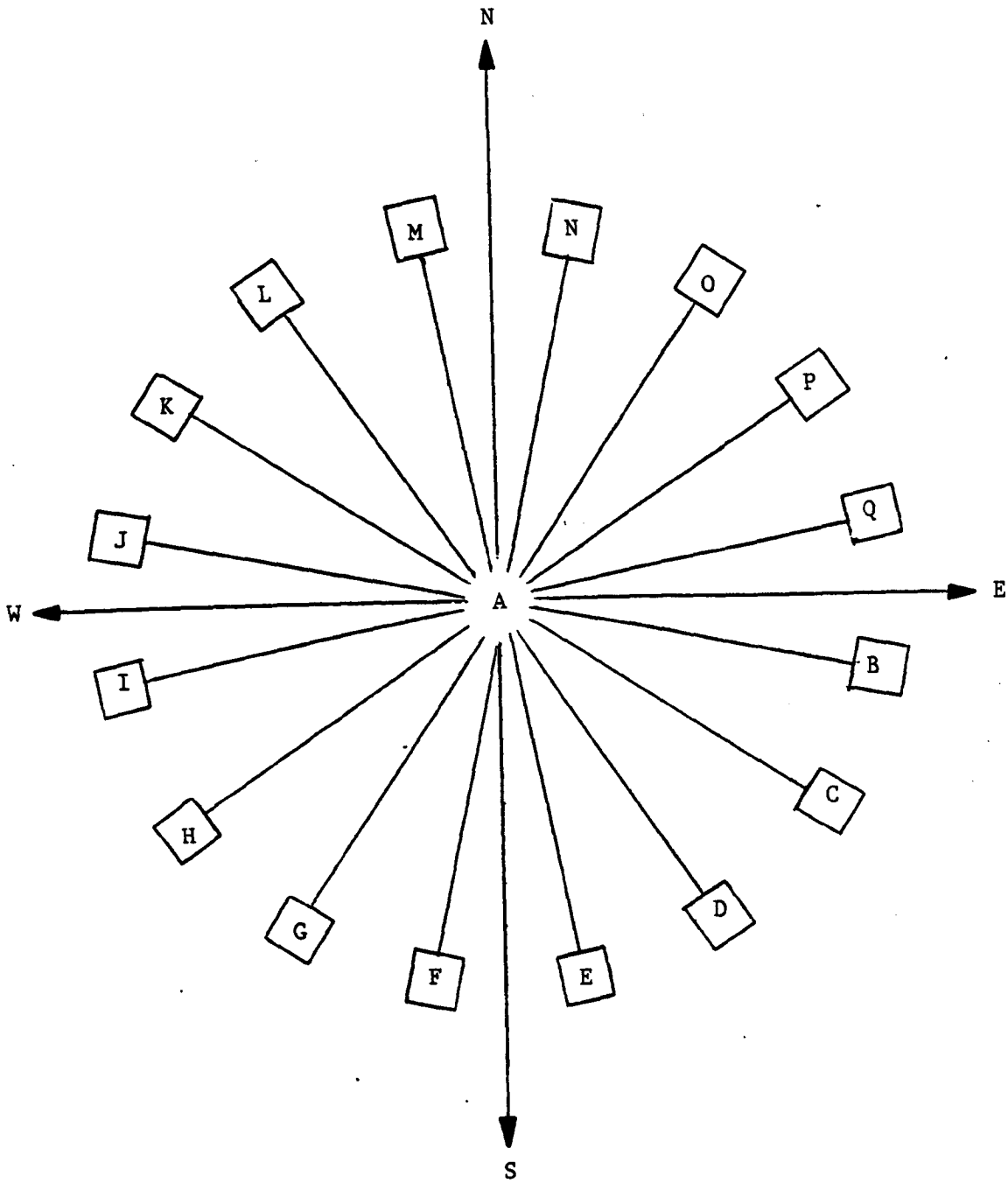
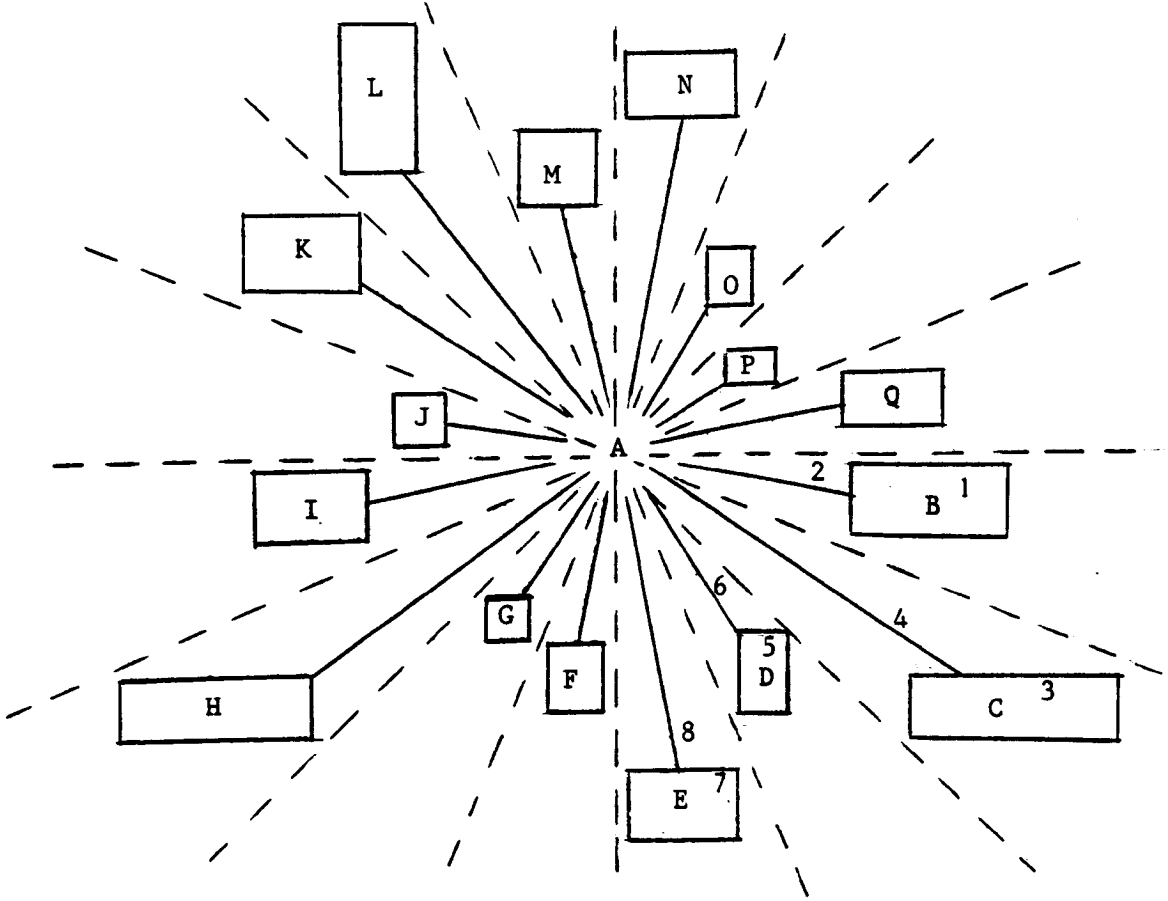
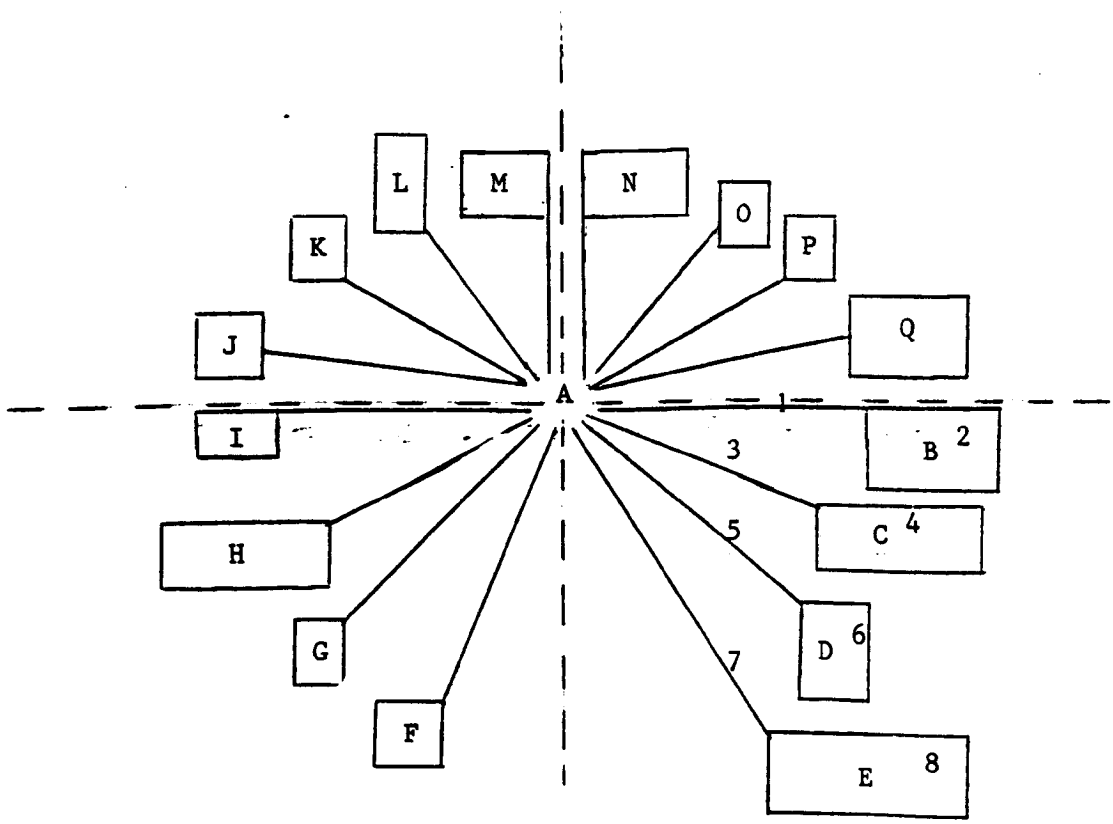


Figure 5-4. Four Quadrant RT-diagram.



(a) Constant Angle RT-diagram



(b) Variable Angle RT-diagram

Figure 5-5. Drawing Algorithms - integers show order in which components actually drawn.

5.3. One Quadrant Diagrams

There are significant practical advantages to be gained from drawing RT-diagrams so that they only 'grow' within one quadrant. Figure 5-3 can be redrawn to fit into one quadrant, the south-east in figure 5-6, by closing up figure 5-3, rather like a lady's fan, around the root symbol. This can be done for all subtrees, too.

The advantages of one quadrant RT-diagrams are that the root of any subtree can be in a fixed position relative to any enclosing rectangle (real or imaginary) and that the diagram is bounded on two sides, bounds which are determined by the position of the root symbol and known in advance of drawing all edges and subtrees. The one quadrant diagram allows the variable angle drawing technique to become a *preorder* drawing algorithm. RT-diagrams can be drawn root first with each subordinate subtree drawn later. This procedure (outlined in figure 5-7) is cheap and simple to automate and very simple to use manually.[83, 85]

However, the disadvantage of one quadrant diagrams is that they accentuate the problem of representing many different relationship types in one diagram because the restriction to one quadrant reduces the 'angular vocabulary' which may be used by the constant angle technique. The variable angle technique is no better off as compressing the RT-diagram makes edge marking more difficult.

However a compromise solution is possible given a little sympathy on the part of the user. Consider, in figure 5-4, the subtree A, B, C, D, E. Let this subtree, the contents of the south-east quadrant, be enclosed in a rectangle (figure 5-8a). Let the edges leading from the root symbol 'A' to the subtrees in the south-west quadrant be extended so that each subtree in the south-west quadrant lies wholly outside a rectangular area which is defined by the area

formed by rotating the south-east enclosing rectangle about the north-south axis (figure 5-8b). Let the whole south-west quadrant be rotated about the north-south axis so that it is 'folded' under the south-east quadrant then the resultant diagram (figure 5-8c) can be thought of as being a 2-D projection of the folded quadrants (see figure 5-8e). If all the other quadrants are rotated into the south-west quadrant and then folded into the south-east (figure 5-8d) a complete, unambiguous and clear representation of the original tree is obtained, for if any edge is encountered which terminates at the boundary of an enclosing rectangle then the symbol to which it was originally connected may be simply identified as being that symbol which lies at the intersection of all the edges which terminate at that same rectangular boundary (see figure 5-8e).

Using this technique a practical diagram of a Relationship Tree may be constructed which has the following properties:

1. Any number of instances and types of relationships can be represented.
2. The position on the page (enclosing rectangle) of the root symbol of an RT-diagram (subtree) is always 'top-left' for south-east version.
3. Edges and subtrees may be drawn without knowledge of the sizes of subsequently drawn subtrees (preorder drawing).
4. Edges are easier to mark as the enclosing rectangles can be used as scope delimiters to allow marking conventions to be re-used in different contexts.
5. The complete diagram is contained in one (south-east) quadrant. This means that the diagram has two fixed boundaries (south and east axes) and expands only rightwards (east) and downwards (south) in the same way as written text which is a very natural way for a human draughtsman to proceed.

Thus this folding or overlaying of quadrants technique allows an uncluttered representation of an n-ary tree where n is arbitrarily large. The final problems are the choice of layout algorithm and edge marking technique.

The constant angle technique is the simplest way to mark edges. The number of meanings of the angular orientations can be 'overloaded' by using the enclosing rectangles to define the scopes of different sets of relationship types. However, the constant angle layout technique is not the best. Fortunately, in the special case where a maximum of three relationship types is used per enclosing rectangle the angles of the edges never vary when drawn by a variant of the variable angle algorithm (see figure 5-9 and [85]). The restriction to three different relationships per rectangle does not limit the number of different relationships in total which may be portrayed as any number of 'overlays' may be employed.

Restricting the number of distinct relationships to three per enclosing rectangle has a second advantage. It allows the reintroduction of the perceptual benefits of the 3-D spatial layout projected onto 2 dimensions idea. Figure 5-10 illustrates how the planar representation of 'folded' quadrants can be augmented by adding perceptual clues to heighten the RT-diagram into the full 2-D from 3-D cuboid technique introduced in section 4.1.Informal Introduction.

The full Dimensional Design representation is the cuboid version of RT-diagrams in which three further conventions are observed:

1. Each outgoing edge from a symbol must represent a unique relationship type in the set of outgoing edges for that symbol.
2. Dimensional Designs which are 'sparse' in that some enclosing cuboids contain no subtrees have these cuboids removed and the axes of individual cuboids are explicitly labelled.

3. Cuboids around and edges within subsystem textual descriptions are omitted if the Dimensional Design remains unambiguous to the reader.

Convention 1 implies that a named set of symbols is represented by a 'right-threading' technique (figure 5-11b) rather than a 'fan' (figure 5-11a). This cuts down a lot of clutter when representing large named sets or sequences.

Convention 2 is illustrated by figure 5-12. This again helps to cut out redundant clutter and eases the problem of edge marking.

Convention 3 saves cluttering up the RT-diagram with a lot of, from experience, unnecessarily pedantic cuboids (figure 5-13). This convention may only operate when 2 and only 2 relations (next character and next line) are used within a textual subsystem. By encoding these two relations as horizontal and vertical juxtaposition ie zero length edges, they can be easily differentiated from the Set and Sequence relations which are always expressed by non-zero length edges. Thus Convention 3 actually extends the edge vocabulary from three to five (see Chapter 8: Assessment); even so, the cuboids around the lines of text are still strictly necessary in order to avoid ambiguity.

If the above drawing techniques and conventions are employed, the Dimensional Designs may be physically manipulated, drawn, formatted, etc. with ease both manually and mechanically. Of the four types of manipulations introduced at the start of this chapter ie

Physical Manipulations

1. drawing

2. reducing size of description (meaning independent)

Logical Manipulations

3. predicting the real world

4. reducing size of descriptions (meaning dependent)

the above discussion has covered type 1, drawing. Type 4 was discussed in section 5.1 of this chapter. Type 3, the prediction of the real world, is discussed in Chapters 3, 4 and 7 and is covered extensively in the literature as it encapsulates the essence of programming. The type 2, meaning independent, description reducing manipulations were discussed informally in Chapters 3 and 4. Now they will be discussed rather more formally because, according to Hoare's strategy, once having intuitively recognised the relevant abstractions, found a convenient representation for them, experimentally manipulated them, then it is time to put them on a more formal basis for only when and if this final stage of axiomatisation is complete can the full power of the abstractions be employed with confidence.

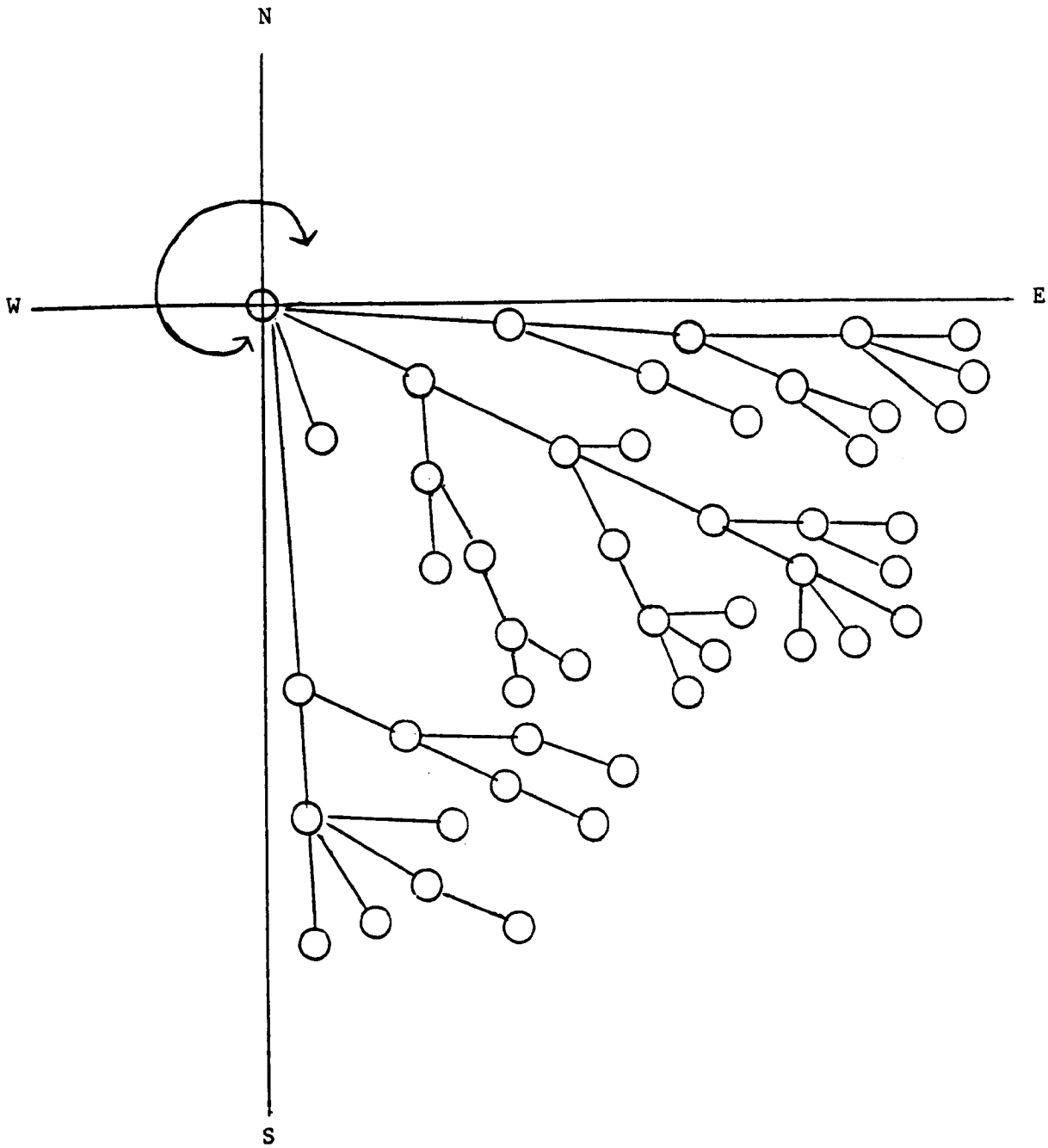


Figure 5-6. Closing up into one quadrant.

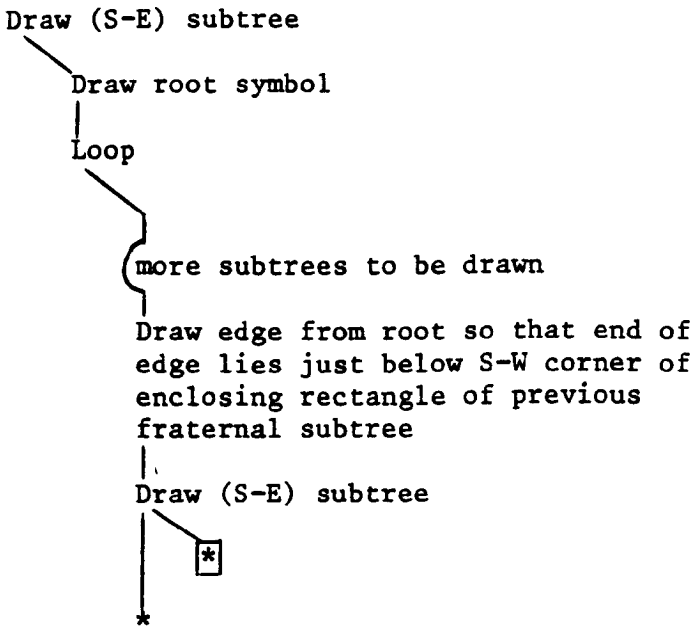
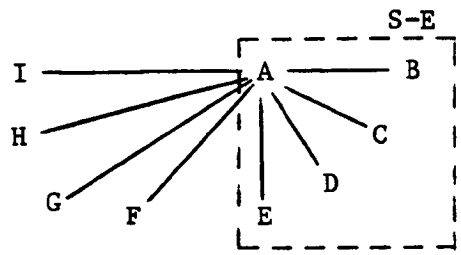
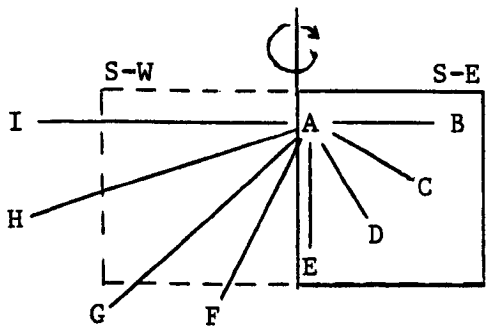


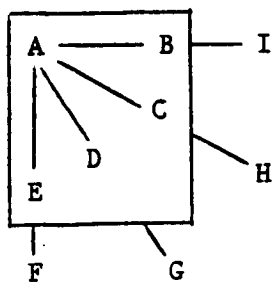
Figure 5-7. Preorder, One Quadrant, Variable Angle Algorithm.



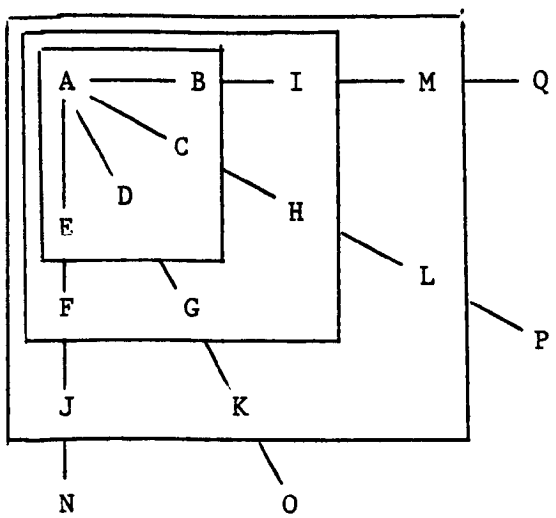
(a) Enclose S-E



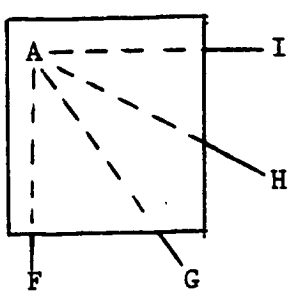
(b) Extend S-W edges



(c) Fold S-W under S-E

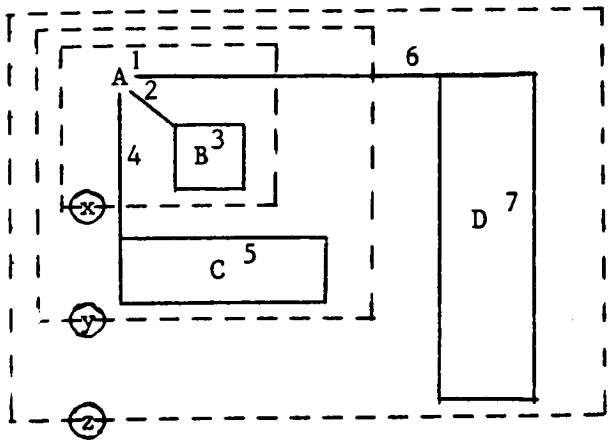


(d) Rotate next quadrant & repeat

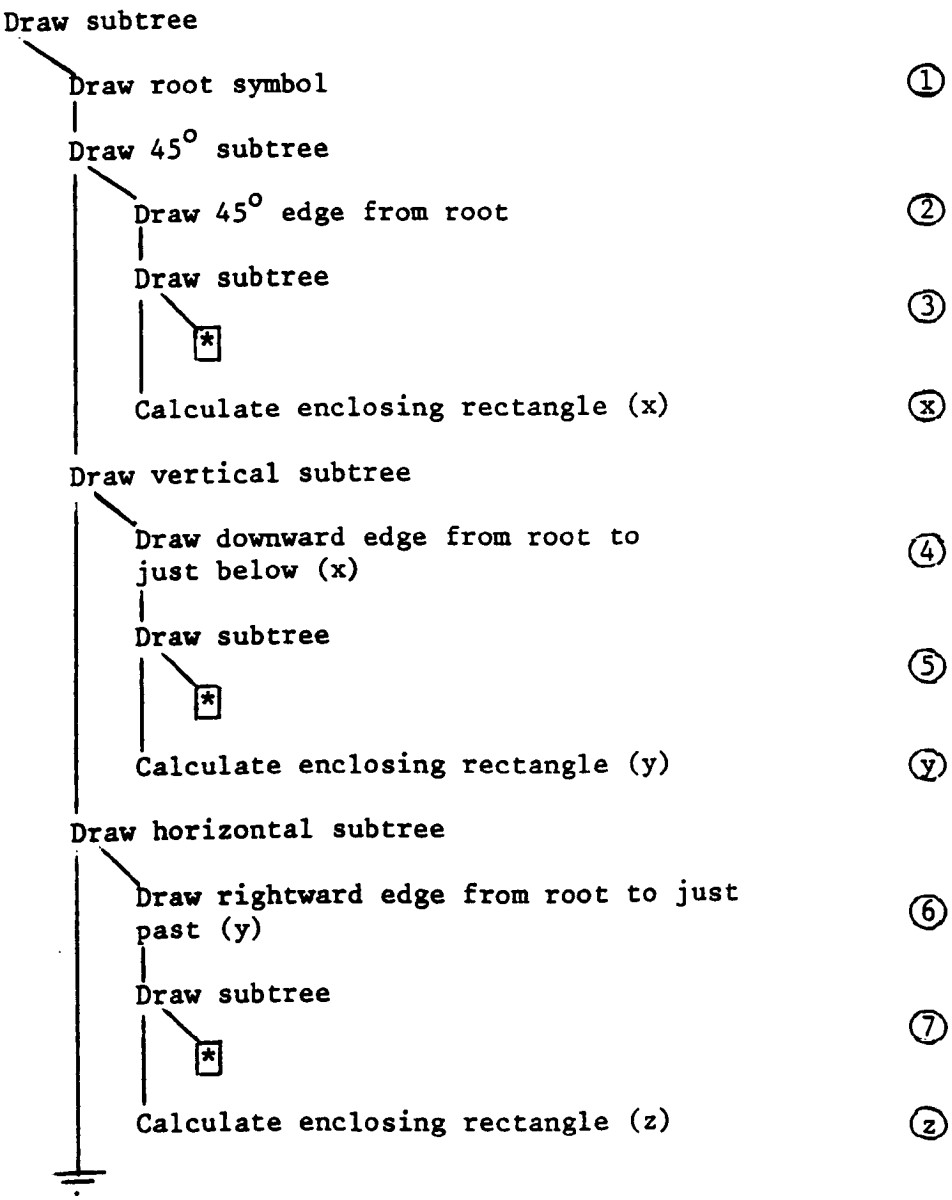


(e) What lies beneath S-E quadrant

Figure 5-8. Quadrant Folding.

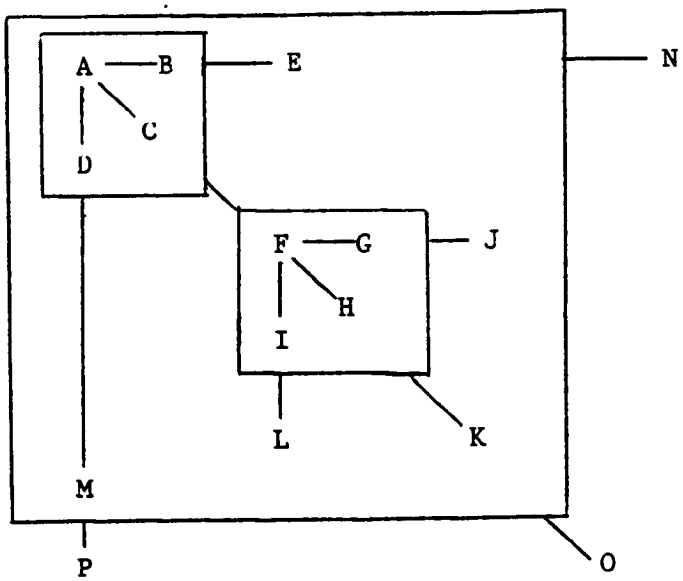


(a)

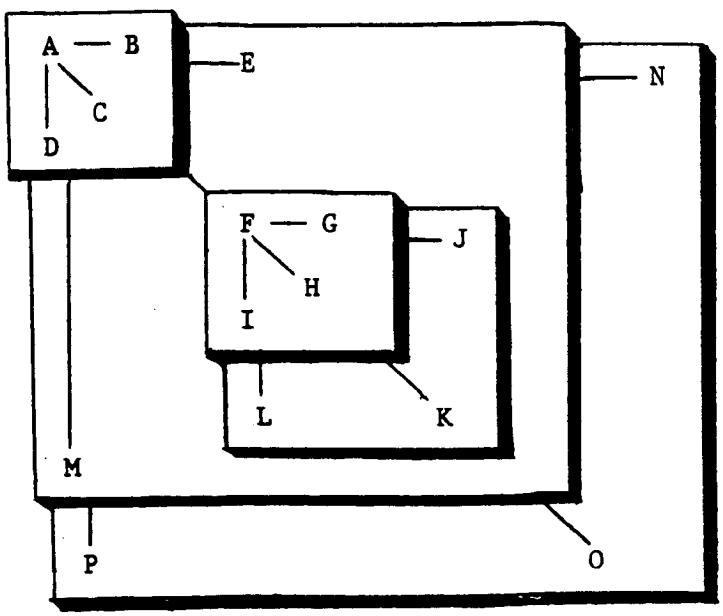


(b)

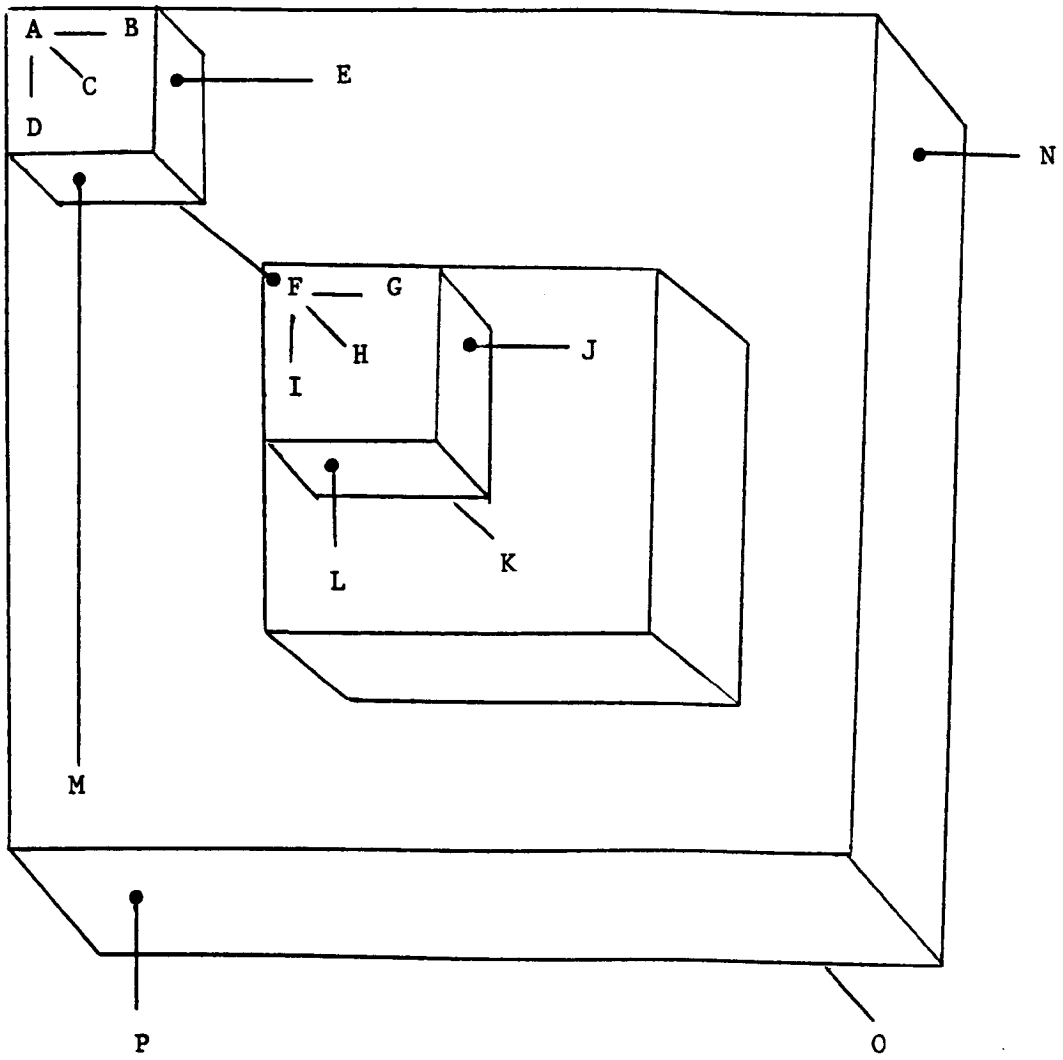
Figure 5-9. Basic Dimensional Design Layout Algorithm.



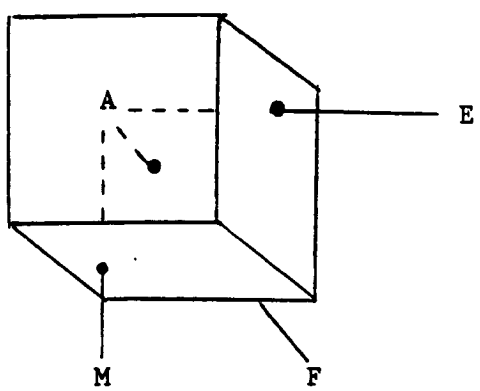
(a) Planar



(b) 'Thick Paper'

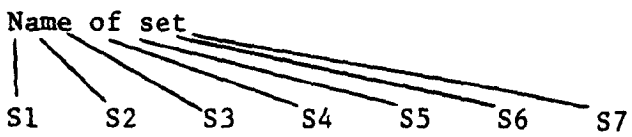


(c) 3-D Cuboid RT-diagram

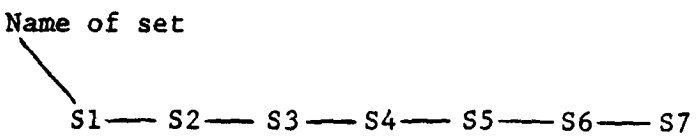


(d) Root-subtree connections

Figure 5-10. Building up to the Cuboid Model.

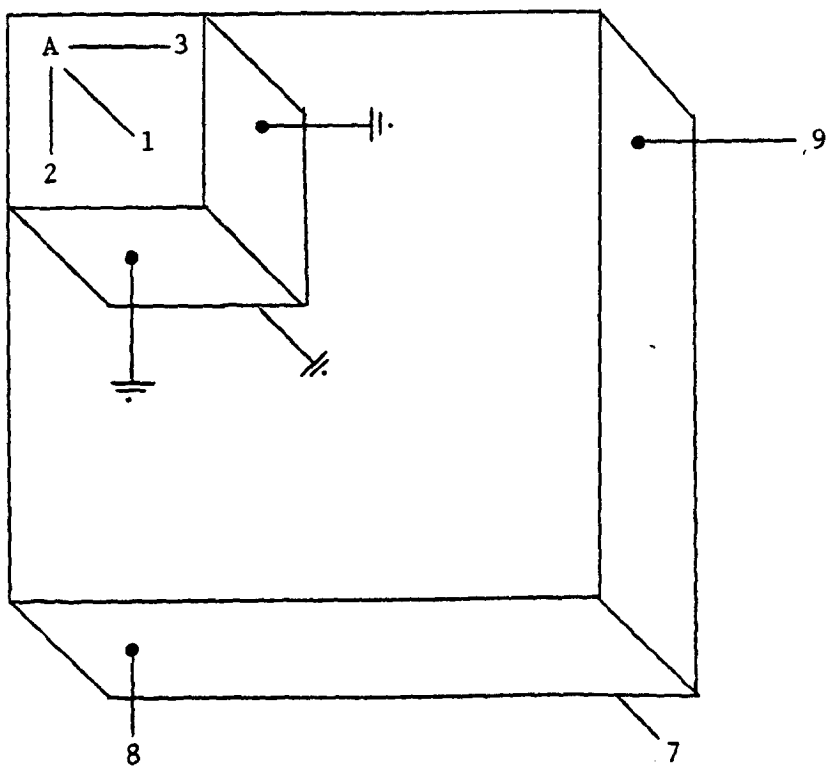


(a) Radial 'fan' of named set

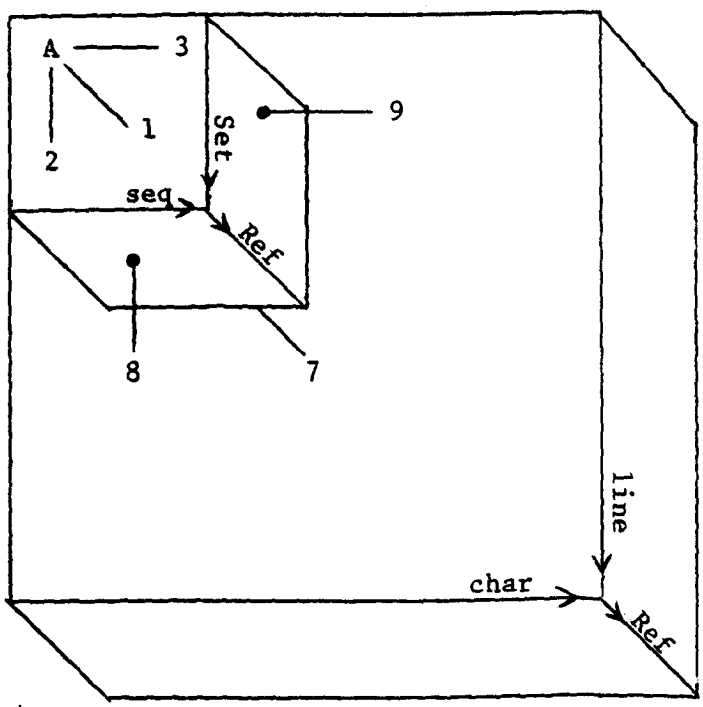


(b) 'Right-threading' of named set

Figure 5-11. Named Set.

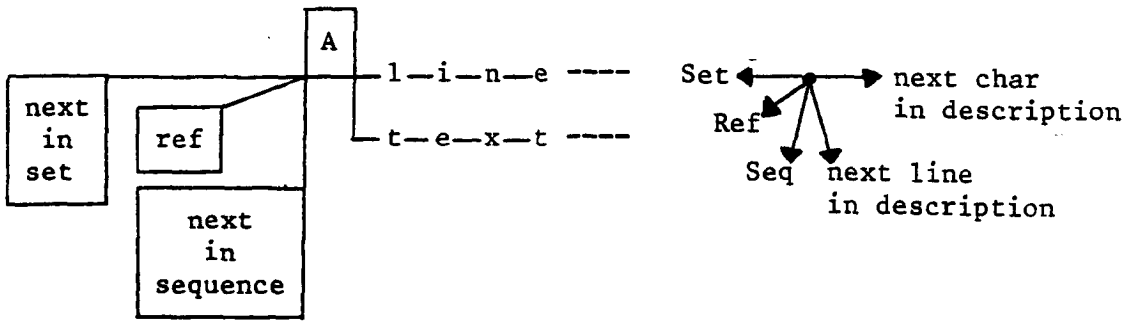


(a) 9-RT in full

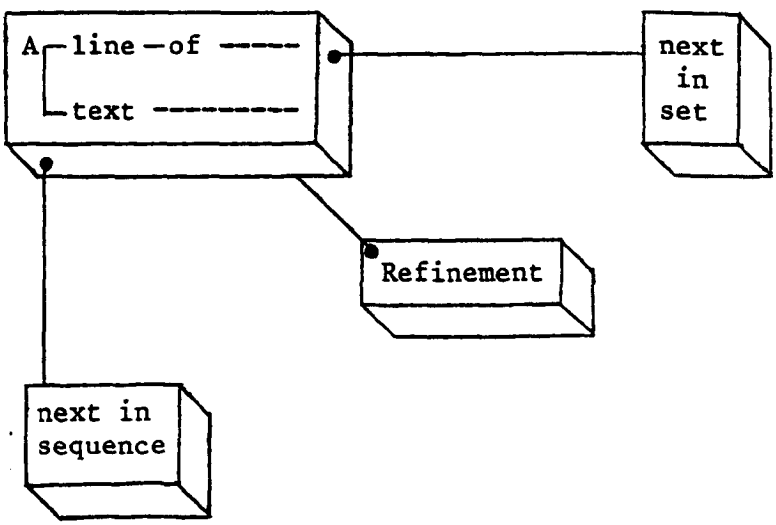


(b) Simplified to 6-RT

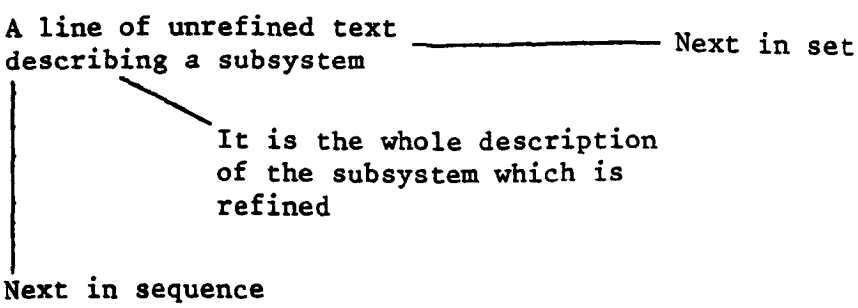
Figure 5-12. Sparse Subtrees.



(a) Radial Layout (5-RT)



(b) Folded into One Quadrant & put into Cuboids (5-RT)



(c) Conventional Dimensional Design (3-RT) with the 2-RT textual subsystems removed

Figure 5-13.

CHAPTER 6. DIMENSIONAL DESIGN: AXIOMATISATION

OUTLINE

In the wake of the Software Crisis and Software Engineering eight programming abstractions were abstracted, represented and manipulated to show they form an intuitively adequate basis for Dimensional Design to represent programs. In this chapter the final stage of abstraction, axiomatisation, is undertaken to specify the underlying rules which govern the construction and manipulation of Dimensional Design. Based on a set of definitions, axioms are given which specify the principles of Dimensional Design. Axioms are given to reduce the size of a description independent of its meaning, encapsulating the techniques of Hierarchical Reduction, Integration, Induction, Enumeration and Generation. Some theorems are illustrated whose proofs show the power of these abstractions in quantitative terms.

This chapter does not attempt to produce a full, consistent Theory of Dimensional Design, but, perhaps, goes far enough to enable the succeeding chapter on the practical application of Dimensional Design to be more easily appreciated as it attacks the small scale software engineering problem of improving the quality and tractability of software design products. The penultimate chapter compares Dimensional Design against existing techniques.

6. DIMENSIONAL DESIGN: AXIOMATISATION

Throughout Chapters 3-5 Dimensional Design has been used to represent and manipulate many facets of programs, but the underlying rules governing Dimensional Design have never been specified. This process of rigorous specification, axiomatisation, is the final step, according to Hoare's strategy,[15] needed to complete the development of a new abstraction or technique. If successful, axiomatisation crystalises intuition into solid, consistent, well understood theory. If unsuccessful, axiomatisation pinpoints the weaknesses and gaps where intuition has made unjustifiable assumptions and omissions. The following discussion specifies the previously informally introduced principles of Dimensional Design and shows the way to a Theory of Dimensional Design.

6.1. Definitions

Given-1: SYMSET - a set of symbols

Given-2: RELSET - a set of relationships, instances of which may hold between pairs of symbols from SYMSET.

Definition-1: A Relationship Graph is the graph of a Description (as defined in Assumption-1, section 4.9.1).

Definition-2: For any connected graph G, the Spanning Tree of G is the tree formed by repeatedly removing an edge from a circuit until there are no circuits left. The graph so formed will still be connected.

Definition-3: A Relationship Tree is a Spanning Tree projection of a Relationship Graph.

Definition-4: An RT-diagram is a 2-D physical representation (drawing) of a Relationship Tree.

Definition-5: n = cardinality of RELSET.

Definition-6: An n-Relationship Tree (n-RT) is a directed n-ary tree in which every vertex is an instance of a symbol in SYMSET and every directed edge is an instance of a relationship in RELSET ie every edge is 'marked' to show which relationship it represents.

Definition-7: A Planar RT-diagram is an n-RT drawn in a 2-D plane by the One Quadrant, Enclosing Rectangle drawing algorithm.

Definition-8: A 3-Dimensional Design is an n-RT drawn in a 2-D plane by the One Quadrant, Cuboid drawing algorithm.

A natural way to express definitions 6, 7, and 8 is by way of context free grammars which generate n-RTs, planar RT-diagrams and Dimensional Designs as preorder tree walks eg:

symbol = ('a' | 'b' | ..... | .....) ie SYMSET

edge = (rel1 | rel2 | .... | .....) ie RELSET

= <null subtree>

λ = <nothing>

n = <cardinality of RELSET>

m = <number of relationships per enclosing rectangle>

[] = <enclosing rectangle>

$$nRT = symbol \{(edge\ nRT \mid \lambda)\ \#\}^{n}$$

$$PnRTd = symbol \mid [\ PnRTd \{(edge\ PnRTd \mid \lambda)\}^{m}\ ]$$

DD = (symbol | [DD]) (diagedge DD | λ) #

(vertedge DD | λ) #

(horzedge DD | λ) #

Definition-9: All relationship types in RELSET may be classified as being either Enumerative or Generative. An Enumerative Relationship between any two symbols 'a' and 'b' implies nothing about any other occurrences of these symbols either separately or as a pair. Enumerative relationships are indicated by the suffix 'e' eg 'a rel_e b'.

Definition-10: A Generative Relationship between two symbols 'a' and 'b' such that 'a rel_g b' implies that all instances of the symbol 'a' are related by 'rel_g' to the subtree of which 'b' is the root. Generative relationships are indicated by the suffix 'g'.

Definition-11: An Enumerated Description is an n-RT in which all the members of RELSET are Enumerative.

Definition-12: A Generative Description is an n-RT in which at least one member of RELSET is Generative.

Definition-13: All relationship types in RELSET may also be classified as being either Real or Artificial. A Real Relationship between any two symbols 'a' and 'b' implies that the relationship holds in the 'real world' between the 'real things' that 'a' and 'b' represent. Real Relationships are indicated by the 'r' suffix.

Definition-14: An Artificial Relationship implies that the relationship between any two symbols is not present in the 'real world' but has been added to the description for some purpose such as H-Reduction or exposition etc. Artificial relationships are indicated by the 'a' suffix.

Definition-15: In an n-RT, if there exists a symbol in SYMSET (say '•') and a relationship in RELSET (say 'rel') then '•' is the null symbol if, for any symbol 'a'

$$a\ rel\ \bullet = \bullet\ rel\ a = a$$

6.2. Description Reduction (Meaning Independent) Axioms

The following axioms encapsulate the mechanisms used to reduce the size of descriptions. They do not rely on any property of the relationships in the description, only on whether the relation is enumerative or generative, ie type

2, see Chapter 5. All the following conventions and axioms are applicable to all three axes and so only one axis will be used to define each axiom.

Let DDSYM = { $\overline{\div}$, *, □, ▣, C, •, true, false } where

$\overline{\div}$: end of subtree symbol

* : iteration symbol

□ : any finite, non-recursive subtree which has been removed by H-reduction

▣ : any infinite, recursive subtree which has been removed by H-reduction

C : conditional symbol (see integration axioms)

• : null symbol

true : Boolean for use in Selection

false : Boolean for use in Selection

Let $SYMSET = DDSYM \cup \{<description\ symbols>\}$

Let $RELSET = \{rel1, rel2, \cdots, reln\}$

Let $rel_{(e|g)}$ be represented, in the axioms, as $\frac{}{e}$ or $\frac{}{g}$ to simulate cuboid edge marking.

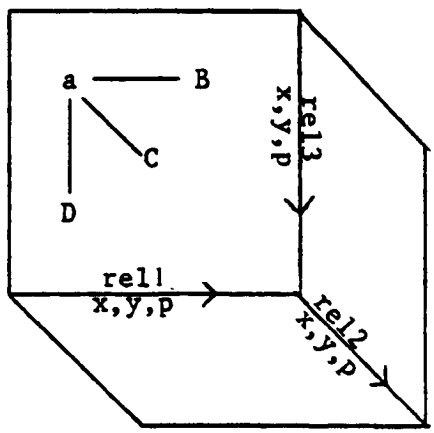
Let a, b, c be any elementary symbols from {<description symbols>} or cuboids ie complete, enclosed Dimensional Designs.

Let A, B, C be any subtrees, not necessarily cuboids.

Axiom 1 : Special Terminal Symbols

These symbols may only appear as terminal nodes in a Dimensional Design.

Axiom 2 : Cuboid Edge Marking



For all edges lying in the volume defined by

$$volume\,(cuboid) - \sum volume\,(inner\ cuboids\ of\ B,C,D)$$

then

$- \Rightarrow rel\,1_{x,y,p}$

$\backslash \Rightarrow rel\,2_{x,y,p}$

$| \Rightarrow rel\,3_{x,y,p}$

where

$x = a|r$ artificial or real relation

$y = e|g$ enumerative or generative relation

$p = \langle integer \rangle$ precedence (see description dependent axioms)

The following axioms may be used to transform Enumerated (Generative) Dimensional Designs into Generative (Enumerated) Dimensional Designs as, in combination, they allow the following techniques to be used:

Generative Description		Enumerative Description
Artificial H-Addition	←→	*Artificial H-Removal*
H-Reduction	←→	*H-Expansion*
Integration	←→	*Selection*
Induction	←→	*Deduction*

6.2.1. Artificial Hierarchy Addition & Removal

Axiom 3 : Null Symbol Addition & Removal

$$\bullet\!\!-\!\!-a = a-\!\!-\!\!\bullet = a$$

Axiom 4 : Artificial H-Addition

If $\exists B$ *with* $RELSET_B$

then B becomes $a \xrightarrow[a,e]{rel} B$ and

$RELSET_B$ becomes $rel_{a,e} \cup RELSET_B$

Axiom 5 : Artificial H-Removal

If $\exists C$ such that $RELSET_C = RELSET_B \cup rel_{a,e}$

then in C: $\forall a$ such that $a \xrightarrow[a,e]{rel} B$.

$a \xrightarrow[a,e]{rel} B$ becomes B and

$RELSET_C$ becomes $RELSET_B$

6.2.2. Hierarchical-Reduction & Expansion

Axiom 6 : Generative Relation

$$a \underset{g}{—} B \Rightarrow \forall a,\ a \Rightarrow a \underset{g}{—} B$$

Axiom 7 : Generation

$$\text{If } \forall a,\ a \Rightarrow a \underset{e}{—} B$$

$$\text{then } \forall a,\ a \underset{e}{—} B \text{ becomes } a \underset{g}{—} B$$

Axiom 8 : H-Reduction

$$\text{If } \forall a,\ a \underset{g}{—} B \text{ and } \exists \text{ n such instances}$$

then for (n-1) instances $a \underset{g}{—} B$ becomes a

and 1 instance remains unchanged

Axiom 9 : H-Expansion

$$\text{If } \exists\ a \underset{g}{—} B$$

$$\text{then } \forall a, \text{ a becomes } a \underset{g}{—} B$$

Axiom 10 : Enumeration

$$\text{If } \forall\alpha,\ a \underset{g}{\text{—}} B$$

$$\text{then } \forall\alpha,\ a \underset{g}{\text{—}} B \text{ becomes } a \underset{e}{\text{—}} B$$

Axiom 11 : Enumerative Relation

$$a \underset{e}{\text{—}} B \Rightarrow \forall\alpha,\ (a \Rightarrow a \underset{e}{\text{—}} B) = \text{false}$$

Axiom 12 : Redundant Recoding of Enumeration

$$a \underset{e}{\text{—}} b \underset{e}{\text{—}}| = a \underset{e}{\text{—}} b$$

Note: in conjunction with Axiom 1, this implies

$$a —| —b = a —| \ b = a$$

(disconnection)

Axiom 13 : Redundant Recoding of non-recursive H-Reduction

$$\text{If } \forall\alpha,\ a \Rightarrow a \underset{g}{\text{—}} B \text{ and } B \text{ is a finite subtree}$$

$$\text{then } a = a—\square$$

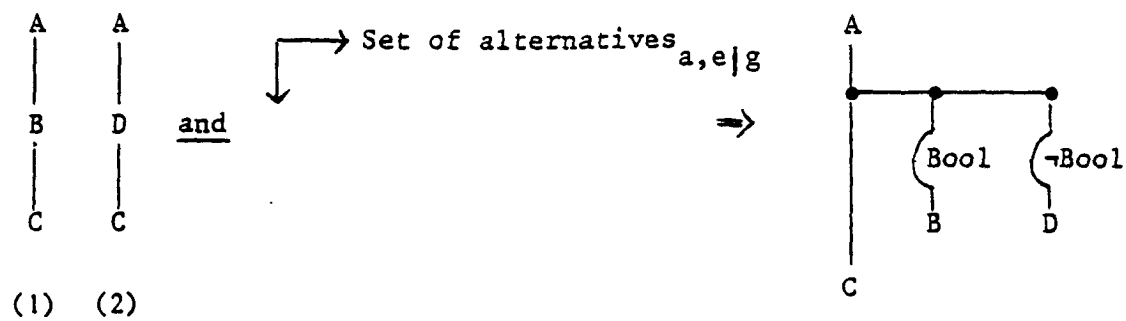
Axiom 14 : Redundant Recoding of recursive H-Reduction

If $\forall a$, $a \Rightarrow a \xrightarrow[g]{} B$ and B is an infinite subtree

then $a = a \xrightarrow{} \blacksquare$

6.2.3. Integration & Selection

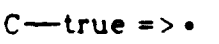
Axiom 15 : Integration

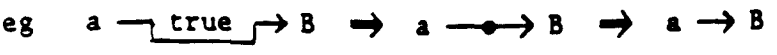


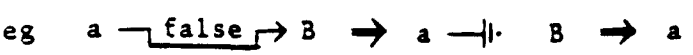
(1) (2)

where 'ς' is the Conditional symbol and 'Bool' is a Boolean expression which gives the criterion which differentiates subtree (1) from subtree (2) and $Set_{a,e|g}$ is a new, artificial relationship between the members of the set of alternative, selectable subtrees.

Axiom 16 : Selection







6.2.4. Induction & Deduction

Recursive Induction

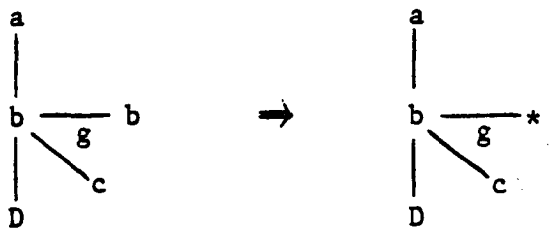
see axiom 8 : H-reduction

and axiom 14 : Redundant recoding of recursive H-reduction

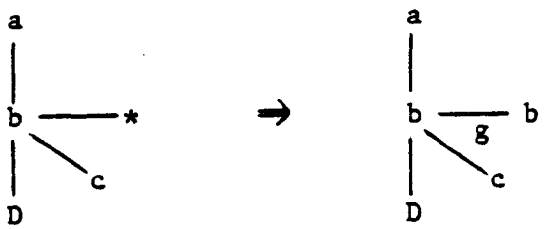
Recursive Deduction

see axiom 9 H-expansion

Axiom 17 : Iterative Induction



Axiom 18 : Iterative Deduction



Note: Axioms 17 and 18 are just special cases of recursion in which '*' saves the need to have 2 or more copies of a repeated component in the description. Only 1 is needed because the repetition is so simple and regular (tail recursion)

that '•' just means replace '•' by a copy of its parental node. The Generative relation will do the rest of the repetition (see section 3.5.1.3 Induction and figure 4-14). Iteration is a very popular construct in programs and the following, strictly incorrect, shorthand is used in Dimensional Design/ROOTS (see Chapter 7: Practical Experience):

Convention

a —e— b —e— c —• is shorthand for [a —e— b —e— c] —g— •

ie a set or sequence is defined enumeratively but is repeated generatively.

6.3. Description Reduction (Meaning Dependent) Axioms

The above description reducing axioms are independent of the meaning of the description they reduce. Thus many more axioms are needed to encapsulate the complete meaning of a description so that its real world analogue can be understood. Some of these relationship dependent properties can also be used to further simplify descriptions. For example the well known mathematical property of Associativity can remove a multiplicity of cuboids, see figure 6-1.

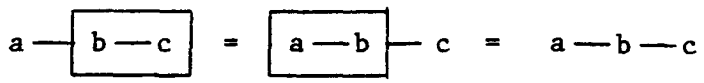
A proliferation of bracketing cuboids can be further stemmed if the relations involved can be given precedences. Figure 6-2 shows how the well known operator precedence can be used to reduce brackets (cuboids) in a one (three) dimensional description.

Precedence is exploited by ROOTS to avoid the excessive use of cuboids. A ROOTS Dimensional Design is input to the ROOTS compiler by treewalking it to produce a linear string of symbols. The treewalking algorithm chooses the order of visiting subtrees according to their fraternal precedences. These precedences of the programming abstraction are, in descending order, Refinement Hierarchy, Sequence and Set. The ROOTS compiler knows these precedences

and so can rebuild the original Dimensional Design from the linear form.

Description dependent axioms are as rich and numerous as the properties of the real world they try to capture and obviously they cannot all be detailed here. The description independent axioms are of primary interest to Dimensional Design.

The above description independent axioms and conventions are used below to illustrate, in figure 6-3, the deduction of a sequence of two 'B's from an inductive ie Generative description. Note the way the generative relationships are weakened to enumerative ones, the way an infinite subtree is disconnected to terminate the induction and the artificial hierarchy is removed to leave the real description.



where — is associative and enumerative

eg a + (b + c) = (a + b) + c = a + b + c

Figure 6-1. Exploiting Associativity.

(((A**B)÷C)+D) = A**B÷C+D

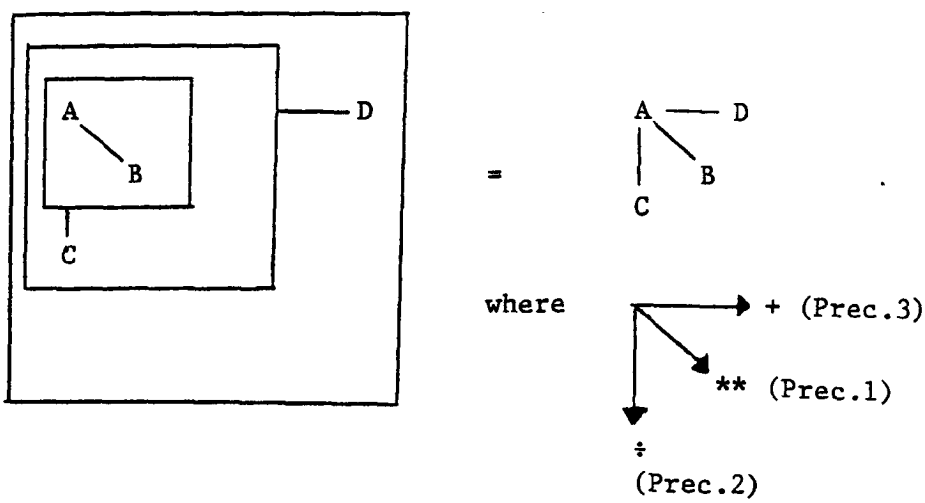
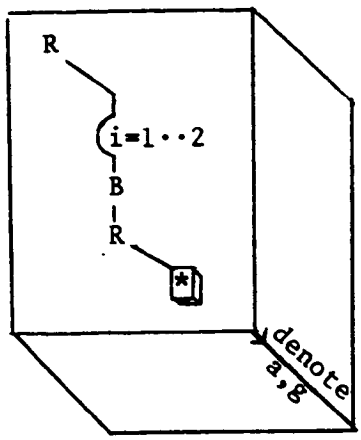
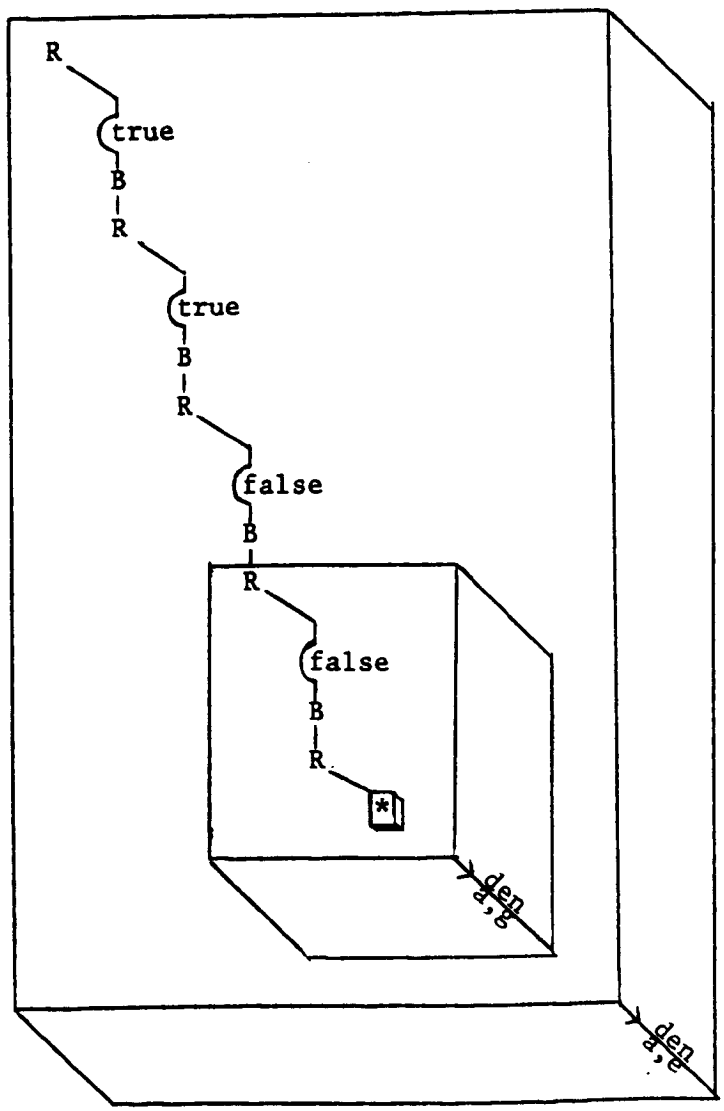


Figure 6-2. Exploiting Precedence.

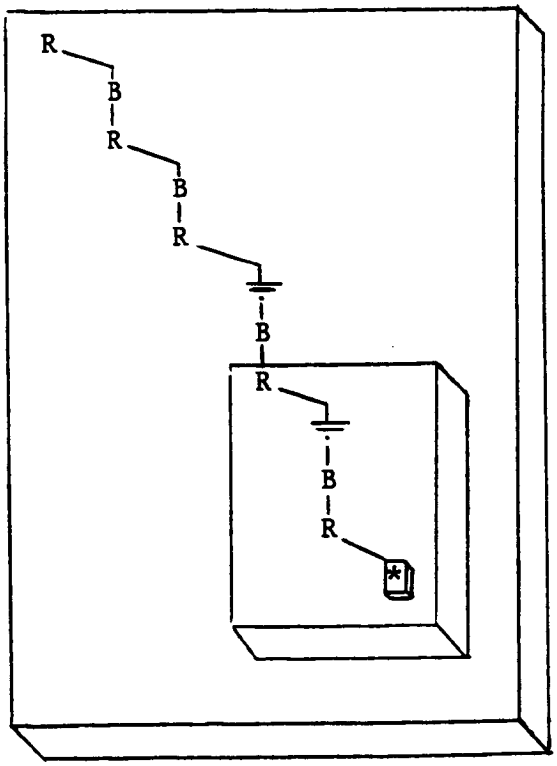


(a) Generative DD

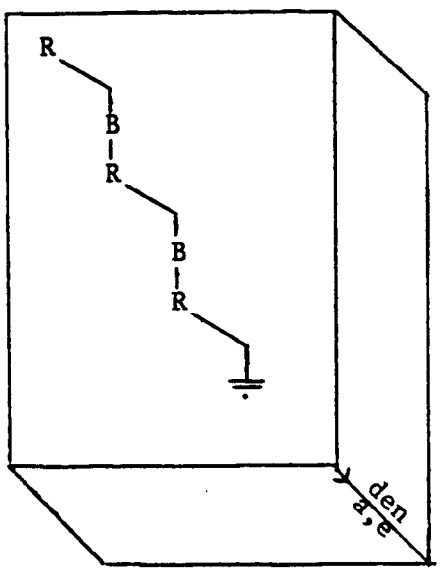


(b) Recursion to generate full description, then determine Selection Booleans.

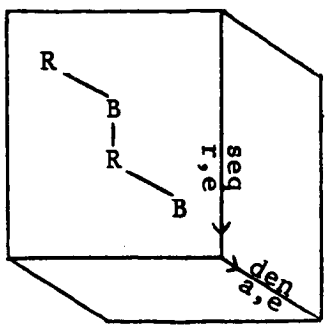
Enumeration (A10) has been used to divide DD into those 'den' relations which can now be enumerative and those which must still be generative



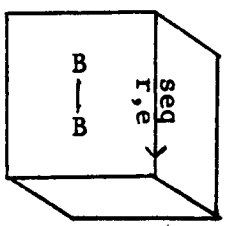
(c) Selection (A16)



(d) Disconnection remove infinite subtree



(e) Removal of null symbol (●) by A3



(f) Removal of Artificial Hierarchy '$den_{a,e}$' by A5 to leave Enumerated DD with all relations Real.

Figure 6-3.

6.4. Theorems

The above axioms may be used to reduce the sizes of descriptions. The concept of a n-Relationship Tree allows a quantitative approach to the scale of reduction possible.

Definition-16: The Size of a Dimensional Design is the total number of vertices (symbols) contained in the equivalent n-Relationship Tree. Note: In a tree the number of vertices (symbols) is one more than the number of edges (relationships) ie

$$v = e + 1$$

In a forest of trees

$$\sum v = \sum e + \sum roots$$

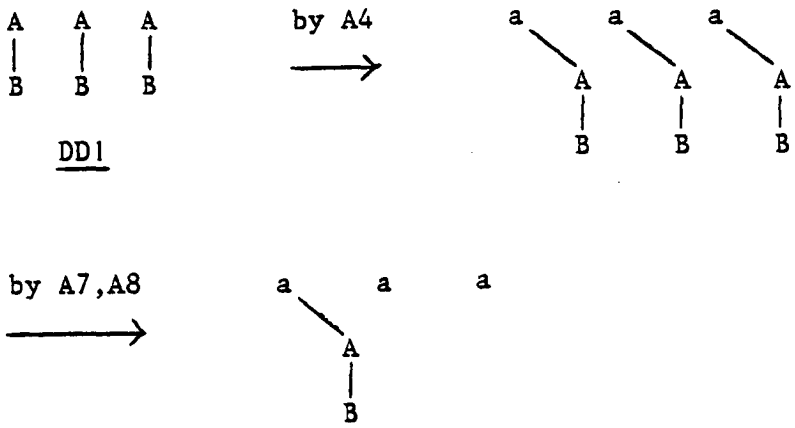
Definition-17: For the purpose of measuring the Size of a Dimensional Design the Size of a Conditional symbol is defined to be the size of the Boolean expression (itself a Dimensional Design) used to determine the polarity of the selection plus 1 for the Conditional symbol itself.

Theorem 1 : H-Reduction

Any forest of Dimensional Designs (DD1) which contains n, $n \geq 2$, identical subtrees may be described by a transformation (by H-reduction) of DD1 (DD2) such that $Size(DD2) < Size(DD1)$ if an artificial hierarchy built from '$denotes_{a,g}$' is introduced by applying axioms A4, A7 and A8.

Given: A A A..... n of, $n \geq 2$ whereA,B are subtrees
| | | of size SA,$SB \geq 1$
B B B

Example:



Proof:

$Size\ of\ DD1 = n * (SA + SB)$

$Size\ of\ DD2 = (SA + SB) + n * 1$

$DD1 - DD2 = n * (SA + SB) - (SA + SB) - n$

$$=(n-1)*(SA+SB)-n$$

Now $n \geq 2$ and $SA.SB \geq 1$

Therefore in the worst case $n=2$, $SA=SB=1$

$$DD1-DD2=(2-1)*(1+1)-2$$

$$=0$$

for $n>2$ or SA or $SB>1$ then $DD1-DD2>0$

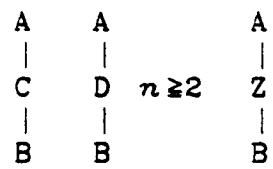
Hence $DD2 \leq DD1$

Q.E.D.

Theorem 2 : Integration

Any forest of Dimensional Designs (DD1) which contains n, $n \geq 2$, subtrees which only differ by one component subtree may be described by a transformation (by integration) of DD1 (DD2) such that $Size(DD2) < Size(DD1)$ provided that a selection criterion to differentiate between individual trees exists such that the size of the selection mechanism is less than the total size of the common components eliminated by using axioms A4 and A16 to build an additional, artificial hierarchy of $Set_{a,e}$

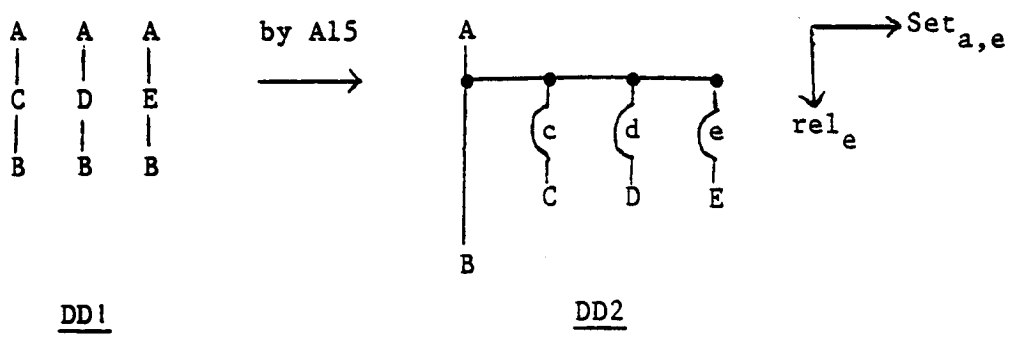
Given:



A,B,C... subtrees of sizes SA, SB, SC

Given: c,d,e... Selection criteria to distinguish between trees ie $c \Rightarrow C$, $d \Rightarrow D$ etc.

Example:



Proof:

$Size\ of\ DD1 = n*(SA + SB) + (SC + SD + SE + \ldots)$

$Size\ of\ DD2 = (x + n*y) + (SA + SB) + (SC + SD + SE + \ldots) + (Sc + Sd + Se + \ldots)$

where $x = 1$ is the horizontal null symbol

$y = 1$ is the vertical null symbol

$DD1 - DD2 = n*(SA + SB) + (SC + SD + SE + \ldots) -$

$(n + 1) - (SA + SB) = SC + SD + SE + \ldots) - (Sc + Sd + Se + \ldots)$

$DD1 - DD2 = (n - 1)*(SA + SB) - (n + 1) - (Sc + Sd + Se + \ldots)$

Now $(n - 1)*(SA + SB)$ = size of total saving of eliminated common components

and $(n + 1) + (Sc + Sd + Se + \ldots)$ = size of selection mechanism

therefore

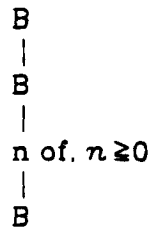
$DD2 < DD1$ if $(n + 1) + (Sc + Sd + Se + \ldots) < (n - 1)*(SA + SB)$

Q.E.D.

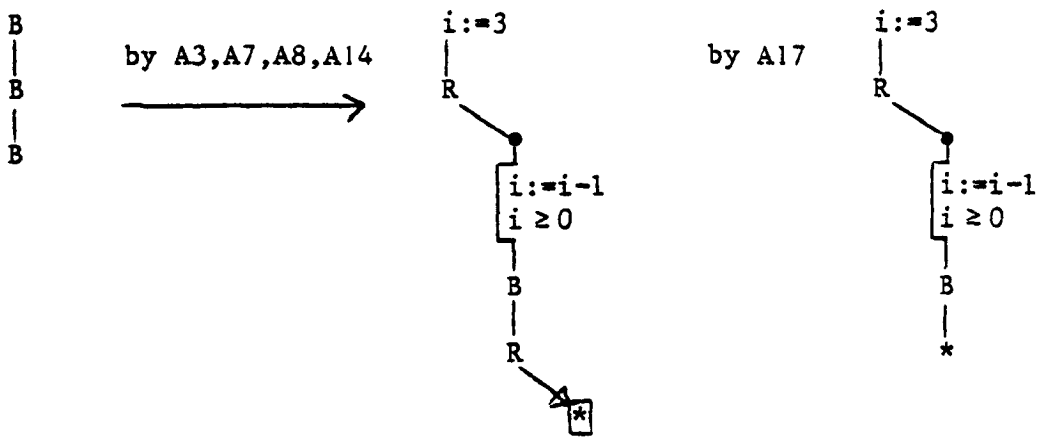
Theorem 3 : Iteration/tail recursion

Any Dimensional Design (DD1) which contains n, $n \geq 0$, consecutive identical subtrees may be described by a transformation (induction) of DD1 (DD2) such that $Size(DD2) < Size(DD1)$ if an induction criterion can be found which is smaller in size than the total size of the eliminated subtrees by using axioms A17 and A18 and convention A18C to build an additional artificial, generative hierarchy.

Given:



Example:



Proof:

$Size\ of\ DD1 = n * SB$

$Size\ of\ DD2 = SI + SR + SN + ST + SB + SIT$

where SI is size of initiatisation

SR = 1 is size of common subtree name

SN = 1 is size of null symbol

ST is size of test to terminate induction

SIT = 1 (iteration), = 2 (recursion) is induction indicator

$DD1 - DD2 = (n * SB) - (SI + 1 + 1 + ST + SB + 2)$

$= (n-1) * SB - (SI + ST + 4)$

now $(n-1) * SB = total$ size of eliminated redundancy

$(SI + ST + 4) = size$ of induction mechanism

therefore

$DD2 < DD1$ if $(SI + ST + 4) < (n-1) * SB$

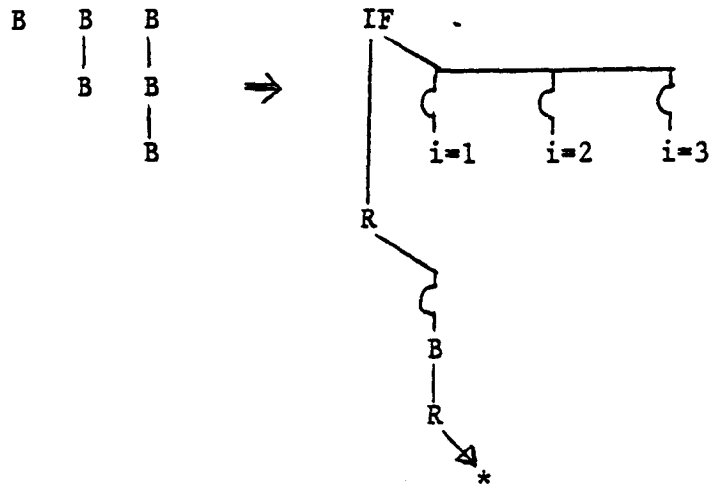
Q.E.D.

Implication-6: If, in theorem 3, $n >> 0$ then $DD1 - DD2 >> 0$ ie induction is very powerful.

Implication-7: Theorem 3 can be generalised to handle full recursion. Obviously as a finite sized inductive description can describe infinite sized

'objects' then potentially $DD\,1-DD\,2=\infty$

Implication-8: Integration may be combined with Induction to give further reductions eg:



Implication-9: Theorems 1, 2 and 3 each introduce an additional relationship specifically for reduction purposes. This increase in RELSET is not always necessary for it is often possible to use an existing member of RELSET as the generative relationship eg 'is refined into' can be strengthened to 'is *always* refined into'. This is an example of Shanley Integration[42] - the implementation of two separate functions by one (shared) mechanism.

6.5. From Theory to Practice

The above definitions, axioms and theorems mark the end of the formal theory of Dimensional Design which has attempted to show that Dimensional Design can be

1. formally defined

2. manipulated according to well formulated rules.

Whilst by no means is the above a complete, consistent and thoroughly rigorous presentation of a formal theory it does perhaps provide a clear indication of the existence of a fuller theory.

The existence of a Theory of Dimensional Design satisfies the first two (of four) tests to substantiate the Central Hypotheses (see section 4.9.1:Generalisation). The second two tests relate to physical representation and manipulation and to human creativity and understanding. These are the 'practical' tests, questioning whether Dimensional Design does indeed help in the design and production of better programs.

Mention of the practicalities of programming perhaps evokes in the reader some remembrance of the discussion of the nature of the Software Crisis with which this thesis began. That discussion led to an examination of the Software Life Cycle and Software Engineering. From this it was clear that Maintenance (ie rectification and development) was a major problem and that one possible solution was to initially produce better quality, more tractable software design products, designed with maintenance in mind. It was clear that it is not yet known how to produce top quality individual programs let alone software systems.

Chapter 3 began by concentrating on this Small Scale problem of producing the detailed design of individual programs. The modern ideas of Structured Programming and Correctness Proofs were examined and the following conclusions drawn:

Specification (eg assertion) => WHAT a program does

Detailed Design (eg source code) => HOW a program does something

Correctness Proof (eg predicate transformer) => WHY a program does something

The artificial model of computing, the Restricted Von Neumann Computer, was introduced to show that programming is essentially the act of describing complex instruction sequences and data. Abstraction was introduced as being the key tool needed to reduce the complexity of these descriptions. The concepts of Refinement Hierarchy, Set and Sequence were used in the programming context and Hierarchical Reduction, Integration, Induction, Enumeration and Generation were seen as more general descriptive techniques. Dimensional Design was introduced as a way of representing these techniques more explicitly than conventional programming languages. Dimensional Design was then developed through Hoare's four stages of Abstraction culminating in the conclusion that Dimensional Design was a (potentially) formally based, general technique for perceptually enhancing textual descriptions.

All this abstract discussion of a formally based design representation has to be related to the central problem of producing better-quality, more maintainable individual programs; that is can, from the Theory of Dimensional Design, an improved, practical method of small scale software engineering be developed? Can Dimensional Design pass the acid test of practical use?

CHAPTER 7. DIMENSIONAL DESIGN: PRACTICAL EXPERIENCE

OUTLINE

Set, sequence, hierarchy, h-reduction, integration, induction, enumeration and generation have been abstracted, represented, manipulated and axiomatised to produce the theory of Dimensional Design, but can it be applied to help in real software production? Chapter 2 described the various stages in the Software Life Cycle which is used to structure this chapter. At each life cycle stage Dimensional Design is examined to see if it can be applied advantageously. This chapter begins with details of real software projects already undertaken using Dimensional Design, an outline of the overall method they employed and the Dimensional Design specific software tools they utilised. One such set of tools, called Dimensional Design/ROOTS, is used to illustrate the application of Dimensional Design at each stage in the Software Life Cycle of the production of a non-trivial example program. Dimensional Design/ROOTS is used at the pencil and paper, drawing board, overall design stage. It produces high quality, machine drawn, detailed design documentation which is always neat, up-to-date and tractable, thereby helping with one aspect of the small scale software engineering problem. Dimensional Design/ROOTS contains novel, experimental quality control tools to tackle the other aspect of the central problem. Rectification and Development are further aided by a run-time monitoring system which measures CPU time, animates and records execution and contains other more conventional debugging tools, all the results of which

are represented according to Dimensional Design principles to maintain conceptual integrity with the design.

After summarising the practical experience gained with Dimensional Design the succeeding chapters compare it with existing techniques and make suggestions for future work.

7. DIMENSIONAL DESIGN: PRACTICAL EXPERIENCE

7.1. Introduction

Dimensional Design has been in practical use at the Science Research Council's Rutherford Laboratory since May 1975 and has been used in the construction of six major programs:

1. FR80 SYSLOG[68]

   a program to log accounting data about user jobs. Hand coded in Assembler (8000 lines).

2. FINGS[20]

   a graphics package coded in Forest (7,900 lines).

3. FR80 DRIVER[84]

   a multi-tasking control system for a microfilm recorder. Coded in DRIL (30,000 lines).

4. DFC[48]

   a program to draw Dimensional Designs on a variety of hard and soft copy devices. Coded in ROOTS (6,000 lines).

5. ROOTS[22]

   a Fortran pre-processor and Dimensional Design software tool-kit. Coded in ROOTS (4,000 lines).

6. MONITOR[22]

   a run-time monitoring package for ROOTS programs. Coded in ROOTS (3,000 lines).

In addition to the above Rutherford Laboratory projects, Dimensional Design/ROOTS is currently being used by Dr. B. Gudmanson's image processing group at Linkoping University, Sweden. Prof. Lawson and A. Jonsson[39] of Linkoping's Computer Systems department have produced an interactive

Dimensional Design editor. Dr M. Bertran's group at ENHER (a Spanish electricity supply company) are using ROOTS in the production of a real-time process control system.[6] Dr. Bertran has also produced Dimensional Design support tools for his text transformation system. T. Povey of the Burroughs Corp. (ASDC) uses Dimensional Design in the production of commercial applications packages and has produced some Dimensional Design/Cobol software tools.

Dimensional Design was created and developed during the design of FR80 SYSLOG. It was decided that Step-wise Refinement and Structured Programming principles would be used for this project. Applying these techniques brought forth the problem that no suitable design representation existed, so Dimensional Design was invented. FR80 SYSLOG's final Dimensional Design was refined right down to the bit level as Assembler was the only available language. The Dimensional Design was 'tree-walked' into Assembler source code and the final program of 8,000 instructions has been running successfully, unmodified, since September 1975.

Following FR80 SYSLOG was a larger project which produced FR80 DRIVER, a multi-tasking microfilm graphics control program to run on a bare machine. From the start of this project it was clear that development and maintenance problems would be major considerations over a predicted 7 year life-cycle. Based on the success of FR80 SYSLOG it was decided to again use Dimensional Design but, this time, supported by new, project specific software tools to combat the problems of size, complexity and staff-turnover etc. The system implementation language, compiler and Dimensional Design drawing tools were collectively called DRIL.[82] FR80 DRIVER, which has been running production jobs since June 1979, currently consists of 30,000 lines of DRIL source code and is represented by a dozen Dimensional Designs each of which contains the equivalent of 1-3,000 lines of source code. As each drawing is around four feet

square it is impossible to reproduce real examples of Dimensional Designs in this thesis. However, all the remarks in this chapter about 'real' Dimensional Designs should be related to such large drawings as Dimensional Design was primarily invented to represent the logic of complex modules of real, production systems. Many of the techniques developed and lessons learnt on these two projects were described in three published papers[83,85,3] - see section 10.1:appendix.

The success of the project specific Dimensional Design/DRIL implementation tools led to the development of a more general purpose tool, FOREST,[21] based on a Fortran pre-processor. Encouraging results with FOREST on a production project, FINGS,[20] led to the creation of an experimental comprehensive software engineering 'tool-kit' called ROOTS.[22]

In the following sections ROOTS will be used as a vehicle to show how the Dimensional Design technique can be used, in practice, to design and construct a working program. An attempt will be made to show, at each stage,

1. What a programmer actually does.
2. Which tools he employs.
3. What conclusions can be drawn from actual practical applications experience as well as this artificial example.

It is regretable that the development of a 'production-sized' program cannot be illustrated but instead a small, highly artificial, contrived but non-trivial example, Binary-to-Decimal conversion, must be employed. Many of the issues in software construction are functions of size and such toy examples are not, unfortunately, accurately scaled-down models of real problems. Even so, the development of a 'production quality' Binary-to-Decimal example program will require a considerable range of skills, tools and hard work. The overall method which will be employed in the succeeding sections is as follows.

Suppose a program is being built from an extant specification. The initial, highly creative design work is done on the 'backs of envelopes'. Top-down or Step-wise refinement of the initial design sketches generates a proliferation of small Dimensional Designs. As the design ideas firm up this set of small, scrappy drawings is combined into a single, unified drawing on one large sheet of paper of up to say four feet square. A full sized draughtsman's drawing board is employed for this task. The equivalent of up to 4,000 lines of conventional source code can be expressed in one such drawing. For larger programs one drawing will probably represent one major module.

Refinement of this design drawing continues until a complete program or module is derived ie when all the terminal nodes of the design tree structure are compilable source code. At this stage the Dimensional Design is encoded into one of the special languages such as DRIL or ROOTS which support 3-D hierarchical structuring. Coding is the process of reducing the logically 3-D and physically 2-D design into a 1-D machine readable form. This is simply achieved by 'tree-walking' the Dimensional Design, ie going through the Dimensional Design tree in a particular order, encoding each line and symbol as it is 'visited'. The DRIL and ROOTS compilers can, from this 1-D form, internally recreate the original 3-D design as they 'know' the encoding/traversal algorithm which is based on the 'preorder traversal' grammar in Chapter 6.

The machine readable version of the Dimensional Design is then compiled. The output from the compilation system is a binary program and, instead of the conventional source listing, a neat, well laid out Dimensional Design. Thus the visible output from the compiler is a logically exact, but physically neater, replica of the program's original design. Any subsequent changes and recompilations will produce new up-to-date Dimensional Designs too. Figure 7-1 illustrates this circle from hand-drawn design to machine drawn documentation

which is a key step forward in producing accurate design documentation.

Execution of a ROOTS-produced binary program produces the desired 'results' plus an optional amount and variety of monitoring information about the program's run-time behaviour. Such items as CPU time and I/O time are logged, then used to augment the actual design documentation itself, for presentation, along with the 'results', to the programmer so that he may examine, side by side, the program's static design, its internal dynamic behaviour and its external, required 'results'. Armed with comprehensive, accurate and up-to-date information the programmer is now in a strong position to pursue the successful development of the required program.

The above method will now be applied to the development of the Binary-to-Decimal program to show the Practice of Dimensional Design.

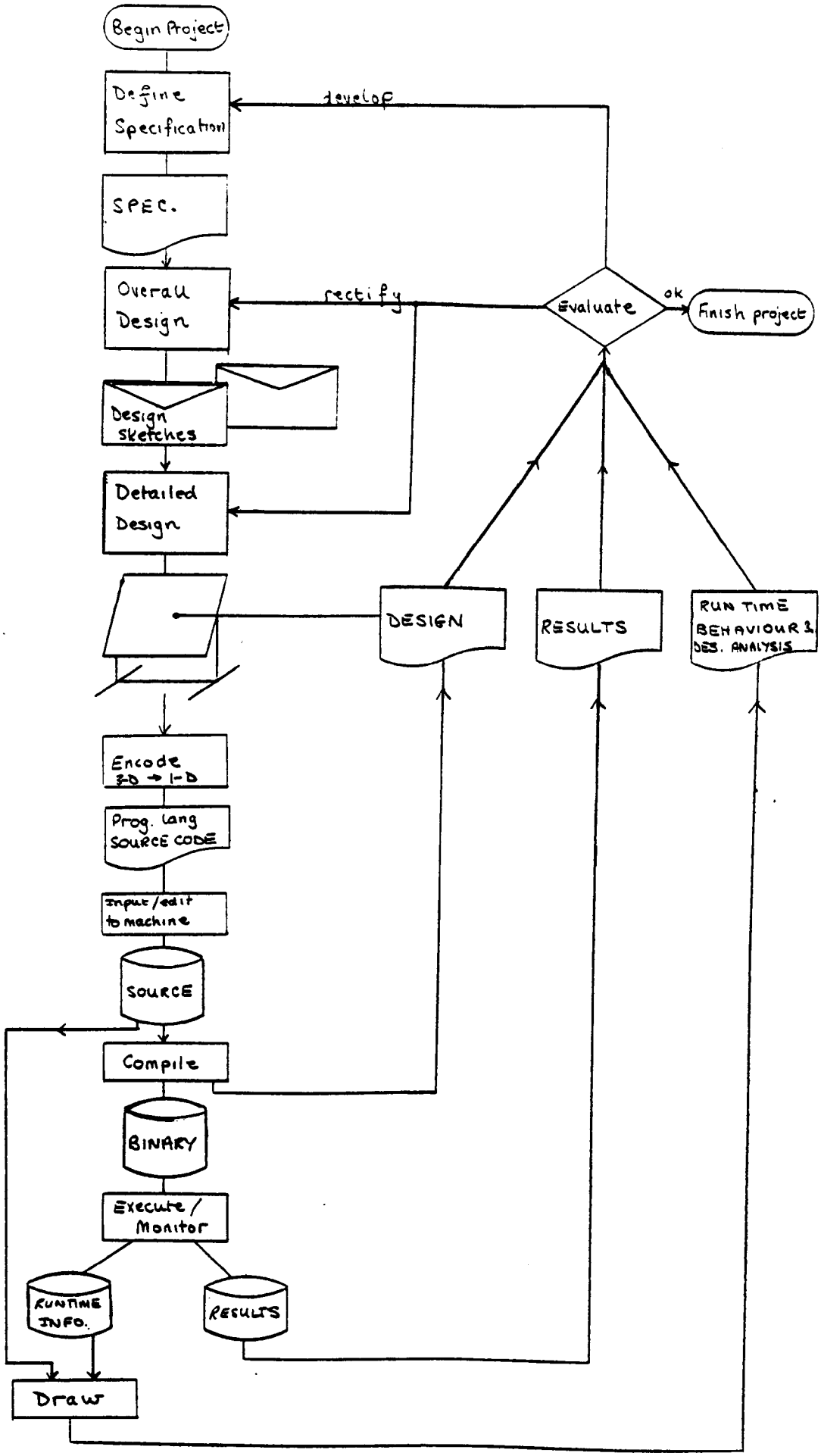


Figure 7-1. Dimensional Design/ROOTS - Method & Tools.

7.2. Requirements Specification

An informal specification of the Binary-to-Decimal conversion example program is:

1. The program name is to be 'B2D'.
2. The program is to implement both an iterative and a recursive method of converting signed binary integers (expressed as 2's complement bit strings) to their decimal equivalents (expressed as strings of decimal digit characters preceded by sign characters)
3. The two conversion algorithms should be implemented so as to bring out, in a tutorial manner, their similarities and differences.
4. The program's design and construction is to illustrate most of the features of the Dimensional Design technique and the ROOTS software tool kit.

No serious practical work has been done on the use of Dimensional Design techniques to represent Requirements Specifications. This is a potential field for future work as currently most specifications are both informal and textual. The above textual specification will therefore be used as the starting point for the overall design work.

7.3. Overall Design

The Overall Design phase is one of the most creative stages in the software life cycle. The software designer invents solutions and compares them against each other. After careful evaluation the chosen solution progresses to the Detailed Design stage. During the Overall Design phase the designer needs a representational technique with which to easily sketch out prospective solutions - real 'back of the envelope' stuff! Figure 7-2 is an attempt to illustrate the use of Dimensional Design to help in the Overall Design of the B2D program.

It shows how the designer has considered two possible solutions and which one he finally chose and why he chose it (see section 4.4.3:Design Alternatives), information of immense value to the maintenance team later on in the life cycle as well as, say, a design conference or walkthrough. Figure 7-2 shows the designer to have chosen the 'complicated' solution. With such a simple example program, Overall Design is rather a grandiose title for an activity which is, in this case, really Detailed Design. The benefit of using the same representational technique for both design phases means that the division between the two stages can be removed to good effect. Contrast this unification with say the divisive effect of flowcharts (Overall) and source code (Detailed), a process which inevitable leads to the rapid obsolescence (and abandonment) of the Overall Design representation with a consequent loss of information to the design and development teams (see reference[85]).

Figure 7-3 again tries to show the designer at work, this time sketching out the subroutine organisation. He decides to organise the subroutines as levels of abstract machines (see section 4.5.2) an idea supported explicitly by ROOTS (see section 10.2.2 and 10.2.7).

During the Overall Design phase the main tools used by the designer to produce a Dimensional Design are paper, pencil and eraser(!). If the design is large then he may use a draughtsman's drawing board. These tools have proved satisfactory on the projects described in section 7.1. Obviously one could imagine the use of an interactive computer graphics computer-aided design system at this 'sketching' stage. There is obviously a trade-off between the cost and sophistication of the design tools and the effort needed to design the required software. For the relatively small scale projects undertaken to date, pencil and paper have been the only economic tools and have worked extremely well. Indeed, one of the major attractions of Dimensional Design is

the ease of free-hand drawing compared to other representational techniques. This aspect and further management issues are discussed in [85]

Having produced an Overall Design the designer can now proceed to refine and expand his ideas - the Detailed Design stage.

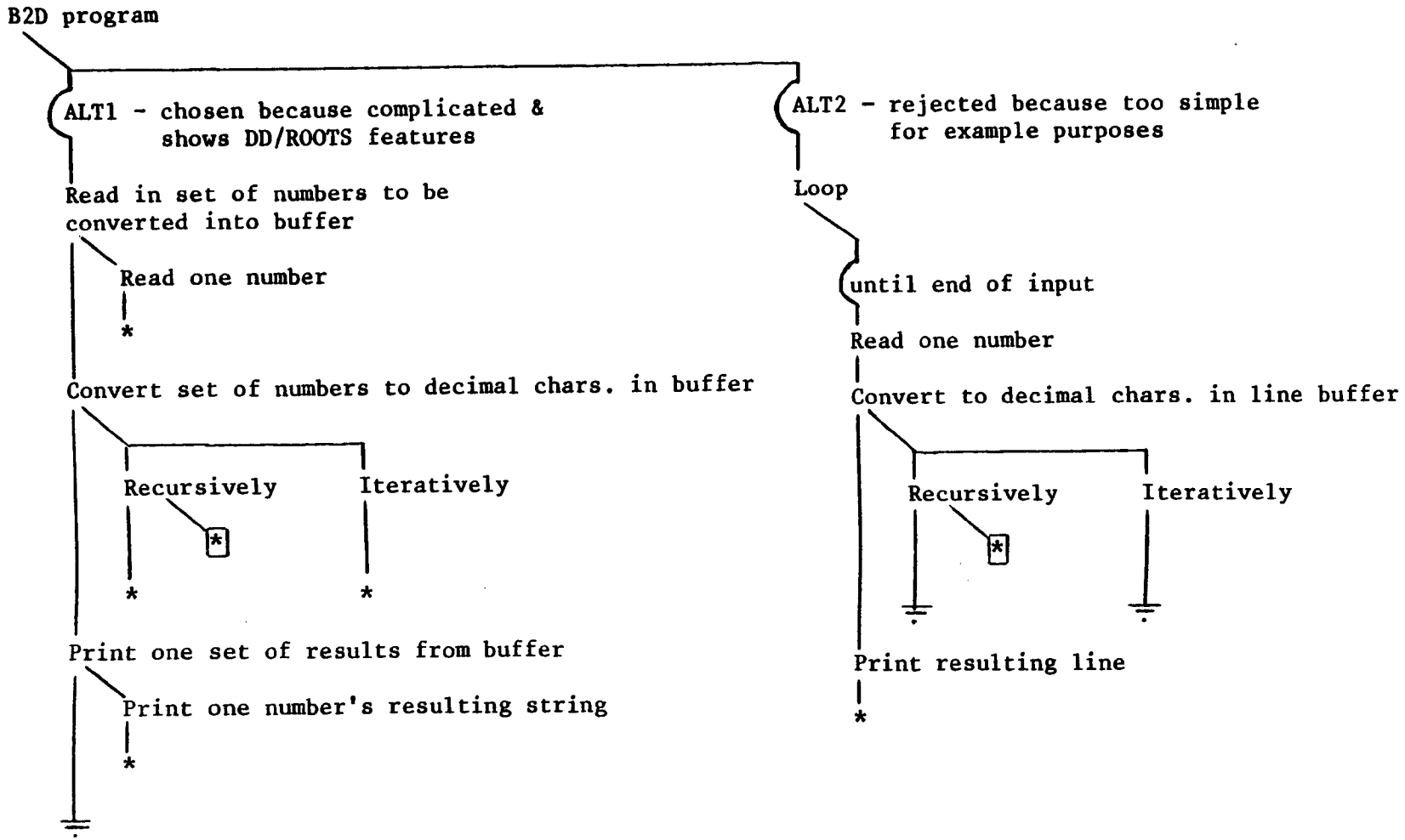
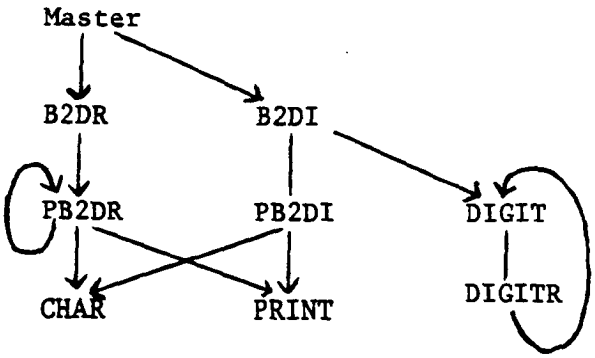
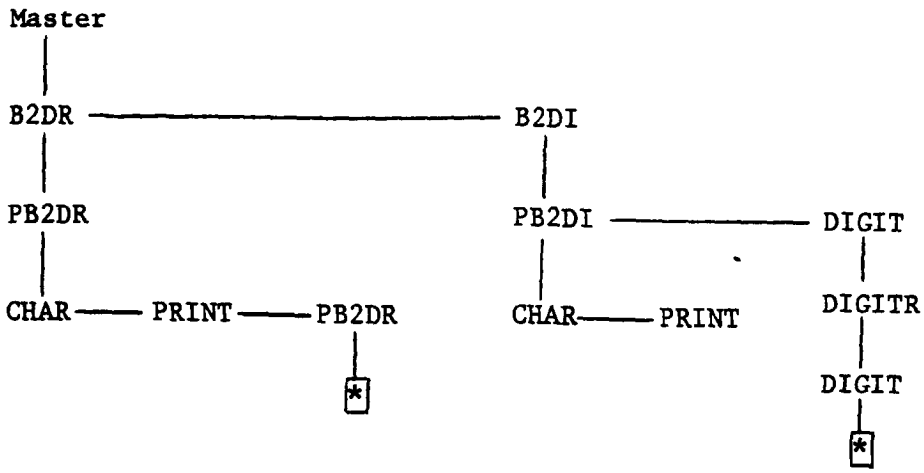


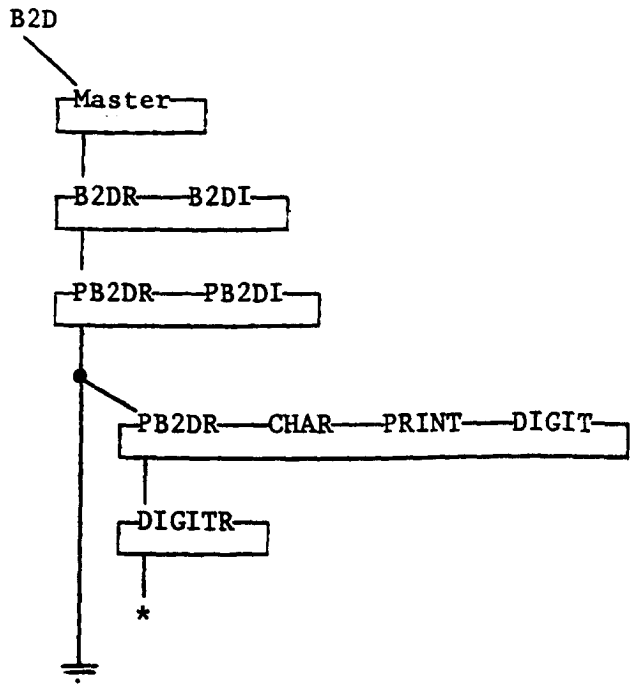
Figure 7-2. B2D Overall Design Sketch.



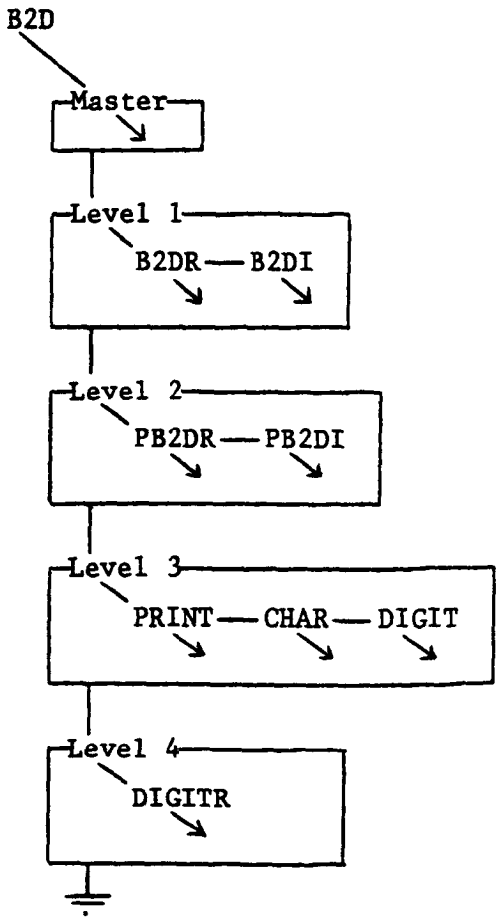
(a) Call Graph



(b) Call Tree



(c) Levels (generative)



Note: ROOTS designed for non-recursive FORTRAN

Note: DIGITR is to circumvent compiler error!

(d) ROOTS Layout

Figure 7-3. Design of Subroutine Organisation.

7.4. Detailed Design

It was argued earlier (section 2.4) that the final product of the Detailed Design phase of the Software Life Cycle is the (compilable) source code program text. Conventionally, a source code program is a textual description with only one explicit relationship between individual symbols, that of juxtaposition ie a source code program is usually a machine readable 1-D linear string of symbols. Typically a source code program contains little or no design information ie information as to how or why sections of source code were created. At best programs contain only informal comments. One of the major objectives of Dimensional Design/ROOTS is to capture formal design information in the source code program by including more explicit relationships between symbols and by retaining the design hierarchy as an integral part of the source code. Chapters 3-6 showed how this could be done in theory. This chapter is concerned with practical programming so Dimensional Designs will be used to represent the 'programming abstractions' of Sequence, Set and the 'What-How' hierarchy (Refinement). As is usually the case when one moves from the theoretical domain to the practical, one adopts certain simplifying assumptions and conventions. Practical Dimensional Design uses the conventions given in section 5.3. For example, cuboids are not used to enclose lines of text and the default axes labels are:

vertical *Sequence*

horizontal *Set*

diagonal *Refinement*

These conventions serve to make a program's Detailed Dimensional Design look very similar to a conventional programming language in which only the key

relationships have been made explicit. In practice individual projects and sites have tended to make their own conventions, small changes and adaptations to the Dimensional Design notation (as laid down in Chapter 4) to suit their own particular needs - see references.[6.44] Adaptability would seem to be a strength and an advantage of such an essentially simple idea as Dimensional Design.

Some of the conventions used in ROOTS are that Sets are expressed as sets of sequences of which the first element is always the null subsystem. This trick is used just because it gives a little extra clarity (see figure 7-4).

It has been found in practice that in almost all cases enclosing cuboids can be omitted without confusing the meaning of the design whilst improving its visual clarity (cf 'lines of text' above). It has also been found that by far the most popular 'projections' involve the three relationships, Set, Sequence and Refinement. Other projections such as Data Path Dimensional Design and Representation Down to the Operator Level (see section 4.8:Subsystems) have not been implemented in ROOTS because they are too rarely used. Neither has explicit Step-wise Refinement been implemented because, although in this thesis top-down design is used almost exclusively, it has been found best just to portray the final product of the Detailed Design activity in the experimental context of ROOTS. Figures 7-5 and 7-6 try to show the top-down evolutionary process of producing the detailed design of B2D. Of course in practice things never go so smoothly! The complete Detailed Design for B2D is given in section 10.2.2.

Section 10.2.2 gives the detailed design of the logic of B2D. Many other aspects of the design have been omitted. For instance, on a production quality project one might like to see proofs, performance prediction calculations, storage calculations, variable name cross reference listings etc. which are all part of the design product but are omitted as they are not directly relevant to

the Dimensional Design representational technique. However they will be considered later in the chapter.

During the Overall Design phases of projects Dimensional Designs have proved quick and easy to draw. They provide detailed, generative constructs (ie h-reduction, integration and induction) even at the most abstract level of design which is an advantage over some design notations. The explicit representation of hierarchies such as 'refinement' has proved of great help in creating clean designs.

It has been easy to produce formal Detailed Designs from the Overall Design sketches via a draughtsman's drawing board because Dimensional Designs

1. contain no complicated boxes or shapes,
2. have 'statements' (textual subsystems) of any length which do not have to be shoe-horned into Procrustean fixed sized boxes,
3. require few special symbols and rules,
4. need no templates for drawing - only a pencil and ruler,
5. their simple drawing/formatting algorithm (see section 5.3) *really* is easy to use in practice, for example, contrast figures 7-5 and 7-6.

Using one sheet of paper to hold the detailed design of a large module with a unified notation for both program and data structures has helped greatly to improve the quality of software design. Each sheet is an 'engineering drawing' containing the many macros, subroutines and data structures, necessary to implement a complex function such as a sophisticated device handler. Because of the explicit Refinement hierarchy, comprehension of the design is aided by individual 'statements' being typically, refined into less than 10 sub-statements each (see section 7.5:Static Analysis). This is recursively true for

any depth in the hierarchy.

The large single sheet approach to design enables the designer to view his design at any desired level of abstraction. He may investigate the tiniest detail or take a global view. This ability to see the global picture has enabled messy parts of a design to be spotted easily. A proliferation of nested 'if-then-else's is very easy to spot and turn into a 'case' statement. Experience has shown that good, clean designs actually look good, simple and clean when represented as Dimensional Designs. The global view also helps in deciding which pieces of logic to extract as macros and subroutines as similar shaped subtrees are often due to similar functions. The converse can also be true. In DRIVER two sections of code which were meant to be functionally identical (but which were not worth making into a subroutine) were seen to have slightly different shapes. Finding and fixing a bug in one made the two shapes identical.

The global view afforded by the single sheet design can be shared by several people simultaneously, especially if the sheet is held vertically. Discussions, design inspections and walkthroughs are greatly helped by this accessibility. Contrast this with, say, four people looking at the same program listing whilst keeping half a dozen fingers in it to mark the positions of the subroutines currently of interest to them. (However, there is no reason why a Dimensional Design should not be represented on several related sheets of A4!).

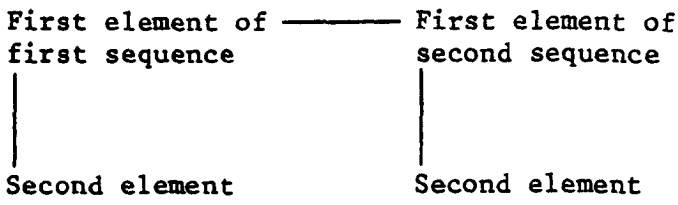
Dimensional Design makes programming/design progress visible. Managers are now able to see, and appreciate, how an Overall Design has progressed as they can see the expansion that has taken place during refinement to a more detailed design (see how figure 7-2 grows to 7-5 and 7-6). This helps to prevent the premature rush to produce code, code being, conventionally the only evidence of progress. Presentations to managers can now be by way of neat, possibly machine drawn, Dimensional Designs rather than, probably to

them, unreadable program listings. Dimensional Designs make it easier for a project leader to see how far there is still to go and they help him estimate the remaining amount of work to complete the refinement of a partial design. It is easier for him to see how to split up the work and for any individual programmer to see where 'his bit' fits into the whole system as he can see the higher level structure above 'his bit'.

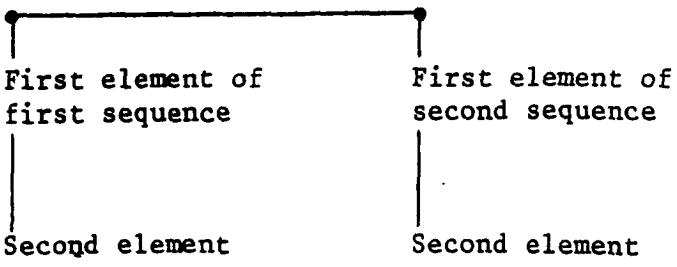
With one representational technique now spanning the whole process from initial design right through to detailed coding and maintenance on different projects (eg DRIVER & FINGS) using different languages (DRIL & FOREST) Dimensional Design has simplified communication and discussion between projects and eased the transfer of staff from one project to another.

As a large design progresses it must be periodically redrawn and tidied up. This can be tedious and time consuming. Thus large, partially refined designs are 'treewalked' into their target source code and compiled. A binary program cannot be produced but a neat machine drawn Dimensional Design can be. The large design is thereafter altered and expanded by editing a conventional source file (see figure 7-1); ideally a computer graphics terminal should replace the drawing board, source file and editor.

Small changes do not now mean a lot of redrawing by hand, and different versions of the same design are easily created and stored. This means that real, accurate, up-to-date design drawings can be easily produced, copied and circulated amongst project members and managers. Such mechanised assistance marks the beginning of the Construction phase.



(a) Theoretical



(b) Practical

Figure 7-4. ROOTS Set Convention.

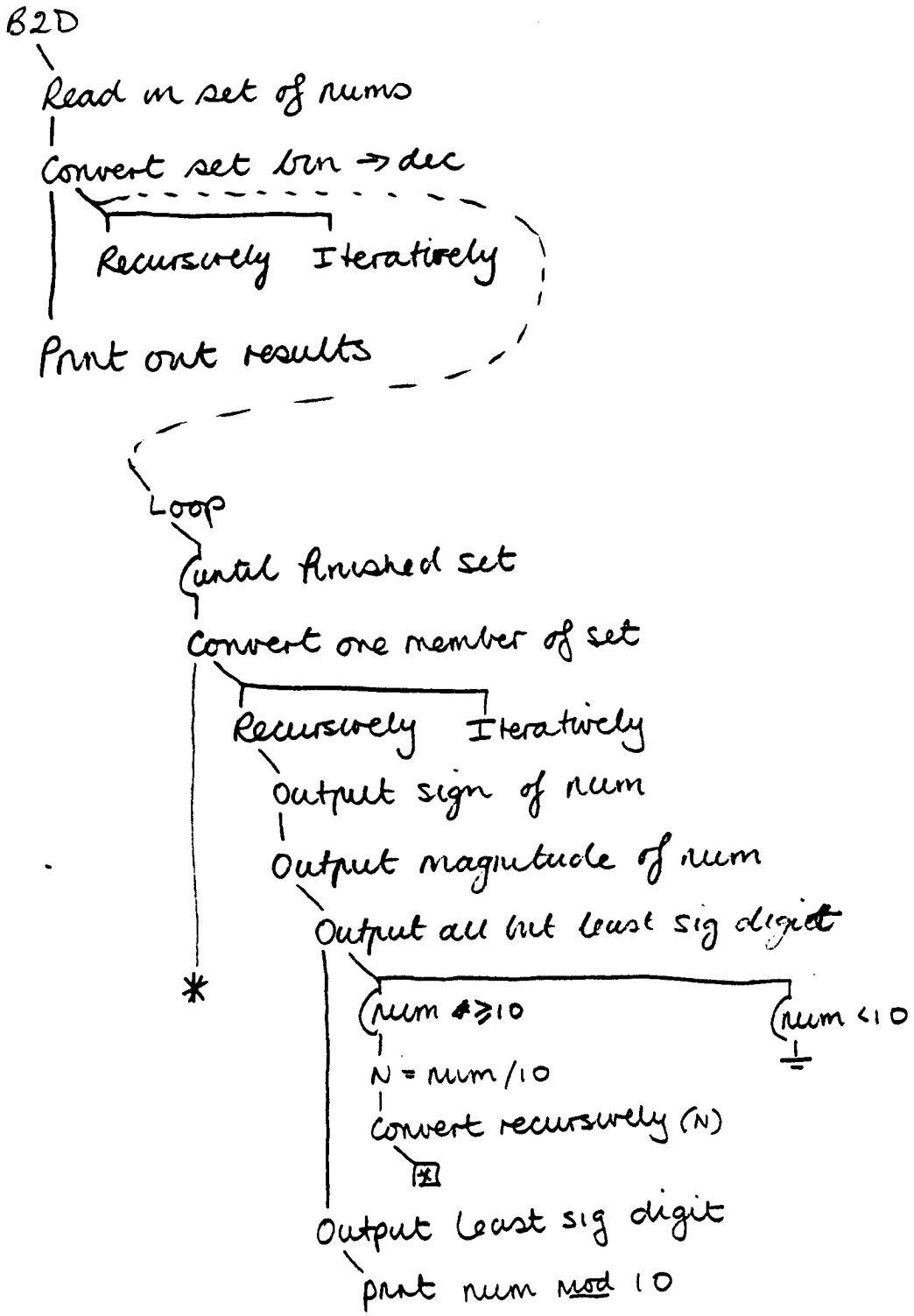


Figure 7-5. Freehand Intermediate Detailed Design Sketch of B2D.

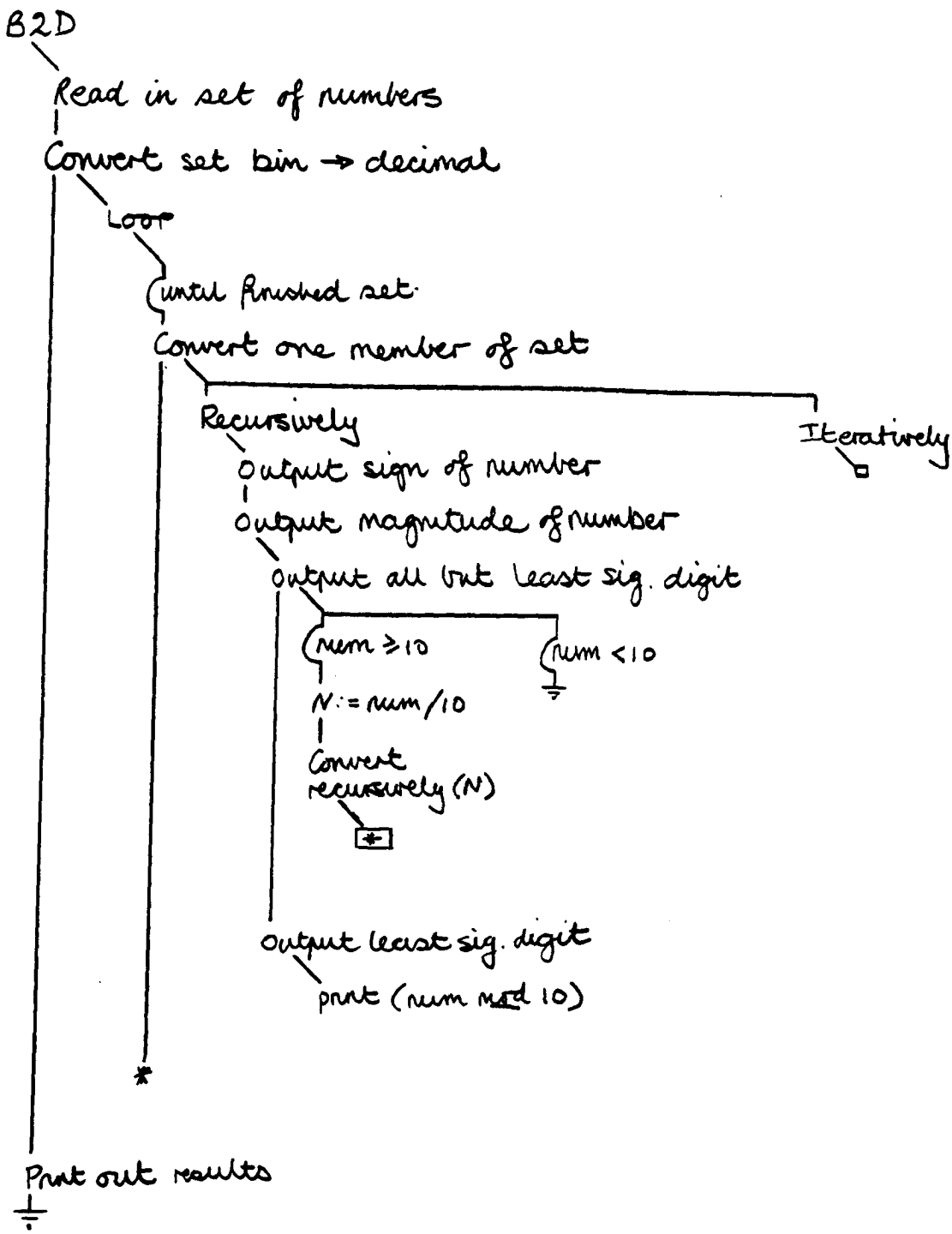


Figure 7-6. Drawing board version of figure 7-5.

7.5. Construction

7.5.1. Coding

The Construction phase produces, from the detailed design, an executable binary program plus various pieces of documentation. The first stage of Construction is the transformation of the final detailed design into a machine readable form. (This will be unnecessary if CAD tools were used in the previous phases.) For the B2D program the final design is a hand-drawn Dimensional Design. This must be encoded, by treewalking, into the ROOTS source language. The key point of this exercise is that the whole final design is encoded, not just the executable (terminal) leaves of the design tree. Full details of the ROOTS language are given in reference[22] - see section 10.1.2). The main features of ROOTS, over and above its Fortran base, are designed to facilitate the encoding of Dimensional Designs and are outlined below.

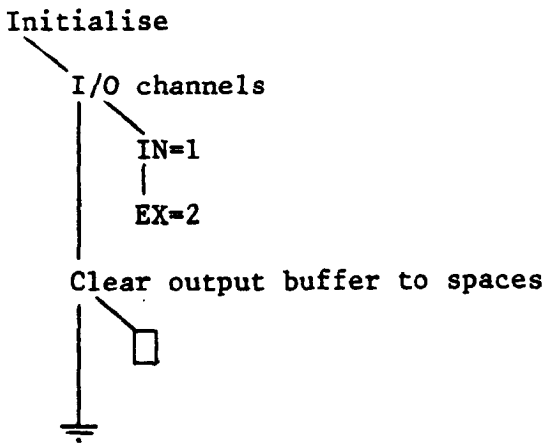
7.5.1.1. Artificial Hierarchy Addition/Removal

To capture the expression of the What-How Refinement Hierarchy (the 'diagonal' relation) ROOTS has a novel language feature known as the Scoped Comment.[83] Conventional comments only indicate the start of their 'influence' or scope but never the end of it. A simple marker (.EC) shows the boundary of a Scoped Comment's influence terminating the sequence of statements which is the 'How' of the comment's 'What' (see figure 7-7).

ROOTS forces all Scoped Comments (.C and .N) to be refined into compilable source code statements unless use is made of the 'stub' concept. This is most useful during both the design phase (to get neat drawings) and during construction and testing if the system is built (in contrast to designed) according to the top-down method (see figure 7-7).

A third ROOTS feature is the ability to turn on and off the portrayal of the

refinement of individual Scoped Comments during the drawing process. This allows the programmer to only have drawn in detail those particular aspects of the program in which he is currently interested. In this way potentially huge and unmanageable drawings can be reduced to a convenient size without losing any relevant information.



(a) Dimensional Design

```
·C  Initialise
    ·C  I/O Channels
        IN=1
        EX=2
        ·EC
    ·C  Clear Output Buffer to Spaces
        ·IG Stub
        ·EC
    ·EC
```

(b) ROOTS Source Code

Figure 7-7. ROOTS Scoped Comments and Stub.

7.5.1.2. H-Reduction/Expansion

ROOTS has three features to support H-reduction. They are macros, subroutines and ADD files.

The ROOTS tool kit includes the Kernighan and Plauger[40] macroprocessor. This is a powerful and convenient way to provide H-reduction/expansion capabilities. It was adopted because a) it was good and b) given its availability it would have been a waste of effort to build a special ROOTS macro system (see use of 'define' in B2D source code - section 10.2.7).

The basic subroutine mechanism in ROOTS is that of the underlying Fortran language. However the .LEVEL construct has been provided to allow designers to organise their subroutines into layers of abstract machines if they so wish (see section 4.5.2:Further Instruction Examples and figure 7-3). The .CALL statement has been enhanced to allow the subroutine's level as well as name to be specified. A recursive subroutine call may also be identified by .CALL(*).

The third feature is the .ADD command which replaces itself with the contents of the file named as its argument. The H-expansion of the .ADD statement may optionally be drawn out or not as required.

7.5.1.3. Integration/Selection

The ROOTS language has unified the syntax of the IF and CASE statements and requires a mandatory ELSE clause for which the OK, NULL and FAIL statements have been designed. For example note the difference in meaning (but not action) between the (theoretically redundant) ELSE clauses in the following examples:

```
.C Check status
   .IF (STATUS.EQ.BAD) .THEN
      .FAIL(ex,'bad status')
   .ELSE
      .OK
   .ENDIF
   .EC
.ASSUMPTION 1:Status is ok to proceed
```

```
.C Make sure value is positive variety
  .IF (VALUE.LT.0) .THEN
     VALUE = VALUE * -1
  .ELSE
     .NULL
  .ENDIF
  .EC
.ASSUMPTION 2: New value = ABS(old value)
```

7.5.1.4. Induction/Deduction

In addition to the traditional FOR and WHILE loop constructs, ROOTS has CYCLE which is a bounded, multiple termination, iteration construct especially designed for Safe Programming.[3] An example is given below. (Note the hierarchically structured UNTIL-EXITIF construct).

```
.CYCLE K=1,lenmax .TILL (1) .DO
  .UNTIL (end of input) .IE
    .EXITIF (BIN(J).EQ.terminator) .TOSITU(1)
  .C Convert Jth binary to decimal
    .IG stub
  J=J+1
.REPEAT
  .SITU(1)
    .OK
  .LIMIT
    .FAIL(ex,'loop bound exceeded')
.ENDCY
```

7.5.1.5. Proof

ROOTS contains features to help with both the proofs of termination and correctness. Proof of termination is assisted by the CYCLE construct (see section 7.5.1.4 above) which is designed to be used with the Safe Programming

technique ([3] - see section 10.1.3) in which termination is made simple to prove by bounding all loops.

Proofs of correctness are aided by the ASSERTION (run time verification possible) and ASSUMPTION (no run time verification possible) statements which allow proof aspects to be included directly in the detailed design. The OK, NULL and FAIL constructs were also designed with correctness concerns in mind. A secondary feature is the absence of constructs in ROOTS which make proofs difficult (eg GOTOs).

7.5.1.6. Run-time Monitoring

The ROOTS language contains features to allow the programmer to control the ROOTS MONITOR, a sophisticated run-time support package which produces traces, snapshots, control flow histories and performance measurements (see sections 7.6 and 7.7).

7.5.1.7. Inadequacies

It should be born in mind that ROOTS is an experimental tool kit not a production system and as such it suffers from certain inadequacies, stemming mainly from the fact that it is essentially a Fortran pre-processor and not a specially constructed compiler. The main inadequacies over and above those common to all preprocessors are

1. ROOTS (like Fortran) has little support for data structures (DRIL has more).
2. ROOTS does not yet support the full cuboid Dimensional Design representation.
3. The subroutine level structuring mechanism is not general enough to handle recursion (most Fortrans do not handle recursion).

Most of the above features mentioned in sections 7.5.1-6 above can be seen in the ROOTS source code version of B2D given in section 10.2.7 which is a machine readable version of the detailed design.

7.5.2. Production of Documentation

As soon as a machine readable version of the Detailed Design is available it can be input to a variety of software tools which can produce further valuable design documentation such as cross reference indices etc. ROOTS produces two documents, by analysing the source code, which are relevant to the Dimensional Design technique. One is to do with Quality Control (see section 7.5.3) and the other is the neat, machine drawn version of the Detailed Design itself. The initial compilation produces a machine drawn Dimensional Design which should be logically identical to the hand-drawn version from which the ROOTS source code was derived. This direct, visual comparison enables the encoding process to be verified in a novel way (see figure 7-1). If at the stage of initial compilation the ROOTS source code is not 'compilable' because it contains stubs then the programmer is legitimately using the compiler as a CAD tool.

From this point on in the Software Life Cycle any subsequent modification to the program which involves recompilation will automatically produce a new, right-up-to-date copy of the detailed design drawing which is a step forward towards solving the maintenance problem. If the maintenance is sloppy and changes the terminal parts of the design without appropriate amendments to the relevant design hierarchy, experience has shown that this degeneration quite quickly becomes (literally) visible in the Detailed Design drawing and can sometimes be detected from the Quality Control documents produced during compilation.

7.5.3. Quality Control

Detailed designs, expressed as Dimensional Designs, have, to some extent, the following desirable properties:

$$\left\{\begin{array}{c}different\\well-structured\\badly-structured\end{array}\right\}\ programs\ look\ \left\{\begin{array}{c}different\\well-structured\\badly-strutured\end{array}\right\}\ to\ the\ reader's\ eye$$

whereas detailed designs expressed as conventional source code do not have these properties because large numbers of lines of source code all look rather similar (see Chapter 8 for further discussion). These highly visual properties of Dimensional Designs mean that sloppy maintenance becomes, quite literally, visible. However, a problem of scale arises. The maintainer of non-trivial programs (and certainly the manager of several such maintainers) does not have the time to constantly keep visually inspecting all of the code for which he is responsible and so the question naturally arises "can this inspection be automated?".

Lord Kelvin once observed that "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be". So, to automate the inspection of Dimensional Designs requires that a metric be defined which measures or captures what, at this stage, is just an intuitive feeling that the maintainer senses when he looks at a Dimensional Design. He is somehow unconsciously abstracting some property or properties by means of his overall vision of the diagram; this overall picture is a function of the logical structure of the program being represented. For example, experience has shown that a program having a 'poor' descriptive hierarchy (in the refinement sense) will be visually fairly 'flat'. Maintenance

work to 'improve' the quality of the design by adding explanatory, scoped comments will significantly increase the depth and variety of the program's tree structured Dimensional Design; long sequences of uncommented executable statements will become hierarchically structured sequences of scoped comments explaining the functions of the (now shorter in length but greater in number) sequences of executable statements (see figure 7-8).

So a first attempt to measure the 'quality' of a program's explanatory refinement hierarchy might be to measure the lengths of sequences of executable statements (terminal nodes). This would be an adequate metric to differentiate figure 7-8a from 7-8b. A more general metric would include sequences of comments (non terminal nodes) as well. As there are many sequences in any given Dimensional Design it was decided to display the results of such sequence length measurements as a histogram. An example of such a histogram is figure 7-9 giving the lengths of sequences in the B2D example program.

In fact many, many different histograms can be computed from a Dimensional Design because, as it is an n-ary tree, there are n degrees of symmetry in the set of possible histograms. Some can measure 'local' features such as the number of subsystems (ROOTS statements) per individual sequence (set, refinement etc). Some histograms can measure 'cumulative' features such as the total number of sequences (sets, refinements etc) which make up one, higher level subsystem, or 'global' features such as the depth (ie distance from the tree's root in a tree walk sense) at which a subsystem is located.

The number of possible histograms is increased if it is decided to differentiate between, say, sequences whose descriptions are enumerated, iterated or abstracted. In ROOTS, these description reducing techniques can be used to define classes of source code statements, sequences, sets and refinements

thus:

STATEMENT	1.Action	1.Terminal ie unrefined leaf
	2.Conditional	2.Nonterminal ie refined leaf
	(ie part of condi-	
	tional symbol)	
SEQUENCE	1.Enumerated	
	2.Repeated	ie iteratively generated
	3.Missing	ie removed by H-reduction
SET	1.Enumerated	
	(Set induction &	
	abstraction not yet	
	implemented in	
	ROOTS)	
REFINEMENT	1.Enumerated	
	2.Recursive	ie inductively generated
	3.Missing	ie removed by H-reduction

The full set of possible 'feature measuring' histograms is large. For the simple (3+2)-ary trees which represent ROOTS programs there are over 175 'local' feature histograms and over 30 'depth' feature histograms, all of which quantify some aspect of a Dimensional Design tree which can be interpreted as one component of overall quality (see figures 7-10 and 7-11). The concept behind 'depth' histograms is that the triple which uniquely identifies the position of any given node in a 3-ary tree shows how many statements must be 'reasoned about' before being able to understand the given statement. For example suppose "A=B+C" is at position (refine=3, sequence=5, set=2) in a Dimensional Design then the programmer must go down through 3 levels of refinement, sort out 2 alternatives and 'do' 5 sequential instructions to understand the context and role of "A=B+C". This metric is related to the Reduction of Enumerative Reasoning (see section 3.4).

To date only a subset of all possible histograms is generated by ROOTS and no extensive trials have been made of their use. However, even from the limited experience gained so far it appears that certain histograms do seem to capture the intuitive feel that visual inspection gives, particularly when combinations of different histograms are interpreted collectively. For example when the local feature Number of Statements per Sequence histogram is taken together with the global feature Refinement Depth per Statement histogram they seem to give an insight as to how well structured is the 'What-How' explanatory hierarchy.

The Number of Statements per Sequence histogram reflects how many statements are needed to refine individual scoped comments. A good management technique is to develop an awareness for the expected distribution of this metric for 'good' programs and then ask for explanations of extreme deviation, eg "why do some comments require an exceptionally large number of state-

ments to implement?". The answer is usually, but not always, that executable instructions have been added without amending the higher levels of the Dimensional Design ie changing the code but not the design. (The author's staff once found the author guilty of this crime by checking the histograms of a program being developed by the whole team. He had 'patched' the program late one night to cure a bug and meant to amend the design later - honestly!).

The Refinement Depth per Statement histogram shows the 'depth' of each statement in the explanatory 'What-How' hierarchy ie how many levels of refinement down does any statement occur. Figures 7-12 and 7-13 show the same program (B2D) with and without an explanatory hierarchy and their respective histograms. (Note in passing how B2D's Dimensional Design has been scaled down giving an overall view of its structure.) The two histograms in combination seem to be a promising technique for spotting poorly maintained programs. If such histograms are recorded regularly throughout the life time of a program and compared, it seems, from limited experience, that the degeneration of quality due to 'normal' maintenance work is reflected in the time sequence of histograms. When work is performed specifically to improve the deteriorated quality then this 'bringing back up to standard' seems to again be reflected in the changing shapes of the histograms which appear to 'animate' Belady's Law of Increasing Unstructuredness (Entropy): "The entropy of a system increases with time unless specific work is executed to maintain or reduce it".[5]

Thus the histograms allow a manager to

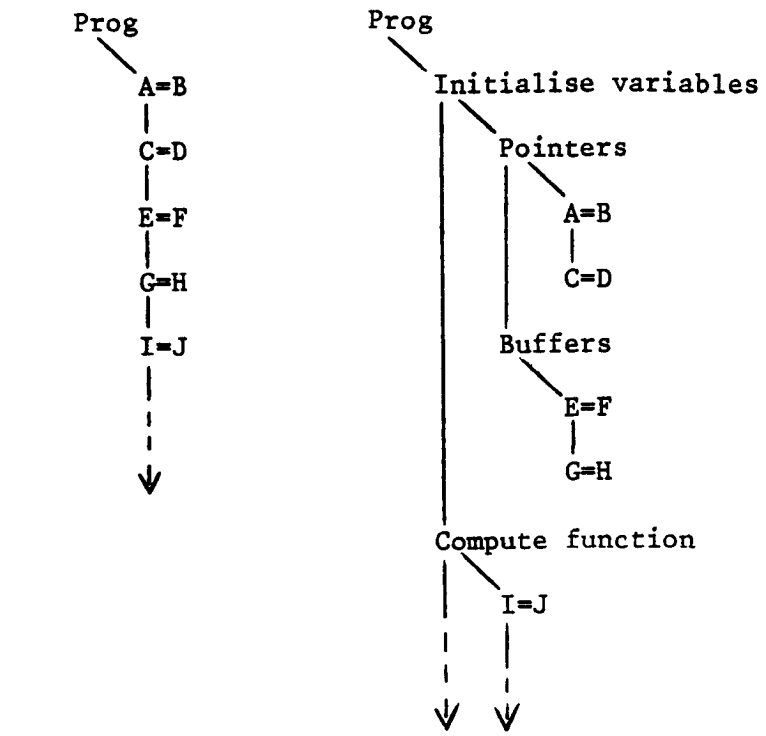
1. question deviations from expected norms
2. watch for unhealthy trends

by somehow abstracting useful properties related to program quality. However, this line of research has not been pursued as far as is desirable and no

substantial claims can be made for it but preliminary experience has shown that a maintenance programmer can gain much more from studying a set of histograms than he can from contemplating the results of calculating 'magic number' complexity measures such as McCabe's Cyclomatic Complexity[49] and the number of possible paths through the program (iteration!) both of which have been implemented in ROOTS, or Halstead's Software Science approach[29] which has not been implemented in ROOTS. This is not too suprising as the set of histograms approach portrays much more information than 'magic numbers' ever can but histograms require 'interpretation' by the programmer which 'magic numbers' superficially do not. The power of the histogramming technique comes from three factors unique to Dimensional Design. One factor is that the histograms analyse a tree. The second factor is that this tree contains the explanatory 'What-How' hierarchy in an explicit form which is amenable to analysis and the third factor is that the histograms are easy and cheap to compute. No conventional programming system has these properties. Still, all that can be said at this stage is that the histogramming technique is a promising avenue for future work.

Histograms are not the only possible quality control techniques which may be used at this stage of Construction. For example, software tools may be used to check for likely portability problems (eg PFORT[69]) or to enforce project standards. ROOTS, for instance, forces all comments to be scoped and explicitly refined into, eventually, executable code. As no original quality control work, other than the invention of the histogramming technique, has been undertaken the prototype ROOTS tool-kit has not yet been stocked with existing quality control tools. Much more work is needed in this vital area, for the customer, designer and maintenance programmer alike all need effective ways to measure the quality of their software.

The quality control techniques discussed above only relate to the quality of the (static) detailed design or source code. Software must also execute correctly and efficiently and so sets of quality control techniques are needed to quantify these dynamic aspects too; but before this is possible the static source code/design must be turned into an executable binary program.



(a) Poor & Flat

(b) Good, varied & deep

Figure 7-8. Program with & without explanatory hierarchy.

```
STATEMENTS PER SEQUENCE

     BIGGEST VAL=       37 SCALED BY=        1 WIDTH=      100

  0       <    1!
 37            1!*************************************
 31            2!*******************************
 35            3!***********************************
  5            4!*****
  5            5!*****
  2            6!**
  1            7!*
  0            8!
  0            9!
  0           10!
  0           11!
  0           12!
  0           13!
  0           14!
  0           15!
  0           16!
  0           17!
  0           18!
  0           19!
  0           20!
  0           21!
  0           22!
  0           23!
  0           24!
  0           25!
  0           26!
  0           27!
  0           28!
  0           29!
  0           30!
  0           31!
  0           32!
  0           33!
  0           34!
  0           35!
  0           36!
  0           37!
  0           38!
  0           39!
  0           40!
  0       >   40!

TOTAL STATEMENTS =        269
```

Figure 7-9. Statements per Sequence Histogram for B2D.

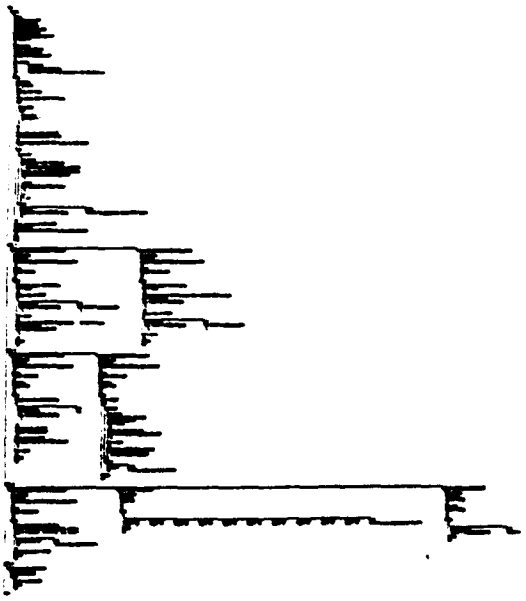
Major Relation	Histogram	Node types		Relation	Interpretation
Seq	$\sum$*Nodes* per individual sequence ie length of sequence	STATEMENTS -action(a) -conditional(c) -terminal(t) -nonterminal(nt) -a and t -a and nt -c and t -c and nt -node (ie a DD) -total	PER	SEQUENCE -enumerated -repeated -missing -total	Overall: reflects quality of descriptive What-How hierarchy for sequential programs. Complexity of refinement ie size of 'how' for a given 'what'.
Set	$\sum$*Nodes* per individual set ie size of set	STATEMENTS -as above	PER	SET -enumerated -total	Overall: reflects decision structure. Width of case statements
Ref	$\sum$*Nodes* per individual refinement	STATEMENTS -as above	PER	REFINEMENT -enumerated -induced -abstracted -total	Overall: reflects complexity of design. Distance between highest levels and lowest detail Complexity of a given 'what' shown by how many stages of refinement it requires to implement

Figure 7-10. ROOTS 'local' feature histograms.

Major Relation	Histogram	Depth	per	Statement	Interpretation
Sequence	Sequence depth per Statement ie min. no. of sequence steps from root to individual statements	SEQ DEPTH	PER	STATEMENT -as fig. 7-10	Amount of enumerative, sequential reasoning required to understand role of any given statement. Overall reflects how well the program uses the What-How descriptive hierarchy technique to reduce enumerative reasoning.
Set	Set depth per Statement ie min. no. of steps from root to individual statement	SET DEPTH	PER	STATEMENT -as fig 7-10	Overall reflects the global 'decision' complexity and how well uses hierarchy to reduce set reasoning. Amount of enumerative set reasoning required to understand role of any given statement

Major Relation	Histogram	Depth	per	Statement	Interpretation
Refinement	Ref depth per statement ie min. no. of refinement steps from root to individual statement	REF DEPTH	PER	STATEMENT -as fig 7-10	Overall reflects the program's functional complexity and how well uses hierarchy to reduce enumerative reasoning. Amount of enumerative refinement reasoning required to understand role of any given statement.

Figure 7-11. ROOTS 'depth' feature histograms.



STATEMENTS PER SEQUENCE

BIGGEST VAL= 30 SCALED BY= 1 WIDTH= 100

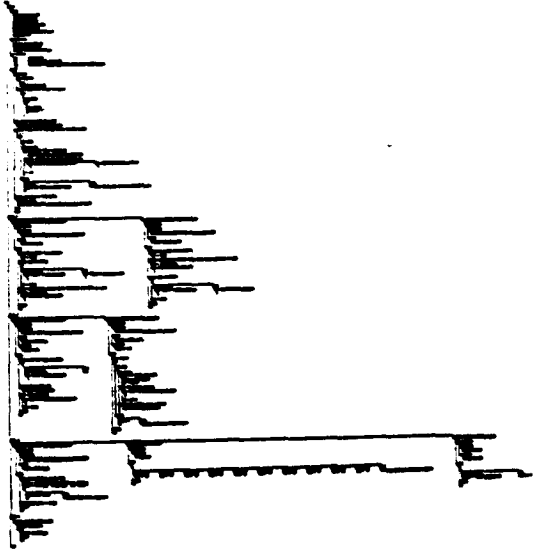
```
 0      <    1!
13            1!*************
 9            2!*********
30            3!******************************
 3            4!***
 0            5!
 2            6!**
 0            7!
 1            8!*
 5            9!*****
 0           10!
 1           11!*
 3           12!***
 2           13!**
 0           14!
 0           15!
 0           16!
 0           17!
 0           18!
 0           19!
 1           20!*
 0           21!
 1           22!*
 0           23!

 0      >    40!
TOTAL STATEMENTS =      314
```

DEPTH OF REFINEMENT PER STATEMENT

BIGGEST VAL= 90 SCALED BY= 1 WIDTH= 100

```
 1      <    1!*
 6           1!******
36           2!************************************
90           3!**********************************************************************************************
65           4!*****************************************************************
89           5!**********************************************************************************************
25           6!*************************
 2           7!**
 0           8!

 0      >    40!
TOTAL REFINEMENTS =      50
TOTAL STATEMENTS =      314
```

Figure 7-12. B2D without What-How hierarchy.



STATEMENTS PER SEQUENCE

BIGGEST VAL= 37 SCALED BY= 1 WIDTH= 100

```
  0      <    1!
 37           1!************************************
 31           2!******************************
 35           3!**********************************
  5           4!*****
  5           5!*****
  2           6!**
  1           7!*
  0           8!
  0           9!
  0          10!

  0      >   40!
TOTAL STATEMENTS =      269
```

DEPTH OF REFINEMENT PER STATEMENT

BIGGEST VAL= 80 SCALED BY= 1 WIDTH= 100

```
  1      <    1!*
  6           1!******
 20           2!********************
 35           3!***********************************
 54           4!******************************************************
 80           5!********************************************************************************
 54           6!******************************************************
 16           7!****************
  3           8!***
  0           9!
  0          10!

  0      >   40!
TOTAL REFINEMENTS =      94
TOTAL STATEMENTS =      269
```

Figure 7-13. B2D with What-How hierarchy.

7.5.4. Production of the Binary Program

The output from the ROOTS preprocessor is actually standard Fortran. This has to be passed through the conventional Fortran compilation system to produce an executable binary program. ROOTS is a Fortran based system because it has to fit in with the reality of existing programming practice at the Rutherford Laboratory where Fortran programming is standard. The benefits gained from using the Fortran route have far outweighed the disadvantages of developing a new compiler for yet another odd ball programming language.

Options exist within the ROOTS preprocessor to allow the binary to be produced with or without calls to the ROOTS run time monitoring system which has been built to aid rectification, development and testing.

7.6. Testing

Software may be tested at several different stages of its life cycle and for several different reasons. It was argued in section 2.3.6 that testing is only useful if it is undertaken as a scientific experiment performed on a system to test the validity of a hypothesis concerning that system. For example, a program might be tested to experimentally confirm that its performance is in accordance with the designer's predictions. Such testing cannot be exhaustive and so is only valid if it is related to an underlying theoretical model of performance which was derived as part of the detailed design activity. Such testing is a form of Quality Control. In contrast to the Quality Control of the Detailed Design (section 7.5.3) which related to the Static Analysis of the program's design description, such testing relates to the Dynamic run time behaviour of the program. Run time behaviour is a much less visible or tangible aspect of a program than its design and is consequently more difficult to understand, analyse and document. The ROOTS tool kit contains novel and sophisticated run time monitoring facilities which help the programmer to perform a

Dynamic Analysis of his program's execution time behaviour.

7.6.1. Run Time Monitor

It is a weakness of many programming language designs and implementations that they provide no facilities for debugging and performance measurement. Discovering which sections of a program consume the most resources can be exceedingly laborious and hence is rarely undertaken.

The ROOTS tool-kit provides its users with facilities to

1. measure the CPU time and I/O time taken by nominated sections and statements,
2. generate a hierarchically structured execution trace of nominated sections and statements and
3. monitor control and data flow in the program.

At the language level this is achieved by including a formal monitoring specification section at the start of each ROOTS program which defines the type and scale of monitoring required, and then simply tagging those sections and statements throughout the program for which monitoring is required (see section 10.2.6).

It is a further weakness of most other conventional run time monitoring systems that the results produced are in the form of a sequential line printer listing; relating these results to the program itself can be exceedingly tedious. The ROOTS Monitor is closely integrated with the Dimensional Design support tools so that by feeding the output from the Monitor along with the machine readable Detailed Design (source code) into the Dimensional Design drawing program (see figure 7-1) a version of the Detailed Design can be produced which is augmented by performance data and run time snapshots (see example B2D in sections 10.2.4-5). This run time information is thereby placed at the

exact points in the design to which it relates.

The run time execution traces generated by the Monitor are also presented to the programmer as Dimensional Designs (see B2D in section 10.2.6). There are several mechanisms provided to control the style and volume of tracing. All of these Monitor facilities are more fully described in [22] - see section 10.1.2.

7.6.2. Performance

Compiling the complete Detailed Design rather than just its source code terminal statements provides ROOTS with a major advantage over conventional performance monitors because via ROOTS the designer can monitor his program throughout a whole range of levels of abstraction rather than at some fixed, small grain level such as the individual statement or subroutine or the machine's 'next binary machine code instruction' register! Typically, a programmer looking for, say, a CPU bottleneck can monitor at a high level of abstraction to see where the majority of the time is spent. Armed with this knowledge the more detailed levels of the major consumer can now be monitored - a process which can continue recursively until the offending section is identified. At each stage little overhead is introduced because only a few key points are being monitored. Even then the ROOTS Monitor measures its own overheads for the programmer to take into account (see section 10.2.4).

All monitoring activity is optional, that is the ROOTS compiler can be instructed to ignore the monitoring instructions and produce a 'production' version. This means that monitoring instructions can be left in the source code permanently which is a great advantage to the maintenance programmer as he can now 'inherit' the monitoring work of the original designers and developers. Also it enables periodic monitoring to be undertaken to check that, say, performance has not become degraded due to development activities.

and it allows 'useful' monitoring experiments to be saved so that if, say, a problem recurs, the previously successful monitoring instructions can be just turned on rather than laboriously reinvented. The ability to retain useful sets of monitoring instructions is another advantage of the ROOTS tool kit not commonly found in today's programming systems.

ROOTS currently measures CPU time, I/O time, execution frequency and depth of recursion. Many other parameters could be measured. As well as locating bottlenecks this sort of information is useful in giving the designer feedback about the overall dynamic behaviour of his program; performance measurement is an animation tool.

7.6.3. Animation

The ROOTS Monitor generates another 'animation' tool which can be directly displayed on the Static Detailed Design. Snapshots of the values of selected variables can be taken during execution, held in circular buffers and then displayed at appropriate positions on the detailed design so the programmer can again obtain a 'feel' for the behaviour of his program (see B2D in section 10.2.3). Naur[57] used snapshots to prove the correctness of programs. He commented that "Our proof problem is one of relating a static description of a result to a dynamic description of a way to obtain the result. Basically there are two ways of bringing the two descriptions closer together, either we may try to make the static description more dynamic ... or we may try to make the dynamic description more static. Of these the second is clearly preferable because we have far more experience in manipulating static descriptions". Mastering the interdependence of the static description and dynamic execution is the essence of programming skill. Mills[51] feels that the benefits of structured programming are that "these ideas are powerful tools in mentally connecting the static text of a program with the dynamic process it invokes in

execution".

The ROOTS Monitor analyses the dynamic flow of control by two mechanisms. The first is the traditional circular buffer containing the identities of the last few statements executed (see B2D in section 10.2.6.2). This technique is considerably overshadowed by the Trace facility in the Monitor which is a novel feature.

The ROOTS Monitor produces a trace of the execution using *exactly* the same notation (Dimensional Design) as the Static detailed design. (see B2D in section 10.2.6.1). The Detailed Design of a program is a Generative description of the statements to be executed. A ROOTS Monitor trace represents the Enumerated description ie the actual statements executed. In doing this the Trace captures the 'animation' of the Generative process. By placing the static description alongside the dynamic description (see figure 7-1) the designer can obtain a very clear (mental and physical) picture of his program's behaviour.

The complete Enumerated Description of a typical program would be a huge Trace, a totally impractical document to produce. The ROOTS Monitor therefore contains several features to control the volume of Trace output (see [22] - section 10.1.2). Three such novel features are:

1. Depth Control
2. Pruning
3. Repetition Filtering

The Trace, being an Enumerative Dimensional Design, is a tree whose root is the program's name and whose leaves are the executed statements. Non-terminal nodes are the elements of the 'What-How' hierarchy. ROOTS exploits these non-terminal nodes (which are missing from conventional programs) by allowing the user to specify to what depth should the tree of the Trace be recorded

for any given section of the program. Thus the hierarchical structure of the design allows Tracing to occur at varying levels of detail/abstraction.

A further refinement of the depth concept allows the user to specify that the Trace tree be 'grown' down to a certain depth, but upon successful completion of a given section, that the tree be 'pruned back' to a lesser depth. This feature means that the overall progress of the computation can be recorded at a high (low) level of abstraction (detail) whilst if an error occurs, causing premature termination, the tree will not be pruned in that region and a much greater amount of detail is reported around the erroneous region.

A third volume controlling feature is the Repetition Filter. By this means the programmer can specify a filtering function which will only allow Tracing of a subset of all the iterations around a loop. This considerably reduces the Trace volume whilst adequately 'animating' the iteration. For example, the programmer might chose to trace the first five iterations (to check the algorithmic logic ie induction) and then only every 1000^{th} thereafter to monitor the computational progress (to watch, say, convergence occur). [Note: in the current ROOTS Monitor Repetition Filters only work on iterations, but clearly the principle could be generalised to handle all forms of Induction.] Most of the other monitoring features in ROOTS, such as snapshots, also carry 'iteration number' information which is of great value to the programmer in picturing run-time behaviour.

The example program, B2D, was deliberately chosen to illustrate the 'animation' of recursion and iteration so that their similarities and differences could be clearly visualised (see section 10.2.6.1). Both iteration and recursion generate and execute a repeated pattern of instructions. "The recursive procedure, however, forced upon us the recognition of the difference between its (static) text and its (dynamic) execution - its 'incarnation' as it has been

called. The procedure text is one thing; the set of local variables it operates on this time is quite another matter".[15] In a recursive procedure both instructions and data storage are inductively generated. An iterative procedure usually only generates instructions. As the generation of instructions is trivially easy in both cases (manipulation of the program counter) whilst the data storage management can be a complex business it is clear why recursion is sometimes erroneously labelled as less 'efficient' than iteration.

The ROOTS Monitor was designed to allow programmers to obtain a picture of the way their programs behave. Such 'animation' should lead to a better understanding of their programs, should confirm that the designer's intentions have been realised in practice and should help to produce correct programs.

7.6.4. Correctness

The most that the ROOTS Monitor can do is to trap obvious errors such as division by zero and check ASSERTIONs at run time for "program testing can be used to show the presence of bugs, but never to show their absence!".[15]

7.6.5. Practical Experience

The ROOTS Monitor has not yet been used in anger on a major project but it has produced encouraging results with small programs. It was built as an experiment to evaluate some novel ideas and so far techniques exploiting the explicit structure of the Detailed Design to produce a hierarchical Trace with depth and pruning controls and repetition filters have proved successful.

The results of monitoring the performance of even small programs have been illuminating and suprisingly counter-intuitive at times, emphasising the inadequacy of modern practice in not undertaking performance prediction work during the design stage and the inadequate feedback from modern

language implementations on the resource costs of their features (see section 10.2.4).

The hierarchically structured Trace has proved successful at animating the concepts of H-reduction/expansion, integration/selection and induction/deduction by relating the Generative to the Enumerated Descriptions.

Finally the ROOTS tool kit, with its unification of the static and dynamic representational techniques, eases the programmer's burden by improving the Conceptual Integrity[9] (see section 2.3.3) of his tools and strengthens his ability to tackle the major problem in Software Engineering, the maintenance task or more accurately, Rectification and Development.

7.7. Rectification & Development

Maintenance is a euphemism for the detection and correction of *design* errors. Rectification is the detection and correction of errors in the Dêtailed Design, ie the program does not conform to its Specification. Development takes place when the program is operating correctly but its users wish either to change its functional specification or have its efficiency improved.

7.7.1. Rectification

Rectification, ideally, should not be necessary. However even with correctness proofs, run-time checkable assertions and bounded loops, some faults are still likely to occur in practice. The Dimensional Design/ROOTS system helps programmers to detect errors in three ways.

Firstly, the ROOTS Monitor animation features allow the programmer to see how the intended behaviour of the program differs from the actual behaviour (conventional debugging). All the features of Traces, Snapshots, Control Flow and Performance Measurement (see section 7.6) are at his

disposal.

Secondly, the ROOTS system ensures that the programmer is trying to find an error in the actual design from which the erroneous program was constructed by always producing a new design drawing with each compilation. (How many man-hours have been wasted by looking at the (slightly!) out of date version of the listing, which by Murphy's Law, does not actually contain the sought after error!). In addition, the retention of the complete, hierarchically structured Detailed Design, in neat machine drawn form, is of great assistance at debugging time as the programmer needs to appreciate the overall design context when looking for bugs, wandering up and down the logic hierarchy until finally descending upon the faulty part. Having the whole design available helps the maintenance programmer to see how the section under scrutiny interacts with sections written long ago (by people he probably never even met) and which he must now learn and understand.

Thirdly, visual inspection or Static Analysis can hint at those parts of the program which are potential harbingers of bugs, namely those areas which appear scrappy as a result of poor design, bad maintenance or failure to amend design and code together.

The correction of errors relies solely on the skill of the programmer. He must realise that every time he corrects an error by changing the source code he is in fact changing the design. There are a whole host of problems associated with fault reporting, change control, the versions problem, getting feedback about errors to the original designers etc for which ROOTS offers no assistance.

7.7.2. Development

The development of a correctly operating system occurs when its users

wish to change its functional capability or improve its efficiency. This implies possible changes to the program's Specification and Overall Design as well as the Detailed Design. Many years ago programmers were encouraged to write 'efficient' programs. This usually involved 'tricky' practices. Then the pendulum swung in favour of correctness. However, writes Dijkstra[15] "my refusal to regard efficiency considerations as the programmer's prime concern is not meant to imply that I disregard them. On the contrary, efficiency considerations are recognised as one of the main incentives to modify a logically correct program. My point, however, is that we can only afford to optimise (whatever that may be) provided that the program remains sufficiently manageable".

ROOTS offers the programmer a good Monitor with which the success of various attempts to optimise a program can be measured. It is unfortunately all too common to see programmers 'optimising' their programs without any attempt at either analysing the performance of the existing version or measuring the change in performance of the newer versions. Knuth[42] comments that "There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimisation is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgements about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses

fail. After working with such tools for seven years, I've become convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it has been specifically turned off".

When development activity causes changes to the design of a program then ROOTS can help keep the documentation up to date and Dimensional Design can help the developer to see the implications of proposed changes. Dijkstra[15] comments that "As long as programs are regarded as linear strings of basic symbols of a programming language and, accordingly, program modification is treated as text manipulation on that level, then each program modification must be understood in the universe of all programs (right or wrong!) that can be written in that programming language. No wonder that program modification is then a most risky operation! The basic symbol is too small and meaningless a unit in terms of which to describe this ... To rephrase the same argument: with the birth of Algol 60, syntax was discovered as a powerful means for expressing structure in a program text. (Syntax became so glorified that many workers in the field identified Computing Science with Syntactic Analysis!). It was slightly overlooked, however, that by expressing structure by syntax, this structure is only given very indirectly, ie to be derived by means of a parsing algorithm to be applied to a linear sequence of symbols. This hurts if we realise that many a program modification leaves large portions of the structure unaffected, so that after painful re-parsing of the modified text the same structure re-emerges! I have a strong feeling that the adequacy of context-free methods for the representation of structure has long been grossly overestimated". Dijkstra then goes on to give a beautiful example of the inadequacy of the indirect, syntactic approach in which two compensating errors in

a program (missing closing quote symbol followed by missing opening quote symbol) render the program syntactically correct but nonsensical. [Note that the visual two dimensional technique of Dimensional Design renders such errors immediately obvious.]

Both Rectification and Development boil down to redesign work, closing the loops in the program production process (see figure 7-1).

7.8. Summary

The dependence of Rectification and Development on a redesign cycle completes the description of the practical use of Dimensional Design with a suitable reminder that the quality of software is critically dependent on the quality of its design.

The theory of Dimensional Design, advanced in Chapters 3-6, has been shown in this chapter to have been used successfully on several real projects. The B2D program and the ROOTS tool kit have been used to illustrate how Dimensional Design is used to produce a working, maintainable program. Figure 7-1 outlined the various stages and tools involved in a program's design, construction and development. Subsequent sections showed how Dimensional Design was used to produce a Detailed Design which was captured, completely, by the ROOTS coding and compilation scheme, to enable automatic design document production, performance measurement and animation/debugging information, all presented in a single, common representational technique.

In this way Dimensional Design/ROOTS has attempted to reduce (but not solve) the Software Crisis' major component, the maintenance problem, by firstly improving the initial design of individual programs and secondly by providing the maintenance programmer with a complete Detailed Design together with a set of software tools to help him rectify and develop this design. How

well has Dimensional Design/ROOTS achieved its stated goals in comparison to existing techniques?

CHAPTER 8. DIMENSIONAL DESIGN: ASSESSMENT

OUTLINE

The small scale software engineering problem has been tackled by developing a theory of Dimensional Design and then putting the theory into practice. The representational technique was the key feature of Dimensional Design. This chapter assesses Dimensional Design by first looking at existing techniques such as high level programming languages (an example of pure textual representation - or are they?), traditional flowcharts (created to perceptually model induction), trees and nested boxes (perceptible 'structure' is now fashionable). Hybrids are introduced. They are textual descriptions augmented by redundant, perceptual recoding of key features. All of these techniques use certain basic elements, their 'vocabularies', which may be quantitatively compared against Dimensional Design. Nested boxes emerge as the most well developed competitor to Dimensional Design so a detailed comparison of the two is undertaken. Finally Dimensional Design is assessed against its stated goals - has it actually solved the small scale software engineering problem?

8. DIMENSIONAL DESIGN: ASSESSMENT

8.1. Introduction

The assessment of the Dimensional Design technique hinges around the question

> "Has Dimensional Design achieved its stated goals; specifically, can it improve the initial quality and tractability of a software design and reduce the subsequent burden of rectification and development?"

This question contains the implication that 'improvement' means improvement over existing techniques. The general state of Software Engineering was reviewed in Chapter 2 but after the decision to concentrate on the problems of producing individual programs (Small Scale Software Engineering) little has been said about existing small scale production techniques. It is therefore necessary to compare Dimensional Design with conventional practice before making an overall assessment.

8.2. Existing Techniques

8.2.1. The Major Types of Representational Techniques

From the very beginning of modern computing in the 1940s there has been a consistent descriptive trend away from the definitive binary machine code program to more abstract and tractable formulations. The development of more humane representations of programs has proceeded in two major directions, the Textual and the Perceptual, driven mainly by the need to restrict the generality of the von Neumann machine so that programmers can only produce intellectually manageable programs (see figure 8-1). Figures 8-2 to 8-6 are examples of the major techniques currently used to express Detailed Designs.

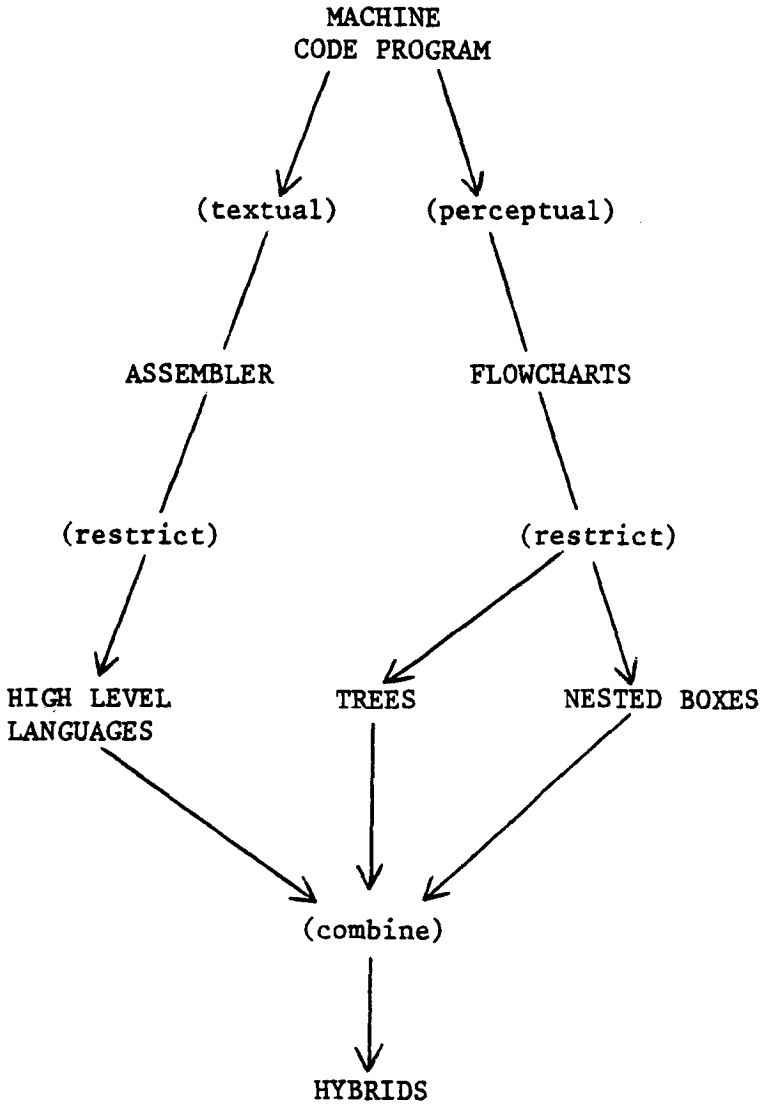


Figure 8-1. Development of Major Representational Techniques.

```
.SUBROUTINEPB2DR(BINARY,PAGE,LINE,NCP)
.CDeclarations.CParametersINTEGERBIN
ARY,Page(widmax,lenmac),LINE,NCP.EC
.CLocalsINTEGERLSTDIG,BIN.EC.CFunctionsINTEGERCHAR
.EC.EC.BEGIN.CConvertallbutleastsigdigit
.IF(BINARY.GE.10).THENBIN=BINARY/10
.CALL(*)PB2DR(BIN,PAGE,LINE,NCP).ELSE.NULL.ENDIF.EC
.CDetatchleastsigdigitLSTDIG=MOD(BINARY,10).EC
.CPrintleastsigdigit.CALL(3)PRINT(CHAR(LST
DIG),PAGE,LINE,NCP)
NCP=NCP+1.EC.RETURN.END.SETSEP
.SUBROUTINEPB2DI(BINARY,PAGE,LINE,NCP).CDeclarations
.CParametersINTEGERBINARY,PAGE(widmax,lenmax),LINE,NCP
.EC.CLocalsINTEGERLSTDIG,J,BIN
.EC.CFunctionsINTEGERCHAR.EC.EC
.BEGINBIN=BINARY.CYCLEJ=1,widmax.TILL(1).DO
.CDetatchJthleastsigdigitLSTDIG=MOD(BIN,10)BIN=BIN/10.EC
.CPrintJthleastsigdigit.CALL(3)PRINT(CHAR(LSTDIG),PA
GE,LINE,NCP)
NCP=NCP-1.EC.UNTIL(onlyleadingzerosleft).IE
.EXITIF(BIN.EQ.0).TOSITU(1).REPEAT.SITU(1).OK.LIMIT
.FAIL(err,'PB2DILCCEGUARDERROR').ENDCY.RETURN.END
```

(a) Pure Text

- no typographical layout.

```
.SUBROUTINE PB2DR(BINARY,PAGE,LINE,NCP)
.C Declarations
    .C Parameters
        INTEGER BINARY,PAGE(widmax,lenmax),LINE,NCP
        .EC
    .C Locals
        INTEGER LSTDIG,BIN
        .EC
    .C Functions
        INTEGER CHAR
        .EC
    .EC
.BEGIN
.C Convert all but least sig digit
    .IF(BINARY.GE.10).THEN
        BIN = BINARY/10
        .CALL(*) PB2DR(BIN,PAGE,LINE,NCP)
    .ELSE
        .NULL
    .ENDIF
    .EC
.C Detatch least sig digit
    LSTDIG = MOD(BINARY,10)
    .EC
.C Print least sig digit
    .CALL(3) PRINT(CHAR(LSTDIG),PAGE,LINE,NCP)
    NCP = NCP+1
    .EC
.RETURN
.END
```

```
.SUBROUTINE PB2DI(BINARY,PAGE,LINE,NCP)
.C Declarations
    .C Parameters
        INTEGER BINARY,PAGE(widmax,lenmax),LINE,NCP
        .EC
    .C Locals
        INTEGER LSTDIG,J,BIN
        .EC
    .C Functions
        INTEGER CHAR
        .EC
    .EC
.BEGIN
BIN = BINARY
.CYCLE J=1,widmax .TILL(1) .DO
    .C Detatch Jth least sig digit
        LSTDIG = MOD(BIN,10)
        BIN = BIN/10
        .EC
    .C Print Jth least sig digit
        .CALL(3) PRINT(CHAR(LSTDIG),PAGE,LINE,NCP)
        NCP = NCP-1
        .EC
    .UNTIL(only leading zeros left).IE
        .EXITIF(BIN.EQ.0).TOSITU(1)
.REPEAT
    .SITU(1)
        .OK
    .LIMIT
        .FAIL(err,'PB2DI LOOP GUARD ERROR')
.ENDCY
.RETURN
.END
```

(b) Hybrid

- indentation, spacing etc.

Figure 8-2. Text (High Level Language).

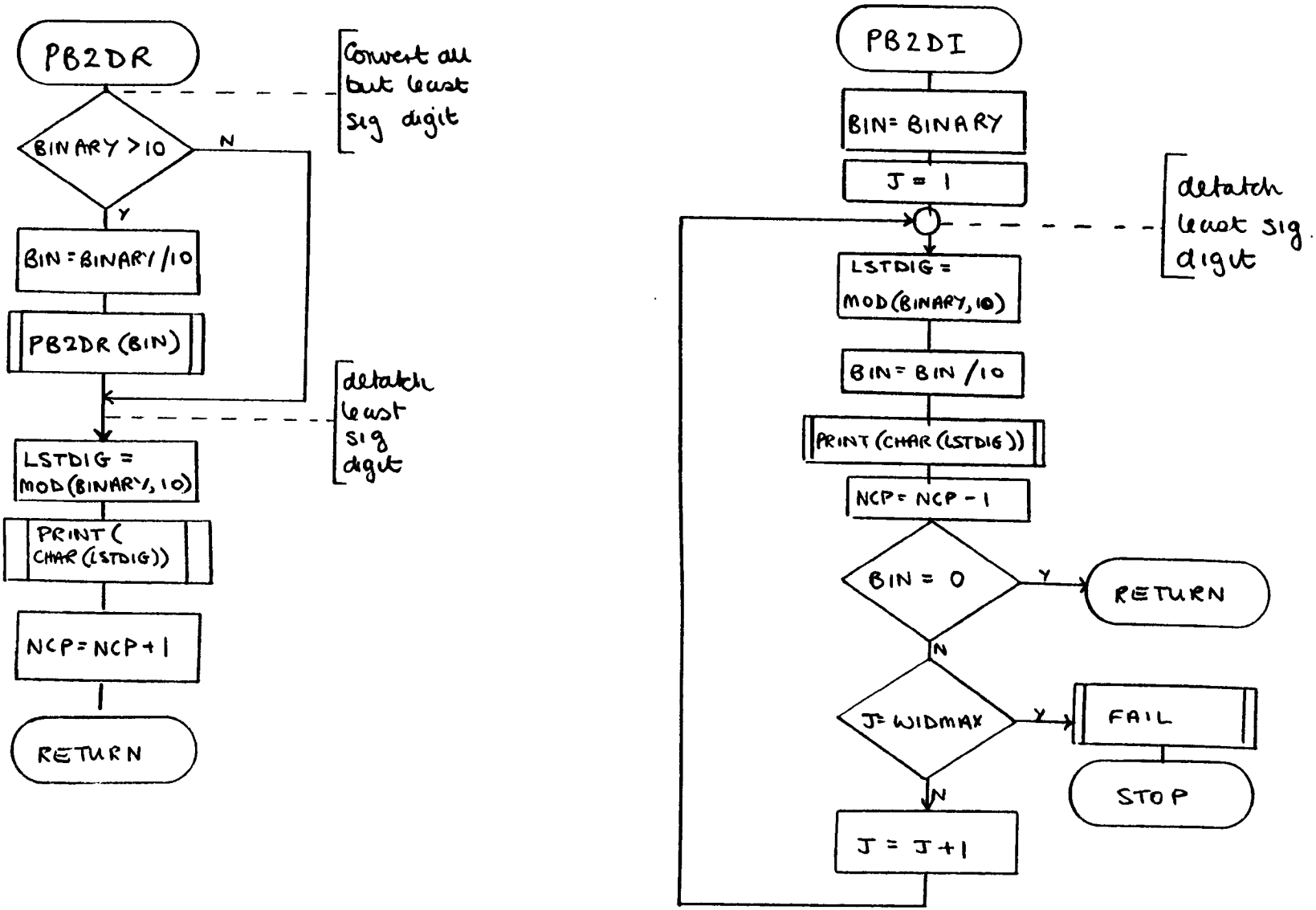


Figure 8-3. ANSI Flowcharts.

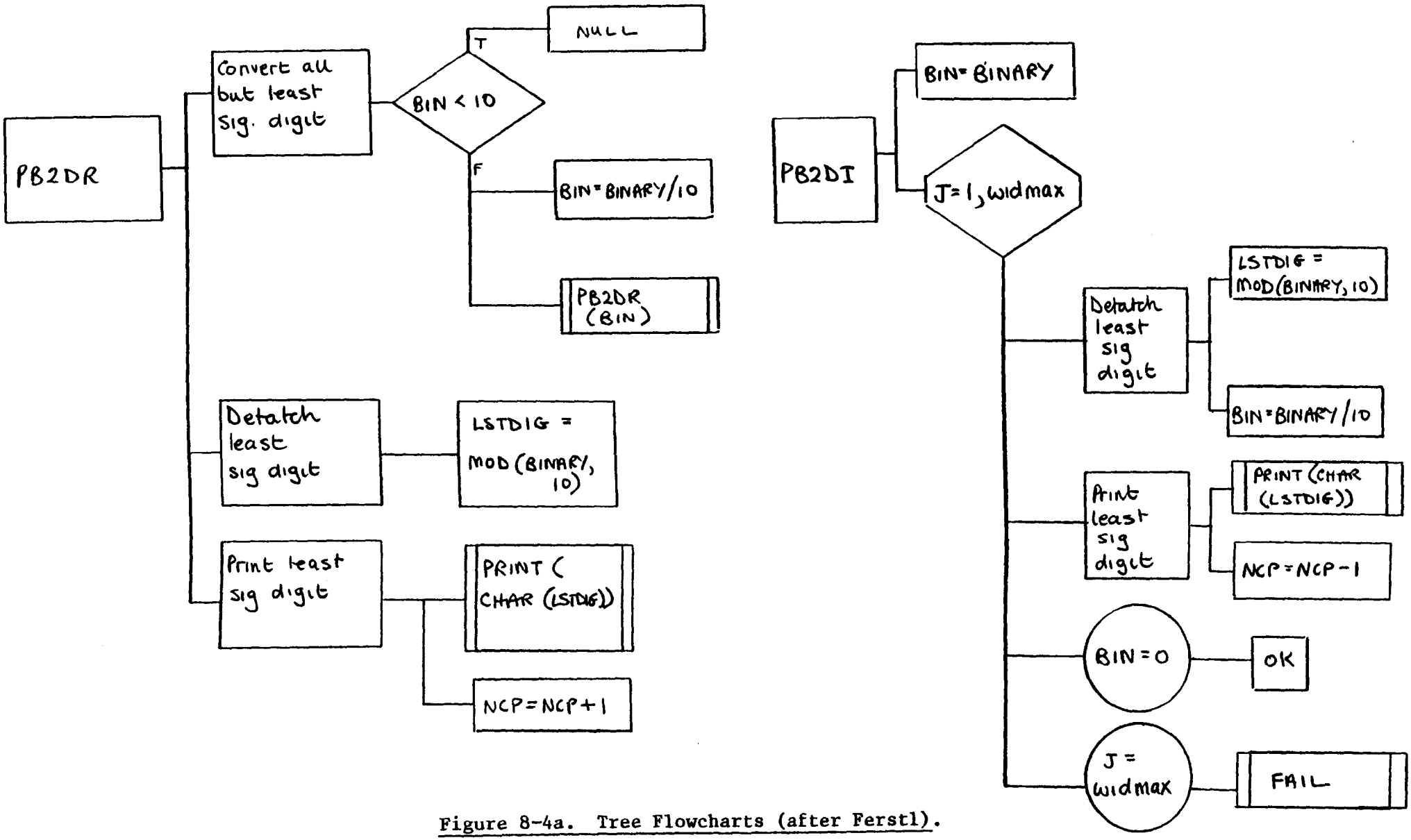


Figure 8-4a. Tree Flowcharts (after Ferstl).

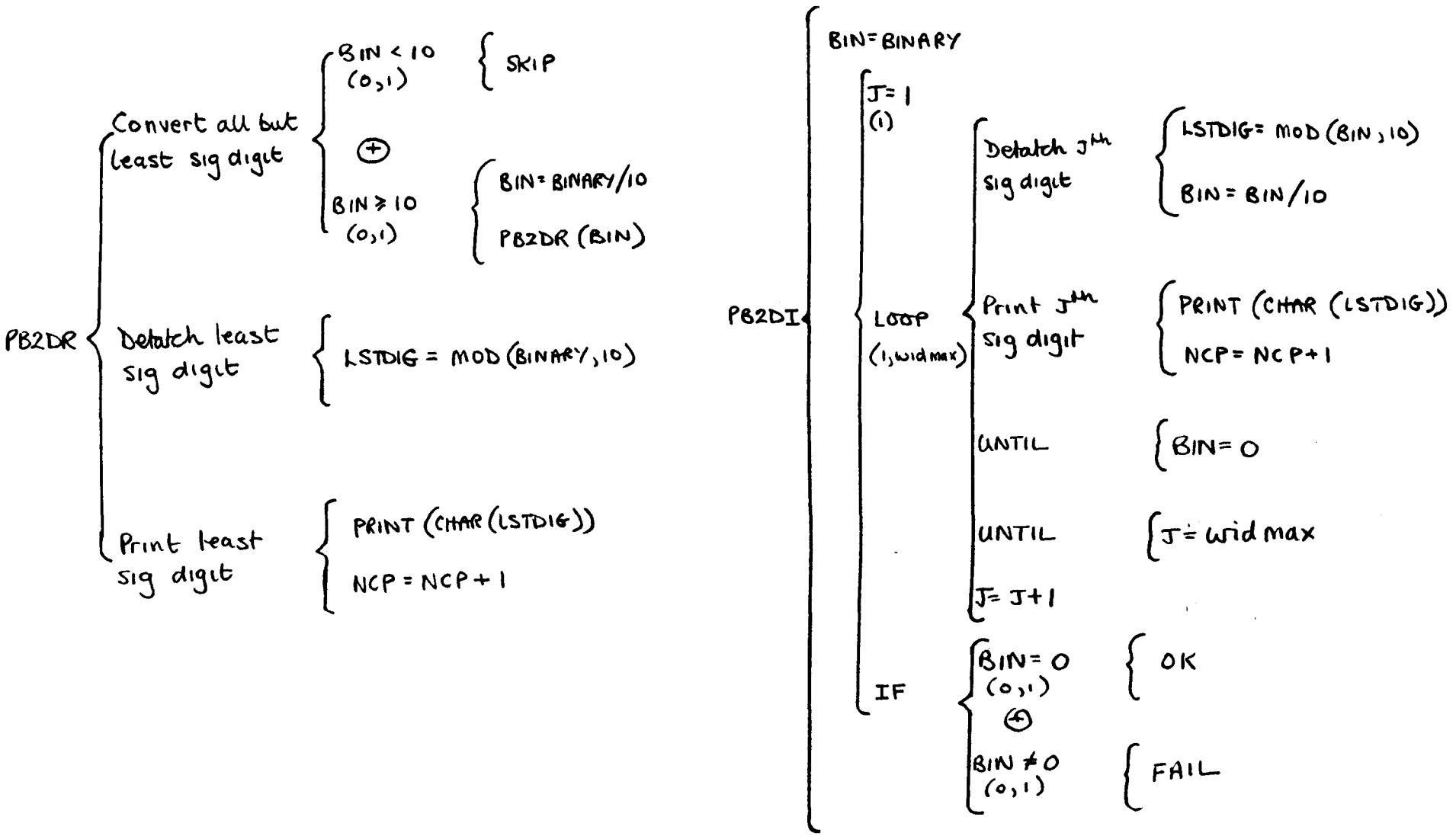


Figure 8-4b. Tree Hybrid (Warnier Diagram).

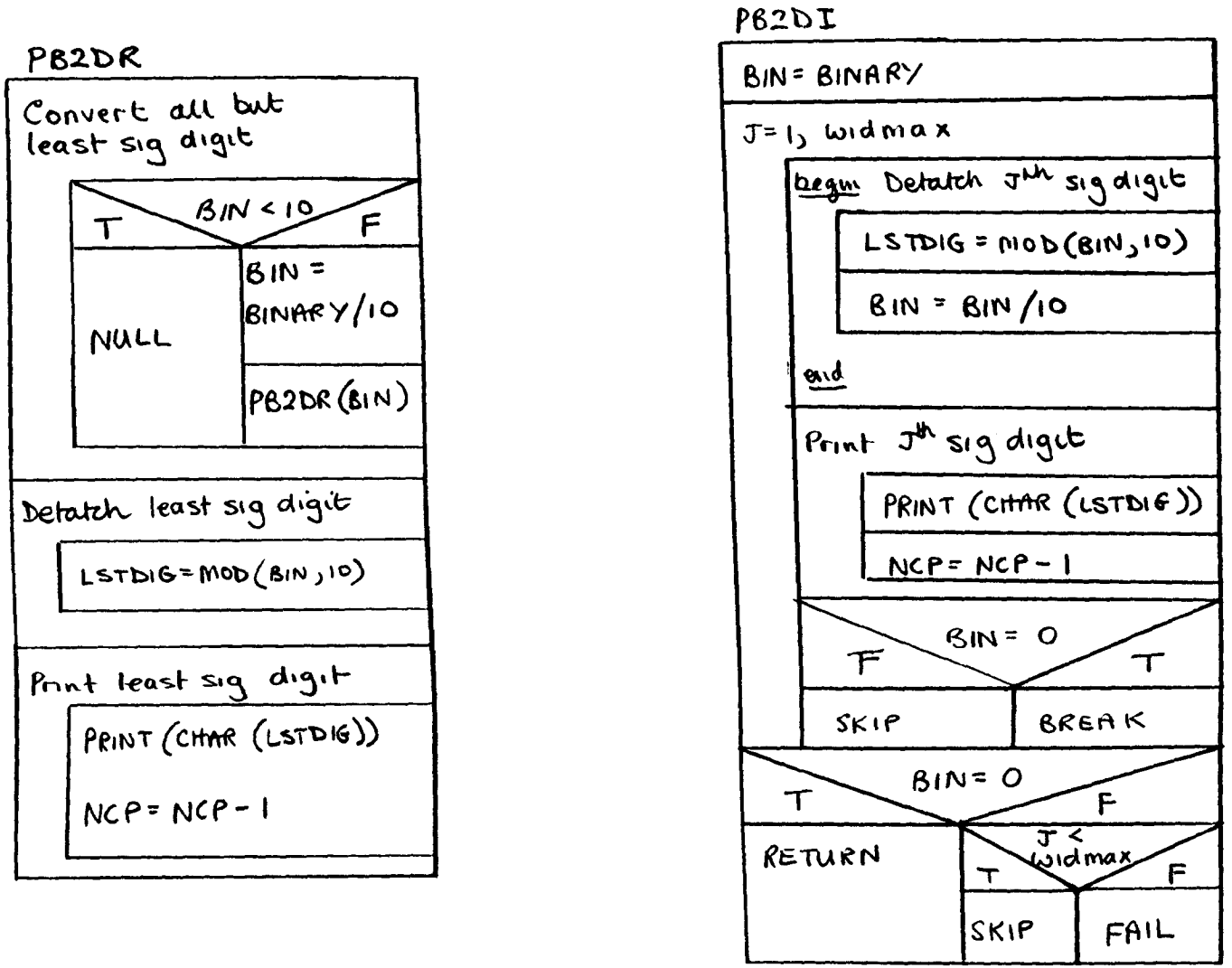


Figure 8-5. Nested Boxes (Nassi-Shneiderman).

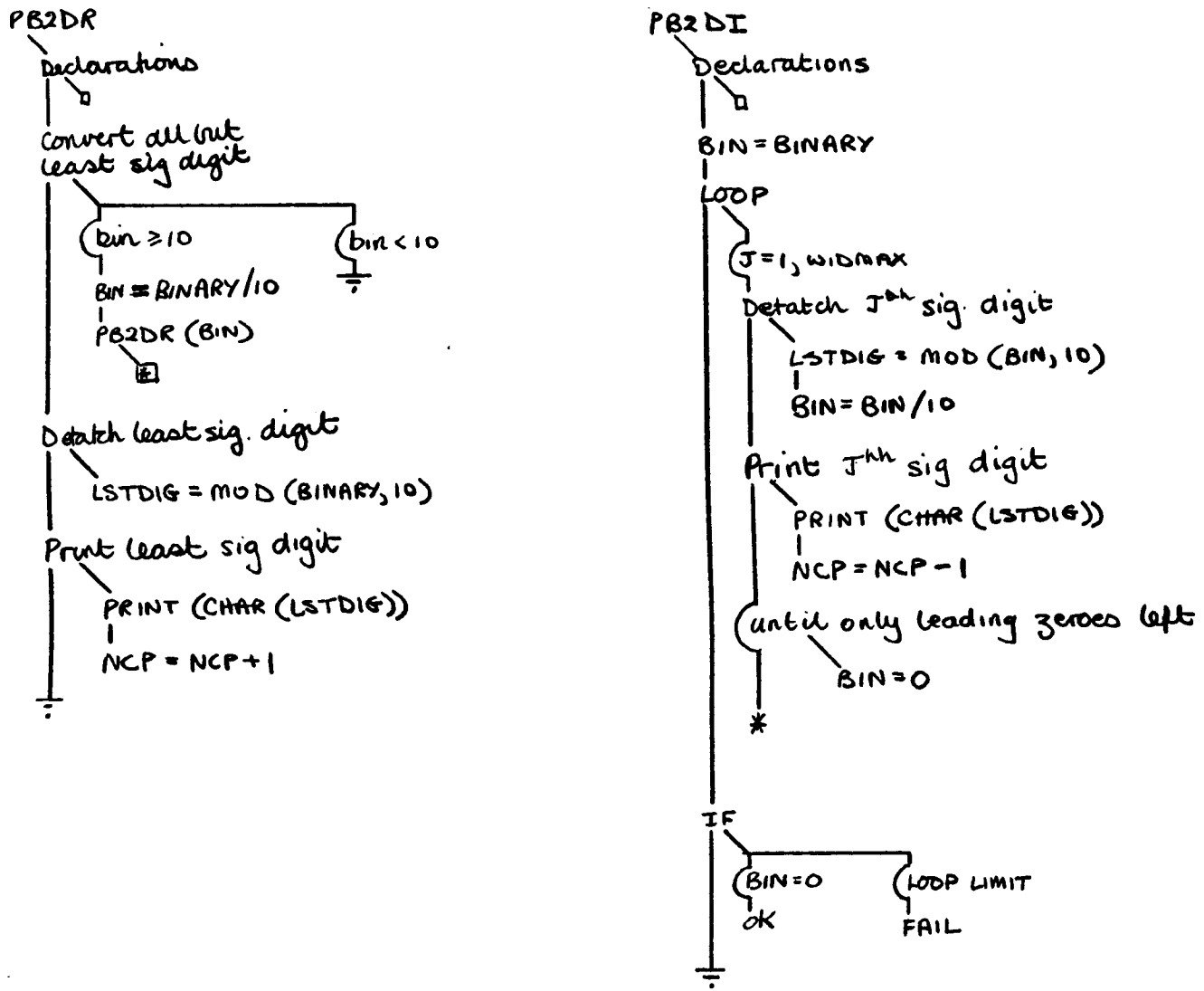


Figure 8-6. Hybrid (Dimensional Design).

8.2.2. Textual v Perceptual Descriptions

Of the examples shown in figures 8-2 to 8-6 four out of the five are 'diagrams' or perceptual descriptions and only one, the high level language, is a textual description. The difference between a textual and a perceptual description is exemplified by comparing a map, which presents its information as an analogue encoding into 2-D space, with an annotated list of grid references.

Commenting on the origins of these two techniques by contrasting 'sentences' (textual) and 'formulae' (perceptual) MacLennan[47] observes "In origin, sentences are written encodings of spoken communication. Since the sounds of speech evolve sequentially in time, the framework in which sentences are constructed is naturally one dimensional, that one dimension corresponding to time. Therefore when sentences are read, the time sequence in which the words are presented to the eye approximates that in which the sounds would have reached the ear. This one dimension is used as the framework in which the information bearing units of the sentence, the words, are arranged. In most languages the written form of a word is a sequence of letters, which does not in itself contribute to the meaning of the sentence; rather, it is a phonetic encoding of the spoken form of the word.

Formulas have a completely different origin since they were, from the beginning, a written language. Formulas are of a class with diagrams, and ultimately derive from pictures whereas *sentences are meant to be read*, either silently or aloud, *formulas are meant to be studied*, or shown to another. They are not readable, per se. In particular, there is only one correct order in which to read a sentence; in grasping a formula the eye may rove over the page, possibly visiting one part several times. Because of this, formulas can take advantage of the richer communication possibilities of a two dimensional

layout.

The above discussion can be summarised as follows. All notations can be analysed into primitives and constructors. The application of the constructors to the primitives (according to the syntax rules of the notation) generates all the instances of that notation. For sentences, the basic constructor is the linear arrangement of the primitives, and the primitives are words, whose written forms encode their spoken forms. The constructor of formulas is the two dimensional spatial arrangement of their primitives, and the primitives are symbols, with no phonetic significance, chosen only for their distinctive shape and, possibly, a pictorial suggestion of their meaning."

"It is possible in principle to develop notations that are purely perceptual, like a scale model, but these are of no more practical interest than a purely (textual) notation would be, an unbroken stream of text without any typographical aids to legibility whatsoever - no paragraphs, no capitals etc. *Usable notations always contain both (textual) and perceptual elements*" .[25] Sometimes the two types of descriptions are independently combined, as in figure 8-7a, where the perceptual encodings (circles) describe the inter-relationships between the sets whilst the textual part describes the nature and contents of the individual sets. Sometimes the two types of description are not independent. Often the perceptual component is, strictly speaking, redundant because the information it conveys can be extracted from the textual component. Typographical layout and source code indentation are good examples of this combination which Fitter and Green[25] call "redundant recoding", where the same information is presented both textually and perceptually. A description which is basically a textual description with certain aspects redundantly recoded perceptually will be called a Hybrid.

"Redundant recoding occurs most naturally not so much in 'pure'

diagrammatic notations as in enhancements to essentially linear notations. When one particular type of information is highly relevant to the user's task, it is helpful to present at least the bare bones in a perceptually-coded form, meanwhile presenting the fully-clothed bones in a (textual) form for closer examination".[25] Thus if a programmer wishes to create a Projection (see section 2.4.1) of a certain aspect of a system it would seem likely that a redundantly recoded Hybrid can highlight the projected aspect whilst avoiding too great an overall information loss.

The redundancy in a Hybrid description is cost-effective if the intellectual advantage of a given aspect being immediately obvious (perceivable) is greater than the effort needed to work out the aspect from its implicit representation by a purely textual description. By perceptual encoding "diagrams avoid the problems of 'hark-back' created by the sequential nature of (textual) codes" (eg Murphy's Law says the body of a subroutine is always on a different, unidentified microfiche to those of its callers), "but on the other hand (diagrams) have problems of their own, notably the need to introduce a vocabulary of special symbols (eg flowchart box types) to enhance the small number of available perceptual codes".[25]

Perceptual encoding, therefore, has a limited *Vocabulary*. Illustrated so far have been the techniques of Insideness (Venn Diagram, Nested Boxes), Spatial Location (a map), Connectedness (ANSI Flowchart) and Indentation (source code program). The size of the perceptual vocabulary may be increased by varying the style of lines (dashed, thickened etc) and by colouring. (Note: non-graphic techniques such as sound are not considered here). Even so, the perceptual vocabulary is still small compared to the textual and symbolic vocabularies. Worse still, the Resolution of perceptual coding techniques is usually rather coarse - for example it would be no help at all to indent a source

code listing by millimetre steps!

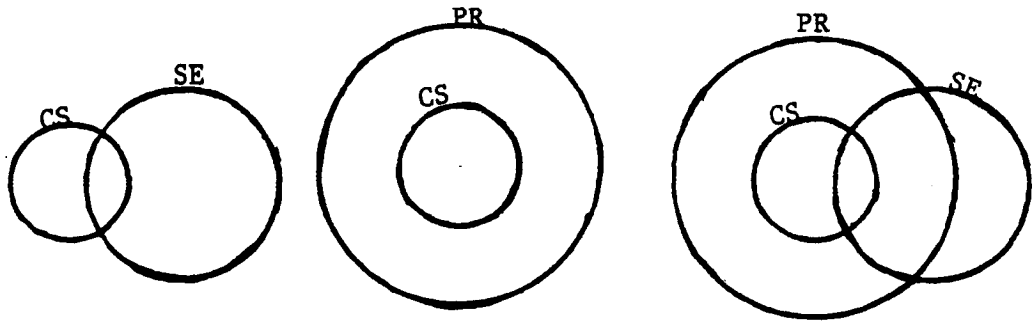
As the whole point of using a diagrammatic notation rather than a purely textual one is to make something perceptually obvious it is clear that the inherent possibilities and limitations of the description's *medium* greatly influence the success in attaining this goal.

Given the limited vocabulary of perceptual encodings and their limited resolutions Fitter and Green[25] suggest five principles to guide the designer of a Hybrid notation:

1. The information that is encoded perceptually rather than textually should be *relevant*.
2. Only important aspects should be perceptually and *redundantly recoded*.
3. Notations should *restrict* the user to comprehensible forms.
4. Notations should *reveal* the underlying structures/processes.
5. Notations should be easily *revisable* and manipulatable.

To these principles could be added that "different programs should be as perceptually different as possible"[25] and that, by inference, a well (badly)-structured program should *be seen to be* well (badly)-structured.

The above guidelines, together with the concepts of textual and perceptual encodings, vocabulary size and resolution provide a good background against which to examine the descriptive techniques currently used in programming.



(a) Perceptual - Venn Diagram

$CS \cap SE \neq \{\}$ $\quad CS \subset PR \quad$ $PR \cap SE \neq \{\}$

where CS = Computer Scientists
PR = Programmers
SE = Software Engineers

(b) Textual - Set Theoretic

Figure 8-7. Textual and Perceptual Encodings Independent.

8.2.3. High Level Programming Languages

8.2.3.1. Abstraction

Programming languages are the most common and well developed form of program representation. Their success is probably based on two factors. Firstly they are highly linear and symbolic, characteristics which match the sequential, symbol manipulating von Neumann machine. Secondly they have a small, crude vocabulary to match the crude input and output devices prevalent throughout all computer installations.

With compilers to map programming languages directly into binary programs, the high level language implicitly contains almost complete information about all aspects of a program. However much of this information is too implicit and cannot be extracted without massive intellectual effort or further machine processing (eg global data flow analysis).

It is very easy to build restrictions into high level languages and modern examples tend to constrain the programmer to intellectually manageable control constructs, although there are still many anomalies such as the lack of fully inductive features to describe all objects (eg data structures, case statements) and the lack of artificial hierarchy creation facilities such as the Scoped Comment.

8.2.3.2. Representation

High level languages, represented by pure textual strings, have only one explicit relation, that of character sequence. This is based on the neatly matching concepts of the time sequence of spoken, natural language and the execution sequence of machine code instructions.

The vocabulary of textual descriptions is small, often only 64 characters, due to the primitive input/output devices of modern computers. The

vocabulary is effectively extended by representing objects by sequences of characters analogous to words in natural language. The resolution of this extended vocabulary is not always adequate and many errors are due to failures to resolve the differences between compound symbols eg 'SPAGHETTI' and 'SPAGHETT1'. Problems of resolution are increased when the same name is used in different parts of the program to mean different things ie context dependence.

High level programming languages are not really languages; they are cosmetically disguised formulae. For example problems arise in some languages because subscripted names have to be linearly encoded in a manner identical to the application of a function to its arguments (see Sentences v Formulae above). The major fault with programming languages is that they are not designed to be constantly re-read and studied but only read once - and that the reader is assumed to be a compiler not a human being! Programming languages have suffered from the fact that "Any pidgin algebra can be dressed up as Pidgin English to please the generals"[43] (Note: 'algebra' not 'language'). It is therefore left to the programmer to try to make his programs as readable (which really should be *studiable*) as possible ie to make the best of a bad job.

8.2.3.3. Manipulation

High level language programs are easy to create, edit and store both by hand and machine. (However machine editing tends to reduce Quality - see indentation, section 8.2.4). Languages generated by context free grammars are well suited to machine analysis for the production of executable code and certain documentation.[50] It is interesting to note that even the compilers for which the languages are designed(!) tend to hold their internal representations as Hybrids, creating parse trees and symbol tables etc.

High level languages are amenable to quality control analysis but lamentably no significant progress has been made in this direction. High level languages are not particularly good at representing performance data, debugging information or animations, although [71] has made good use of colour by making the colours of parts of the program text a function of their execution frequency.

High level languages are good for the initial creation of small sections of programs but due to their very poor explanatory ability they are poor for large design exercises (see 'Columbus' system below).

8.2.3.4. Axiomatisation

Due to their symbolic or formula basis, high level languages are highly suitable for formal treatment. For example, Chomsky's work on syntax, Hoare and Strachey on semantics and the growing body of knowledge on specification, verification and proof techniques shows this aspect of programming languages to be a strength.

8.2.4. The Origins of Hybrids

Pure text, without typographical layout ie as input to a compiler, is unreadable. There has therefore grown up a set of conventions and techniques to try to alleviate this problem by producing textual hybrids. Conventions such as one statement per line and indentation are common. One statement per line in nonsensical with block structured or hierarchical languages and indentation is not a practical technique because of the 'increasing entropy' effect in which, due to repeated editing, the accuracy of the indentation degenerates till it is meaningless. A solution to this problem is Prettyprinting - using a software tool to reformat a textual program. Indentation schemes usually suffer from a lack of resolution. The narrow width of the lineprinter forces

'reorientation' of lower levels of the description to appear at the same physical indentation as higher levels due to the limited number of available, resolvable indentation positions being much less than the number of logical levels in the description. This need not be the case. Irons reports in [43] that "I have put together a program which uses some of these features and which has a standard output which prints the program in an indented manner. If it runs off the right hand end of the page, it produces another page to go on the right, and so forth. While certainly there are some situations that occur when it would be a bit awkward to make the paper go around the room, I have found that in practice, by and large, it is true that this is a very profitable way of operating".

In a large, indented program the problem arises of 'matching the columns' ie of trying to establish, by eye amidst a lot of 'white space', exactly which statements are aligned at any given indentation level. The common practice is for the programmer to draw, by hand, lines connecting statements at the same level. This process can be automated[10,13,53] and the generalisations of such Hybrids, when combined with traditional flowcharting experience, will lead naturally to Nested Boxes and Dimensional Designs (see figure 8-1).

8.2.5. Flowcharts

8.2.5.1. Abstraction

Traditional flowcharts are the product of the perceptual route to a more abstract and tractable representation for machine code programs (see figure 8-1). They are, historically, the first of the popular representations[59] and have been remarkably successful because they make clearly visible the flow of control through a program and thus enable the programmer to imagine the animation of his program's execution. Flowcharts perceptually represent flow of

control, points of decision and points of action on the state. Such 'control flow' flowcharts have been the primary creative vehicle for Detailed Design for many years. It is possible to produce flowcharts at varying levels of abstraction but no totally satisfactory way has been found of relating and/or combining them although promising attempts were made.[41]

Control flowcharts were developed from a numerical analysis/computational background. With the advent of 'data processing' an adaptation was introduced, the 'systems flowchart' which showed the flow of data (or files of data) through a suite of programs. No satisfactory way of combining these two projections has been found.

8.2.5.2. Representation

Flowcharts are two dimensional diagrams whose vocabulary includes 'connectedness' (implemented by directed arcs representing flow) and special box symbols portraying the concepts of decision, action, subroutine call, input/output operation and file media. The resolution of this 'box' vocabulary is generally good thanks to its small size and reasonably successful standardisation. Resolution of the flow lines is good locally but can be poor globally if the flow lines are a) long, b) split (by connector boxes) across page boundaries or c) closely interwoven into complex patterns - the so called spaghetti logic of bad programs. .

Although flowcharts are perceptual diagrams with text enclosed in box symbols they are not usually true Hybrids (as defined above) because a) non-executable information, such as declarations, is missing and b) it is often impossible to squeeze enough text into the Procrustean fixed sized boxes, so abbreviations and abstractions are used. The true Hybrid alternative requires variable sized boxes which are then more difficult to draw by hand and 'size of box' and rectangular aspect ratio then become meaningless and confusing

perceptual encodings. The disadvantages of conventional flowcharts are many and well documented[41,34,85,56] and will not be catalogued here.

8.2.5.3. Manipulation

One of the attractions of flowcharts is that they are easy to draw during the 'back of the envelope' creative stage of design. However, flowcharts of even moderate size are extremely difficult to lay out and tedious to draw neatly and these facts, combined with the extreme difficulty and poor results of drawing and editing by machine,[1] have been a major cause of their decline in popularity. The drawing problem combined with the inability to directly model high level language control constructs[85] has led to the virtual disappearance of conventional flowcharts from modern software engineering methods.

Although on the wane flowcharts still have many advantages, including their expository value which aids design, debugging and understanding. They are also good at representing performance data and resource consumption. By changing the familiar rectangular 'action box' into a 2-D projection of a 3-D cuboid whose volume was proportional to resource consumption Stockenberg and van Dam[74] were able to elegantly represent this vital aspect of a program's dynamic behaviour.

8.2.5.4. Axiomatisation

"A set of block or flow diagrams is a two-dimensional programming language ... As far as is known, a systematic theory of this language does not exist" wrote Bohm and Jacopini[8] in a paper famous for its expository use of flowcharts in promulgating the use of 'provable' control constructs. Another paper,[59] less famous than it deserves and predating reference,[26] used flowcharts extensively to prove programs correct by the method of inductive reasoning; indeed flowcharts were introduced specifically to model induction.

Although it is fashionable to decry the use of flowcharts they have been and continue to be of immense practical benefit to practising programmers. Their inadequacies have led to the search for more restricted forms which are more in line with modern theory. Restriction to 'structured programming' control structures has led to the development of Nested Boxes and Tree representations. It remains to be seen whether these newer techniques will achieve comparable widespread popularity and practical usage.

8.2.6. Trees

8.2.6.1. Abstraction

In order to improve over conventional flowcharts, modern representational techniques seek to show two additional, but independent, concepts to control flow. The first is the complete restriction of control flow to 'provable constructs' and the second is the ability to include an artificial, explanatory hierarchy on top of the executable instructions. The restriction on control flow allows the use of the convention that iteration 'loop back' arcs and selection 'communal join' arcs may be omitted thereby allowing a program's executable instructions to be expressed as a hierarchy; tree layouts and nested boxes are therefore natural candidates, being essentially isomorphs.

8.2.6.2. Representation

The advantage of being able to represent 'structured control flow' constructs and explanatory hierarchies is gained only at the price of losing the beautiful perceptual encoding of dynamic control flow. In tree diagram techniques such as figure 8-4a[24] and figure 8-4b[77] the *flow of* control now follows a complex 'tree-walk' path back and along the arcs of the tree. This is because it is now the *static construction* of the control constructs which is perceptually encoded rather than the *dynamic flow* aspect. However, the gains from good

control structure heavily outweigh the cost of the initial training needed to become familiar with tree-walking rather than tracing the flow of control.

Ferstl's technique (figure 8-4a), being closer to the original flowchart concept, still suffers from the drawback of having to fit text into boxes or coping with variable sized boxes. Warnier's technique (figure 8-4b), being more of a hybrid, dispenses with boxes (as did some 'conventional' flowchart systems[65]) but runs into worse problems because it cannot now perceptually encode iteration and selection. These have to be indicated textually by the '(1,n)' iteration range and the '⊕' mutual exclusion operator.[77] There is also no explicit iteration termination mechanism indication so a simple While loop cannot be explicitly represented (see figure 8-4b, "UNTIL").

8.2.6.3. Manipulation

Both types of tree diagram are difficult to draw freehand because they both grow, from the root, vertically upwards as well as downwards and hence lack the simplicity of the one quadrant drawing schemes (see section 5.3).

There are no known reports in the literature which show that either of these techniques has been automated or used on major practical projects. Automated drawing should not be difficult to implement as tree layout algorithms exist.

Similarly there are no known reports of their use to portray performance, debugging or animation information. However, G. Jackson[36] has suggested a technique for teaching recursion which uses (hand drawn) tree diagrams as animations of recursive algorithms expressed originally as Nested Box diagrams. This is similar to, but slightly less elegant than, the Dimensional Design animation trace of the execution of the B2D example program (see section 10.2.6.1).

Clearly both Tree techniques are an improvement over the conventional flowchart in creating and explaining well-structured programs. They have obvious similarities to Dimensional Design and will be directly contrasted in section 8.3.

8.2.6.4. Axiomatisation

There is no known report of the axiomatisation of these Tree diagrams; however, it is clear that, as they are essentially Hybrids or projections of 'structured programs', the latter's theory applies.

8.2.7. Nested Boxes

8.2.7.1. Abstraction

The Nested Boxes type of diagram (figure 8-5) is essentially equivalent to the Tree type and is used to restrict the control flow to 'structured' concepts. The nested boxes technique does not yet seem to have developed sufficiently to be able to represent an artificial, expository hierarchy but this can easily be incorporated by an extension to Nassi-Shniederman's[56] Begin-End representation (see figure 8-5).

8.2.7.2. Representation

Nested box diagrams have been used for many years, especially in connection with the block structure of Algol 60. The Contour Block models of Johnston and Organick[38,61] describe both the static structure and the execution's animation.

The use of nested boxes to describe well-structured programs is reported in the well known paper by Nassi and Shniederman[56] but this was predated by several papers: Anderson[2] in particular describes a pure hybrid, nested box technique with full machine assistance including 2-D input and source code

production facilities.

The basic vocabulary of the nested box diagram consists of several annotated rectangular symbols together with three spatial arrangements ie horizontal abutment (selection), vertical abutment (sequence) and enclosure or 'insideness' (hierarchy). The Nassi-Shniederman technique[56] cannot represent

1. artificial hierarchy,
2. arbitrary loop termination criteria,
3. recursion. [Note: Nassi and Shniederman claim in [56] that "Recursion has a trivial representation". In fact they mean by this that an 'action' box just contains the name of a procedure which might or might not be recursively invoked. There is no explicit representation to distinguish between recursive and non-recursive calls; they rely on the implicit textual mechanism ie the reader!].

Lindsey[46] improves the Nassi-Shneiderman technique by adding an explicit iteration box symbol which is pictorially suggestive of iteration and incorporates a loop termination criterion sub-box, but then he provides a directed arc mechanism for jumping out of his new loop construct!

Frei[27] reports on the building of an interactive Nested Box editor based on a colour graphics display terminal. This uses colour to help distinguish the 'insideness' aspect of the diagrams. Frei's version of Nested Boxes cannot express any hierarchy other than control structure and the sizes of his diagrams are restricted by the system. This restriction on size appears in other work with Nested Boxes. It is claimed that 'understandable' diagrams must fit onto a single page; it is more likely that large Nested Box diagrams are very difficult to study because the large number of lines forming the many boxes tends to swamp the text and the level of insideness of any given box becomes difficult to resolve.

8.2.7.3. Manipulation

Nested box diagrams are not particularly easy to draw freehand. Nested selection statements are especially difficult to draw because of their reducing size, asymmetric construction and triangular space for selection criteria. It would thus seem the natural step to provide a computer-aided nested box drawing system as Frei proposes. However, it is interesting to note that, in line with the experience with Dimensional Design on real projects, J. Witt[81] found, from his experience on real projects, with a Nested Boxes interactive editor, that "Although this interactive program proved to be highly successful in impressing visitors, it succeeded only moderately with the programmers. Programmers found it much easier to draw their algorithms in Nassi-Shneiderman form on a sheet of paper and then to enter the program line by line using the standard file editor ... Normally the programmers would then run the COLUMBUS file (the linear encoding of the Nassi-Shneiderman diagram) through the program that produces the Nassi-Shneiderman charts on the line printer and check the printed output against their original hand drawn charts ... As soon as they felt satisfied ... they would then have a language dependent preprocessor transform the COLUMBUS form of the program into the pure target language. After possibly having removed syntactical errors, programmers would cease to look at the compilation listing from the compiler and do their debugging work exclusively by using the Nassi-Shneiderman chart".

J. Witt's COLUMBUS system,[81] a pure Hybrid technique, is therefore identical to the Dimensional Design/ROOTS philosophy - only the representational technique being different, denying the benefits of an artificial, explanatory hierarchy to Witt's programmers. The COLUMBUS system appears to exhibit the undesirable property that the maximum width of a machine drawn Nassi-Shneiderman chart is the width of a single sheet of line-printer paper. This

means that as successive boxes are nested inside each other the width of each new box is progressively reduced. This has the very undesirable effect of causing the terminal statements in the innermost boxes to be cramped into very small boxes indeed, less than 10 characters wide, which reduces their legibility considerably and must presumably force an arbitrary restriction on the 'amount' of logic in any given diagram which is not related to the structure of the program being represented.

J. Witt uses the COLUMBUS machine readable representation of programs as the feedstock for a cross-reference listing tool but it appears that no one has used the Nested Box diagrams for any proof, performance or debugging tools, although Nassi and Shneiderman[56] recognise that the animation of Nassi-Shneiderman charts would be very similar to the Algol 60 Contour Model of [38,61] which is the only known use of Nested Boxes for animation purposes.

8.2.7.4. Axiomatisation

There is no known report on the axiomatisation of Nested Box diagrams; however, it is clear that as they are, in theory, projections and, in practice, Hybrids of 'structured' high level language programs that the latter's theory applies.

8.2.8. Postscript on Hybrids

The widespread popularity of traditional flowcharts over machine code languages was based on their superior ability to represent flow of control. With the development of increasingly abstract concepts, such as block structuring, data structures and data typing, which the traditional flowchart could not portray, and aggravated by inadequate input/output devices, the high level language gained its current supremacy. The movement towards hierarchically structured, formally based programming abstractions has made feasible the

development of Hybrid representational techniques which enchance the definitive, implicit descriptive powers of the textual high level language by redundant, perceptual encodings of such major aspects as control structure and descriptive hierarchies. Automated Hybrid programming methods have already been successfully used on real projects and it is likely that their use will spread as the improvements in the cost, performance and capabilities of input/output devices continues to widen their potential user population.

As with any new area of development, Hybrid program representation is currently undergoing an initial exploratory phase during which many rival candidates are proposed and are being submitted to the evolutionary process of natural selection. Dimensional Design is one of this set of Hybrid competitors. How does it rate against its rival?

8.3. Comparison: Dimensional Design v Existing Techniques

8.3.1. Introduction

The only true indication that Dimensional Design surpasses its rivals would be for it to be put into widespread use and see if it won popular support. This evaluation technique is not feasible within the realms of this thesis (but a start has been made - see section 7.1). A smaller version of the global trial would be to instigate carefully monitored, psychologically based field trials to measure programmer productivity and project success rate; again not an economically viable experiment. So, rather than generate an intuitive list of unsupportable advantages, a simpler, more direct, quantitative approach will be adopted.

8.3.2. Basic Vocabulary

The vocabularies of all the representations reviewed in this chapter are made up of three classes:

1. Plain text

2. Special symbols

3. Spatial orientation

In theory each of the representational techniques may use the same set of special symbols and the same plain text and text itself is really just a string of well known symbols. The major differences in technique come from the varying usage of spatial orientation over a 2-D surface. Figure 8-8 shows that Dimensional Design has the largest spatial vocabulary and therefore, in principle, the largest overall vocabulary - a major factor in expressive power.

Given the many acknowledged deficiencies of traditional flowcharts and considering high level languages to be just special cases of Hybrids (in which perceptual redundant encoding is limited to typographical layout techniques) the comparison will now be focussed on the two Hybrid techniques which already claim successful practical field trials - Nested Boxes[81] and Dimensional Design ie the two techniques with the richest spatial vocabularies.

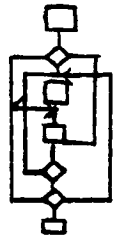
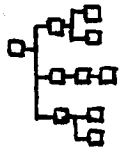
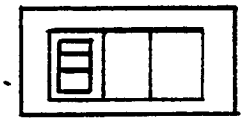
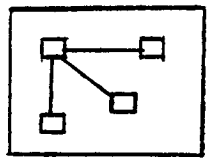
Technique	Example	Size	Basic Spatial Vocabulary
High level language	parbeg a:=3,b:=4 parend;	0	Textual succession is unrelated to any programming abstraction
Flowchart		1	1) Connection
Tree		2	1) Horizontal connection 2) Vertical connection
Nested Box		3	1) Horizontal abutment 2) Vertical abutment 3) Insideness
Dimensional Design		4	1) Horizontal connection 2) Vertical connection 3) Insideness 4) Diagonal connection

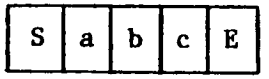
Figure 8-8. Basic Spatial Vocabulary.

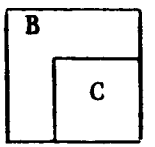
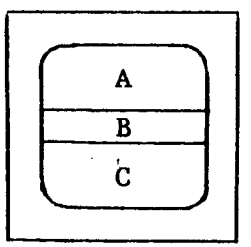
8.4. Comparison: Dimensional Design v Nested Boxes

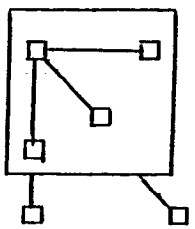
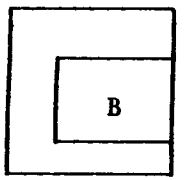
8.4.1. Spatial Vocabulary

The major difference between these two techniques is in the interconnection of symbols by either abutment (Nested Boxes) or edges (Dimensional Design). The disadvantage of abutment is that it tends to constrain symbols to be rectangular. Edge connection places no such constraints on symbol shape. A further advantage of using edges to connect symbols is that the edges may be *marked*, thus allowing a richer overall vocabulary (see figure 8-9). The use of three types of edge (horizontal, vertical, diagonal) also gives Dimensional Design its advantage in basic spatial vocabulary (the abutment of hexagons is unlikely to prove popular).

Both techniques use insideness or enclosure. Dimensional Design uses a simple enclosure (inside or not) vocabulary but Nested Boxes use enclosure in a more complex way: the exact position within the enclosing box, specifically which side is common to both inner and outer boxes, increases the Nested Box's spatial vocabulary (see figure 8-9) but requires greater resolution capability on the part of the reader.

ABSTRACTION	DIMENSIONAL DESIGN	NESTED BOX
Enumeration	S — a — b — c — E —‖· Start: implied by root End : implied by leaf or explicit —‖·	 Start: implied by left (top, outer) box End : implied by right (bottom, inner) box
Selection	Conditional symbol	 Special compound symbol
H-reduction	Definition: → marked edge showing start of definition of subsystem which can be H-reduced Reduction: —□ special symbol □ showing missing, H-reduced, finite subsystem	NOT AVAILABLE

ABSTRACTION	DIMENSIONAL DESIGN	NESTED BOX
Iteration	S—a—b—c—E—*	
	Start: implied by root	while B do C;
	End: —* Special symbol * for missing, infinite, iteratively generated subtree (see Axioms in Chap 6)	Special compound symbol using enclosure 'common side' differentiation
	Termination Criterion: —⌣— Conditional symbol	Termination Criterion : 'B'. Nassi-Shneiderman cannot handle more complex loops but Lindsay suggests following representation for N½ times (Dahl) loop:-
	Dahl loop: loop A B C * Needs nothing special to represent it.	Dahl loop: 

ABSTRACTION	DIMENSIONAL DESIGN	NESTED BOX
Recursion	Definition: → marked edge showing start of recursive defn. of subtree Reduction: —[*] Special symbol [*] for missing, recursively generated, infinite subtree	NOT AVAILABLE
Bracketing	 enclosing 'cuboid'	 Special use of enclosure 'common side' differentiation mechanism. begin B end

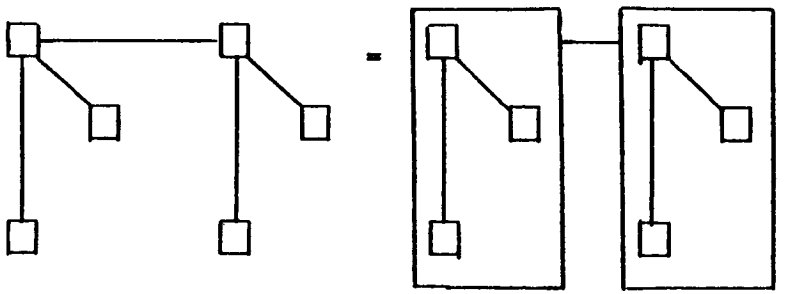
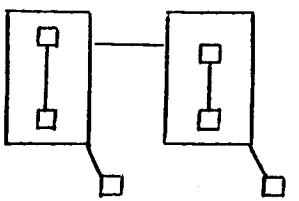
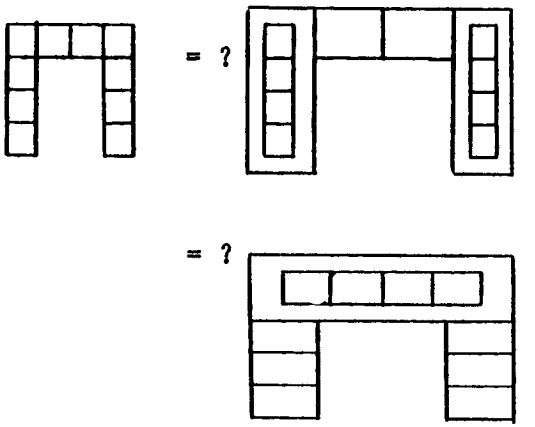
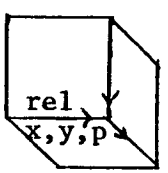
ABSTRACTION	DIMENSIONAL DESIGN	NESTED BOX
Precedence	 because Relations can have precedence defined. In above example \> | > — . not 	 NEVER DISCUSSED in Nested Box papers but is implicitly assumed in compound constructs such as case statements
Connection Labelling	 rel x,y,p where x = artificial/real y = enum/generative p = precedence	NEVER DISCUSSED in Nested Box papers

Figure 8-9. Vocabularies.

8.4.2. Symbol Vocabulary

Both symbol vocabularies have been kept small and simple. Nested Boxes suffers rather from the meaningless wide variation in box sizes.

8.4.3. Descriptive Technique

The spatial and symbol vocabularies are used to illustrate two distinct concepts, concepts too often confused (see Chapter 3) namely:

1. the abstractions required to be represented (control flow, parallelism etc),
2. the techniques used to reduce the size of the description itself (integration, induction etc).

Figure 8-9 compares the description-reducing features of both techniques. It can be concluded from figure 8-9 that the edge connection scheme has many advantages over the abutment scheme. Primarily edge connection allows the expression of all the description reducing techniques in their full generality for all three directions of edge connection. The Nested Box scheme precludes this generality and fails to express perceptually several vital descriptive techniques such as recursion and the distinction between enumerative and generative relations ie abutments.

8.4.4. Extended Vocabulary

Both techniques could extend their vocabularies by

1. increasing the number of symbols and box shapes,
2. increasing the number of relations per quadrant to >3 (DD),
3. going to a full 4 quadrant diagram, giving ≥ 12 relations (DD),
4. varying the line styles,

5. colouring text, boxes, edges etc,

6. varying the size of text, symbols etc,

7. varying the lengths of lines, boxes, edges etc,

but such additional complications are unlikely to improve the representation of the basic description reducing techniques. They are more likely to be of advantage in representing special features such as performance information.

8.4.5. Resolution

The resolution of features in the Nested Box technique is poorer than in Dimensional Design because

1. the 'common side' spatial enclosure technique used to make special compound symbols is not too clear in non-trivial diagrams,

2. non-trivial diagrams are cluttered. Text is obscured by the multitude of lines making up the boxes,

3. variable sized boxes cause text to be 'crushed' into small boxes and 'lost' within large boxes. Box sizes are often more a function of neighbouring boxes rather than the volume of text they hold.

The resolution of Dimensional Design features is better because:

1. the three edge orientations (horizontal, diagonal, vertical) are easily differentiated independently of diagram size.

2. one box type representing simple bracketing by enclosure is easier to resolve than the multiplicity of 'common side' constructions which confuse bracketing with generative techniques like induction.

3. the enclosing boxes are infrequently used (often not at all) and so the Dimensional Design tree is much less cluttered than a nested box equivalent.

4. the text is not constrained to fit into boxes, and boxes whose size is determined by its neighbours at that. Dimensional Design text subsystems are independent of each other and may therefore be laid out better than in nested boxes.

5. there are no arbitrary limitations of the size of a Dimensional Design. Large Nested Box diagrams suffer dreadfully from the variable box size problem.

8.4.6. Abstractions

The conventional Nassi-Shneiderman nested box diagram can only represent two of the normal programming abstractions, set (of alternatives) and sequence (of execution), as against the three, set, sequence and refinement, of Dimensional Design. Even if Nassi-Shneiderman diagrams were extended to include an artificial hierarchy mechanism the Dimensional Design technique would still be preferable because all three abstractions may be represented without the use of enclosing boxes thereby producing a simple, uncluttered diagram.

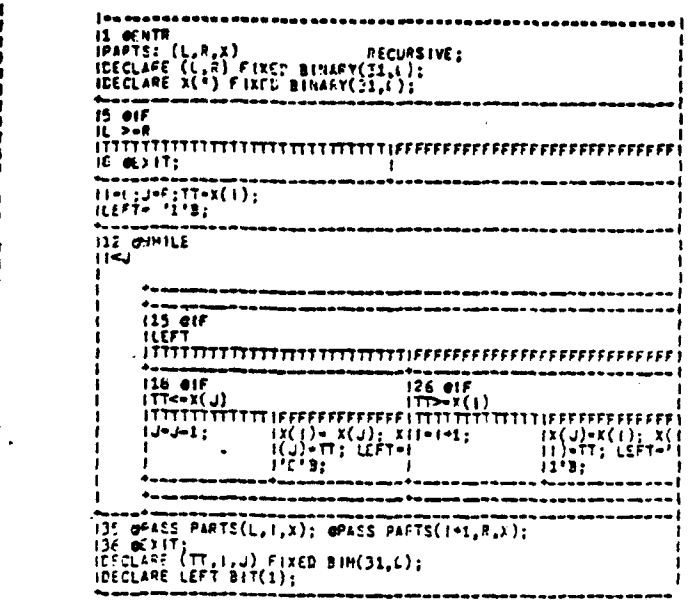
Both techniques can use enclosing boxes to allow the expression of any number of relations (see Chapter 5) but the advantage will always be with Dimensional Design because of its greater ability to represent the full range of enumerative and generative descriptive techniques. For example, Dimensional Design may be used down to the individual character level ie text, normally represented as strings, may be represented as Dimensional Designs allowing, say, the use of induction to shorten a description and refinement to allow the body of a function to be expanded in an arithmetic expression which applies the function. Such fine grain textual description is impossible with Nested Boxes in general.

It must be said that (deliberately) neither technique can describe non hierarchical abstractions explicitly.

8.4.7. Drawing Algorithms

A major advantage of Dimensional Design is its simple, efficient drawing algorithm (see section 5.3). By contrast, algorithms for drawing Nested Boxes are more complex because of the 'context-dependent' box size/matching problem in which the drawing *of* each box is not an independent operation but can only be achieved with considerable knowledge of its neighbours. This manifests itself in the actual diagrams as the unfortunate mismatch between the size of a box and the amount of text it contains (see figure 8-10).

Figure 8: PL/I example (Quicksort) in Nassi/Shneiderman form



(taken from reference 81.)

Figure 8-10. Context dependent box size matching problem of nested boxes.

8.4.8. Summary

The richer spatial vocabulary, the edge interconnection technique and the better resolution give Dimensional Design the edge over Nested Boxes because

1. Dimensional Design can represent the full set of enumerative and generative descriptive techniques for all relations equally.
2. Dimensional Designs are less cluttered and so easier to study.
3. Dimensional Design has the better drawing and layout algorithm.
4. Dimensional Design has the better textual subsystem layout. It can be applied down to the character level.
5. Dimensional Design is amenable to analysis for Quality Control purposes.
6. Dimensional Design is suitable for animation and execution traces. Nested Boxes are unsuitable for such large diagrams.

The strengths and weaknesses found in both systems arise because they are both Hybrids and therefore cannot show more information than is implicitly encoded in the source code version of the Detailed Design. Both representational techniques are capable of practical application and although Dimensional Design wins the simple analytical test of descriptive power, only time will tell whether either will gain widespread acceptance.

8.5. Comparison: Dimensional Design v Stated Goals

It is not enough to suggest that Dimensional Design is the best of the examined representational techniques, for such supremacy was not the major goal set way back in Chapter 2's discussion of the Software Crisis. Indeed, a more immediate objective was set in Chapter 3, namely the substantiation of the so-called Central Hypotheses which, briefly restated, contend that

A: Dimensional Design facilitates creative thought and reasoning about programs (QUALITY).

B: Dimensional Design is a practical technique ie Dimensional Designs can be simply drawn and manipulated by both men and machines (TRACTABILITY).

These hypotheses, said Chapter 3, could be substantiated by satisfying four criteria, given here as questions together with their answers.

1. Can Dimensional Design be formally defined?

   YES - see grammar in section 6.1

2. Can Dimensional Designs be manipulated according to well formulated rules?

   YES - see axioms in section 6.2

3. Can Dimensional Designs be simply drawn and manipulated by both men and machines?

   YES - see drawing algorithm in section 5.3

   YES - see software tools in sections 7.4-7

   YES - see practical experience in section 7.1

4. Do Dimensional Designs facilitate human understanding and creation of programs?

   YES - Dimensional Design Hybrids are the best representation for hierarchies and description reducing techniques, see section 8.3

   YES - Hierarchical structuring and description reducing techniques reduce the burden of enumerative reasoning, see section 3.4

It is therefore suggested that the Central Hypotheses have been substantiated. They may be rephrased by saying the Dimensional Design is a good way to represent the *design product* of programming.

The way is now clear to consider the major, overall goal set in Chapter 2 ie to reduce the prohibitively high cost of software development and specifically the disproportionate expense of rectification and development. Section 2.5 focussed the problem down to the creation of individual programs and asked if successful software engineering techniques could be developed to solve this smaller scale problem so as to reduce the workload of the maintenance programmers. The questions posed in section 2.5 can now be answered.

1. Can a better design product be produced initially?

   YES - The Central Hypotheses contend that Dimensional Design can produce better program designs.

2. Can the design be prevented from becoming obsolete through maintenance changes to the program?

   YES - By making the design part of the program, see Chapter 3.

3. Can the maintenance programmer maintain the design as well as the code?

   YES - Proper software tools operating on the combined design plus code Dimensional Design help him do this. Tools like Quality Control help managers to check this is being done, see Chapter 7

4. Can the maintenance workload be reduced?

   YES - Better initial design plus the availability of an up-to-date detailed design plus Quality Control, performance measurement, debugging and animation tools plus improved conceptual integrity all reduce the workload, see Chapter 7

It is therefore suggested that the Dimensional Design/ROOTS approach to program construction reduces the Rectification and Development workload and that the combination of good design techniques with software tools which support the conceptual integrity of the design philosophy makes Dimensional

Design/ROOTS a promising, prototype small scale software engineering facility.

Amidst such euphoria it should be remembered that these conclusions are only tentative as the techniques are not in widespread use and that no quantitative evidence of significant cost savings on real projects has been produced. The techniques themselves are only a subset of a software engineer's full toolkit, helping with only a subset of the known problems.

Dimensional Design, being *only* a Hybrid, can *only* help a programmer to understand and manipulate the detailed design's source code, which is still usually control flow orientated. There is no way to formally check the 'goodness' of the artificial explanatory design hierarchy's textual content as it is purely informal but at least its overall structure can be checked. The current developments in better specification and verification techniques might solve this problem.

Even so, any detailed design still represents the *implementation* of a solution and not the *principle* of solution. For example, the cooling and lubrication systems of automobile engines may be represented as being independent. However, they are not, especially in air cooled engines; they are an example of the Integration of functions (specifically Shanley Integration[42]) to produce a more efficient design. Such integration is currently deliberately avoided in many software design methods because the extra complexity cannot be handled with current techniques, with a subsequent loss in efficiency. Where it is employed, such integration is often implicitly represented and therefore is error-prone. Hierarchical design methods cannot handle such subtleties.

It is therefore worth remarking that merely improving the description or representation of an object does not, and cannot, improve the object itself. There is still much work to be done before the Software Crisis is overcome.

CHAPTER 9. FUTURE WORK

OUTLINE

Previous chapters have proposed a theory of Dimensional Design and reported on its practical application but the small scale software engineering problem remains unsolved. This chapter suggests two sets of ideas for further work. The first set is qualitative in nature, being extensions to the work described in previous chapters. The second set of ideas is based on the premise that the effective solution to the Software Crisis requires mastery of quantitative techniques.

9. FUTURE WORK

9.1. Qualitative Ideas

Dimensional Design has clearly not solved the software crisis, but has it opened up any promising avenues for investigation or posed any vital questions? Further work could be undertaken on

1. an investigation into the benefits of retaining hierarchical structuring in the executable binary program itself (a suggestion due to Prof. Lawson).
2. the design of a '3-dimensional' programming language, containing the full generality of enumerative and generative techniques for all aspects of code and data specification; its implementation by a purpose built compiler and support tools.
3. an investigation into the limitations and inadequacies of the hierarchical approach to software design as highlighted by the Shanley Integration example in section 8.5.
4. the construction of better tools to support software design such as an interactive Dimensional Design editor or the creation of a true data base/projection system (see section 2.4).
5. an investigation into further techniques to increase the vocabulary and resolution of perceptual encoding techniques for Hybrid descriptions by means of say colour, genuine dynamic (as opposed to static) animation and full 3-D displays.
6. the production of a complete, consistent theory of Dimensional Design.

9.2. Quantitative Ideas

The above suggestion are, perhaps, evolutionary rather than revolutionary opportunities. They are all essentially *qualitative* in nature which evokes

memories of Knuth's words (superficially on Structured Programming) that "there has been far too much emphasis on GOTO elimination instead of the really important issues; people have a natural tendency to set up an easily understood quantitative goal like the abolition of jumps, instead of working directly for a qualitative goal like good program structure. In a similar way, many people have set up "zero population growth" as a goal to be achieved, when they really desire living conditions that are much harder to quantify".[42]

Knuth's observation contains a corollary. Humanity is such that if an easily understandable metric *can* be found which captures the essence of the fundamental, qualitative goal (eg 'run the company well' could be replaced by 'make a 30% profit each year') then the closer the metric is to containing the whole of the qualitative goal the greater the likelyhood of successfully achieving the desired goal. (Note: the metric is a means to an end (goal). Any metric is unlikely to capture 100% of the goal and therefore if the means(metric) is made an end(goal) in itself then something vital is going to be lost eg profit becomes more important than people).

Aspiring software engineers know which qualitative goals to seek - this thesis is full of them - but they have not yet found the required metrics which will enable project managers and customers to easily and accurately monitor the progress and quality of software projects. Any revolutionary solution to the software crisis is more likely to come from a *quantitative* approach to software design. Effective metrics are a prerequisite of widespread acceptance. Further work is required to produce reliable, accurate and acceptable measurement systems for such things as

The measurement of the creative productivity and skill of certain grades of worker using certain methods of practice will no doubt be the most difficult and contentious quantitative exercise but it is vital and must be done using the most respectable of scientific methods. The number of lines of code produced per man-day is clearly an inadequate metric but it is still regularly used to forecast delivery dates and measure progress.

Understandability and maintainability again both border on the psychological domain but perhaps the 'histogram' approach to quality control of section 7.5.3 indicates that simple, uncontroversial metrics may be possible. Correctness and efficiency are perhaps the least controversial, probably because some acceptable metrics already exist, and they are therefore the most likely candidates for short term success. It is criminal that the scientific calculation of performance estimates is not a widespread practice and that easy-to-use performance and resource monitoring systems are not generally available.

When a comprehensive set of measurement techniques, applicable throughout the whole software life cycle, has been developed allowing software engineers to compare predicted estimates with actual progress and products, then and only then, will the software crisis be solvable. What is required is the development of techniques to support a software production method which includes

1. Mathematically based design techniques

2. Formal proofs of correctness of the designs

3. Performance prediction by formal analyses of the designs

4. Actual performance monitoring: comparison with predicted values.

Can such a design and production method be developed? Well, all four techniques were used in the 1940s by von Neumann et al.[59] No wonder it is called re-search.[23]

CHAPTER 10. APPENDICES

10.1 Published Papers
10.2 The 'B2D' Example Program

OUTLINE

Section 10.1 appends four papers published during the project.

Section 10.2 appends the large example program, B2D, which performs Binary to Decimal conversion. Examples of the design stage, both hand- and machine-drawn are given together with compilation, static quality analysis and dynamic performance measurement, debugging information and animation.

10. APPENDICES

10.1. Published Papers

10.1.1. "The Design and Construction of Hierarchically Structured Software".[85]

The design and construction of hierarchically structured software

R. W. Witty, Senior Scientific Officer

Atlas Computing Division,
Rutherford Laboratory,
England, OX11 0QX.

Abstract

The paper describes the principles of, the software tools for and the experience gained with a methodology which allows a single, automated representational technique to be used throughout the design, construction and maintenance of hierarchically structured software. The methodology is based on Dimensional Flowcharting which is a graphical technique to represent Sequence, Parallelism and Refinement - the three major structural features of hierarchically structured software. Examples are given showing how Dimensional Flowcharting represents the structure and content of programs, data structures, file organisations, grammars and documentation.

1. INTRODUCTION

In this paper Section 1 briefly outlines the theory of Dimensional Flowcharting, Section 2 briefly outlines the associated software tools and Section 3 gives details of the experience gained in practice.

1.1 Concepts of Dimensional Flowcharting

Hierarchically structured software has three major structural features - Sequence, Parallelism and Refinement. Dimensional Flowcharting [1] is a graphical technique to display these three features.

To create hierarchically structured software by the Step-wise Refinement method [2] a specification is broken down or refined into a sequence of instructions. Each instruction may itself be further refined. Figure 1 shows how in conventional Step-wise Refinement there is no way of showing which instructions are derived from which specifications whereas in the Dimensional Flowchart of Figure 2 the relationship is obvious. Refinement is denoted by a diagonal line connecting a specification to its refinement ie to a sequence of instructions. (Note that design by Step-wise Refinement is used throughout this paper but any design methodology which creates hierarchically structured software may equally well use Dimensional Flowcharting as its representational technique.)

A sequence of instructions is represented as a vertically ordered list which is executed from top to bottom. If a sequence is to be executed only once then it is called a Single Sequence and is indicated by the '⏚' (earthed) symbol which can be omitted for brevity (Figure 2). If a sequence is to be executed one or more times then it is called a Repeated Sequence and is indicated by the '⏚*' (repeated) symbol which cannot be omitted (Figure 4).

Figures 3 and 4 show an infinitely repeated sequence. Dimensional Flowcharting represents a Conditional Statement by the 'C' (conditional) symbol which, unless the specified condition is met, causes the execution of the sequence to be terminated and the next sequential instruction to be executed. The next sequential instruction is the one following the specification of the terminated sequence, see Figure 4. Reaching an 'earthed' symbol causes termination of the sequence.

In general, any specification may be broken down into one or more sequences of instructions, with each sequence being executed in parallel (Figure 6). A specification which is refined into several parallel sequences terminates if and when all its constituent sequences terminate.

```
SOLVE QUADRATIC EQUATION                level 1 - specification

INPUT COEFFICIENTS
COMPUTE ROOTS BY FORMULA                level 2 - refinement of level 1
OUTPUT RESULTS

READ (A)
READ (B)
READ (C)
COMPUTE ROOTS BY FORMULA                level 3 - refinement of 2 level 2
PRINT (A,B,C)                           instructions
PRINT (POSROOT,NEGROOT)
```

Figure 1. Step-wise Refinement

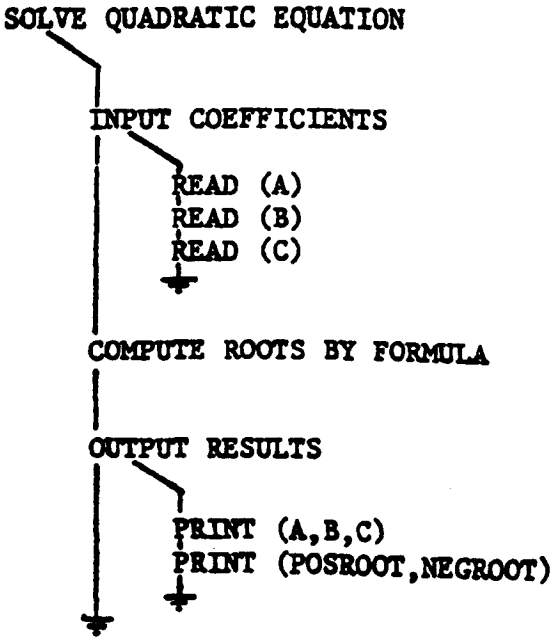


Figure 2. Dimensional Flowchart of Figure 1

```
comment  SOLVE MANY QUADRATIC EQUATIONS

     repeat

          INPUT COEFFICIENTS
          COMPUTE ROOTS BY FORMULA
          OUTPUT RESULTS

     endrepeat
```

Figure 3. An infinitely repeated sequence

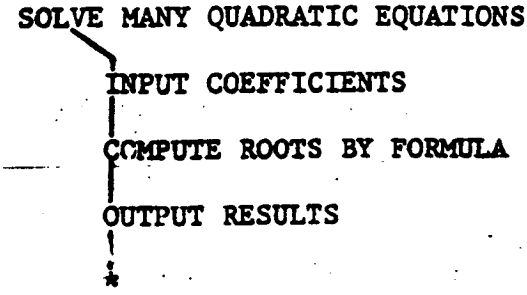


Figure 4. Dimensional flowchart of infinitely repeated sequence

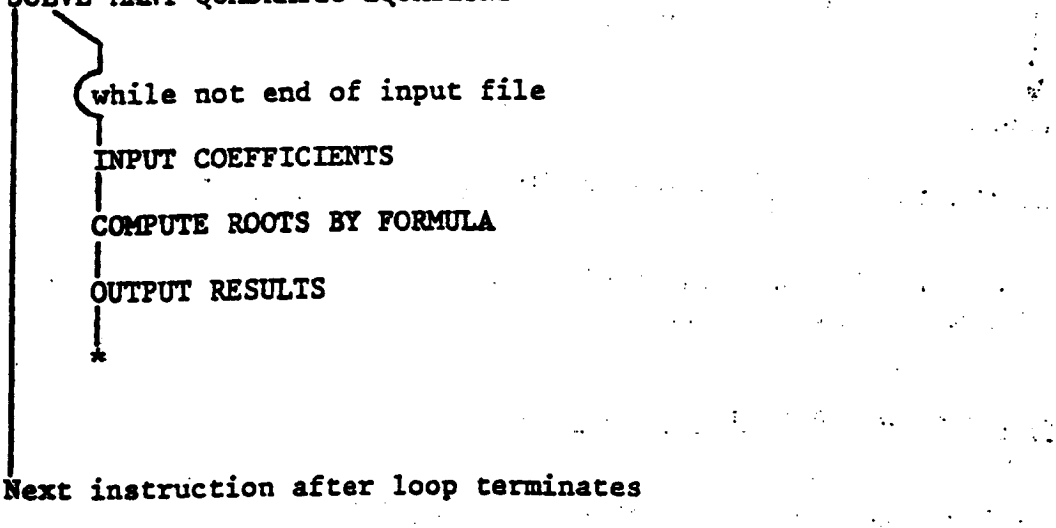


Figure 5. Finite Repeated Sequence

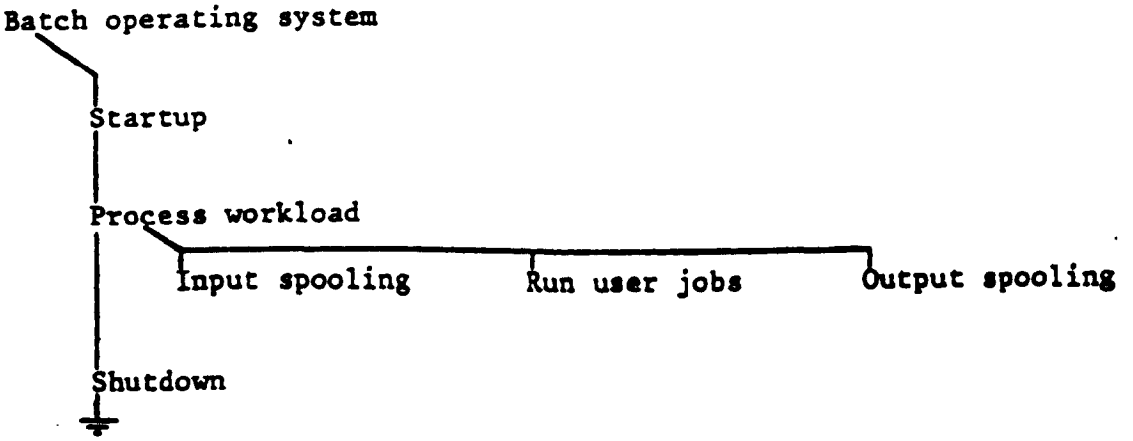


Figure 6. Parallelism

Figures 1-6 show that hierarchically structured software can be represented by tree diagrams in which the three major structural features of such software are represented by lines in particular directions (Figure 7). Each feature may be thought of as a dimension of the structures, hence the name Dimensional Flowcharting. Dimensional Flowchart drawings are thus 2-D projections of 3-D tree structures.

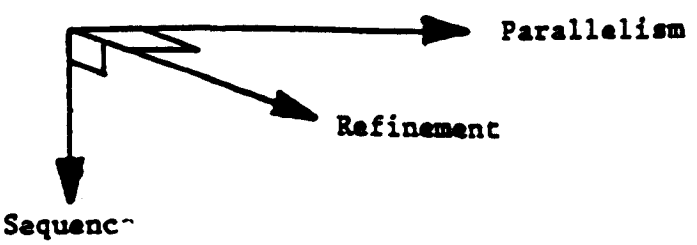


Figure 7. The 3 dimensions of Dimensional Flowcharts

Now that all the concepts of Dimensional Flowcharting have been introduced some examples are presented.

1.2 Examples of Dimensional Flowcharting

Figures 8-12 show how Dimensional Flowcharting directly models high level language control structures such as Dahl's loop, If-Then-Else, Case and Dijkstra's Guarded Commands [3]. Note the use of parallelism to represent alternatives.

Figure 13 is a development of Figures 1 and 2 showing how 'compute roots by formula' is refined into the well known formula for solving quadratic equations. Note the mutually exclusive Boolean expressions used in the IF statement.

Data structures as well as algorithms may equally well be represented by Dimensional techniques. Note the obvious lack of parallelism in the totally sequential tape file of Figure 14, which shows a file of 51 blocks each containing 99 records. Figure 15 outlines three popular file organisations. Contrast their parallelism.

Dimensional Flowcharts, being tree structures, may be described by context free grammars (Figure 16). Such grammars can themselves be represented by Dimensional techniques (Figure 17). (Note that the grammar in Figure 16 uses the extended BNF notation of the Tree-Meta system [4] in which 'empty' is the null symbol which is always recognised, '/' is the alternate symbol and '$' is read as 'zero or more of'.)

Any hierarchical structure with three structural features or dimensions can be represented by the Dimensional technique. Figure 18 is the structure of this paper. Note how the diagrams are to be read in parallel with the descriptive text.

The above examples conclude the outline of the theory of Dimensional Flowcharting. More details can be found in reference [1]. How is Dimensional Flowcharting used in practice to help design and maintain real software?

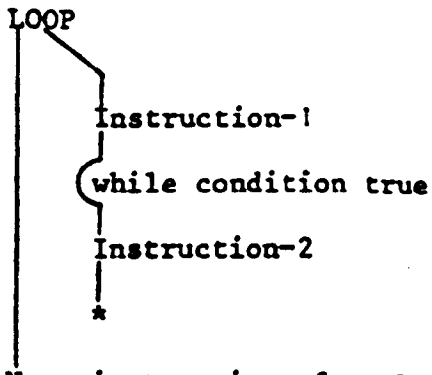


Figure 8. Dahl loop

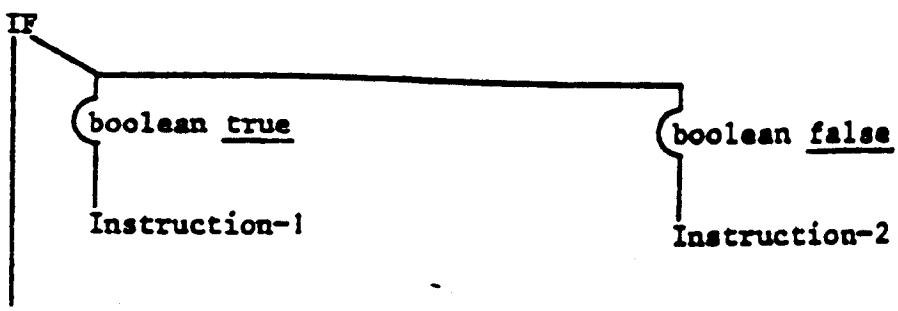


Figure 9. If-Then-Else

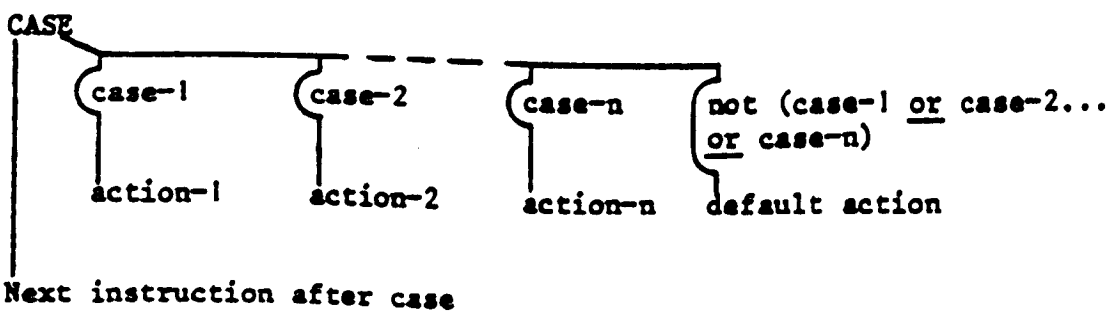


Figure 10. Case statement

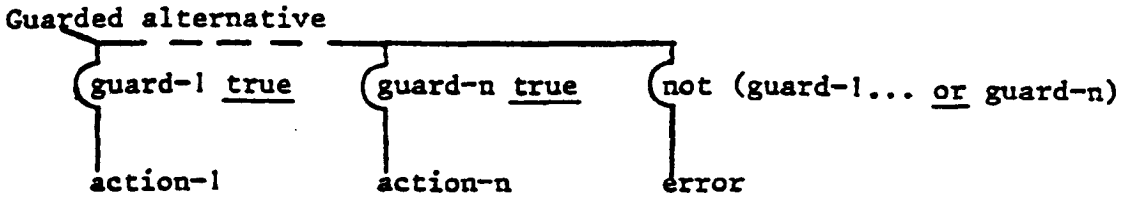


Figure 11. Guarded alternative

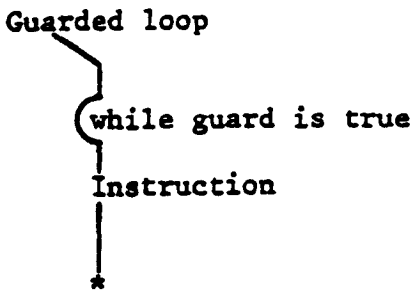


Figure 12. Guarded loop

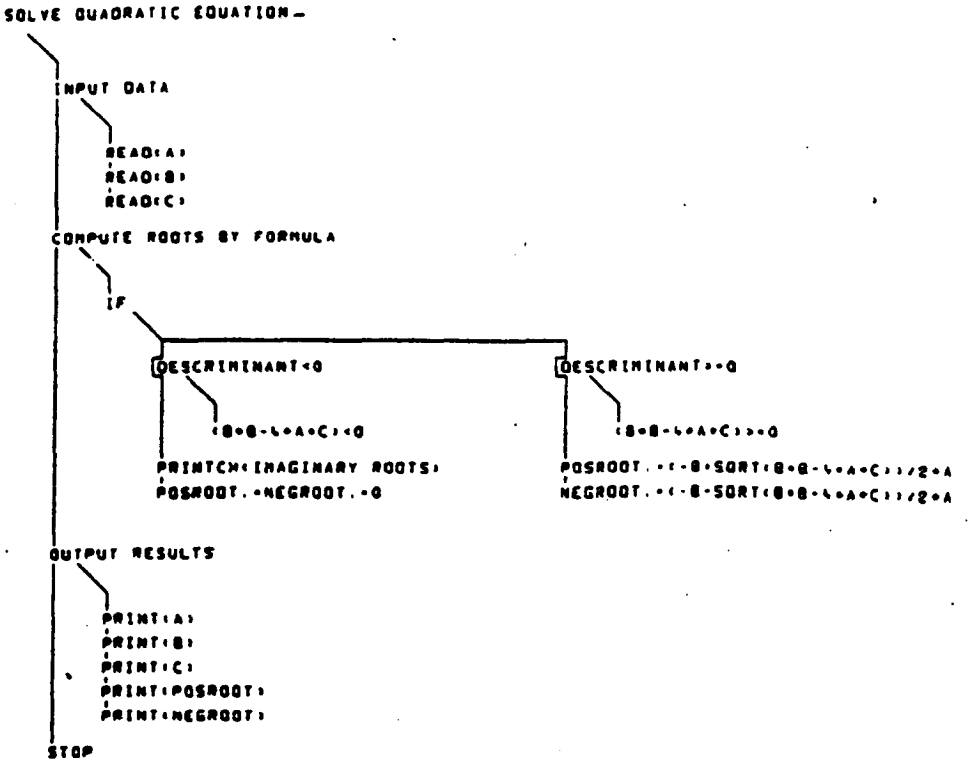


Figure 13. Quadratic equation program

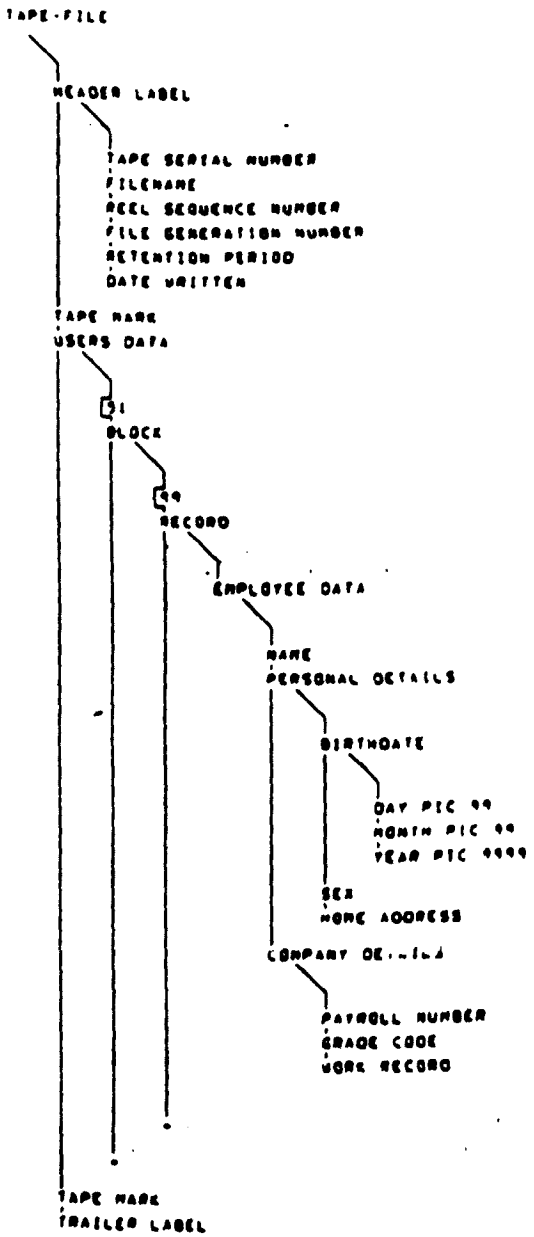


Figure 14. Tape file structure

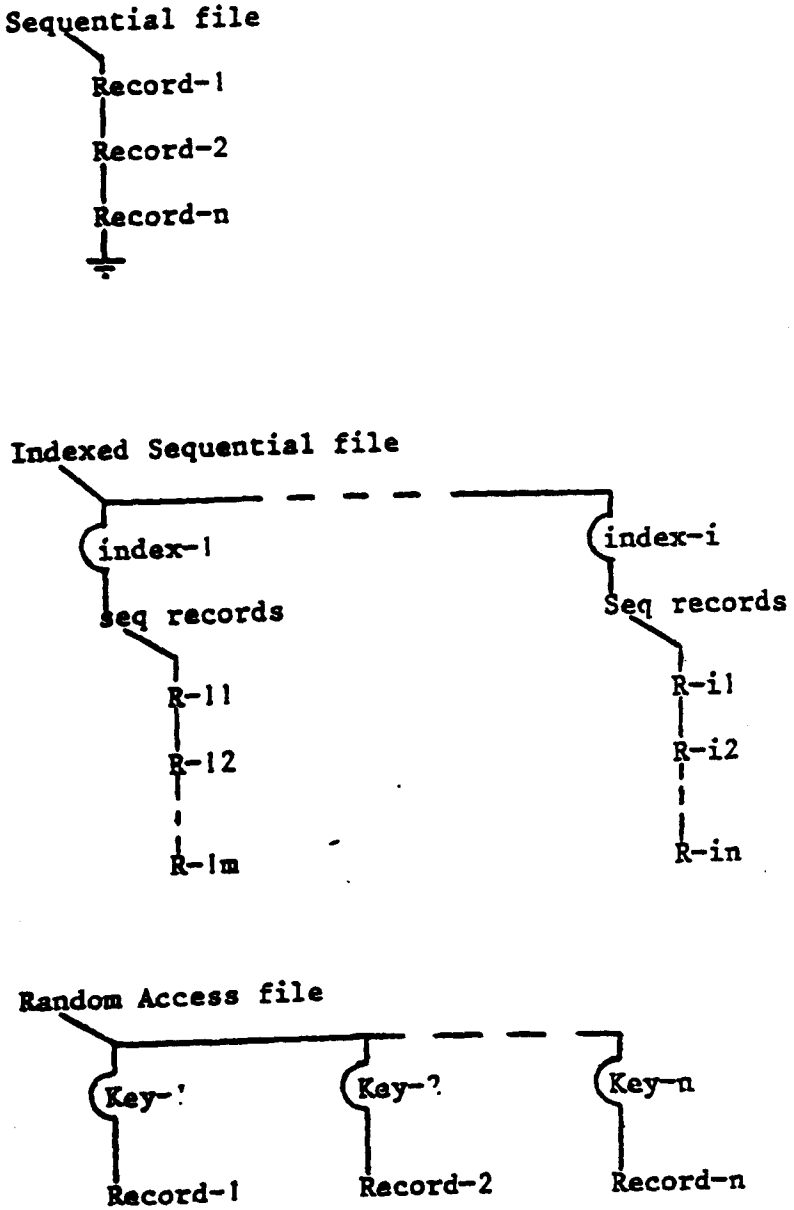


Figure 15. File organisations

```
STATEMENT = (ACTION / CONDITION) (REFINEMENT / .empty)

REFINEMENT = SEQUENCE $(SEQUENCE)

SEQUENCE  = $(STATEMENT) ('⏚'  / '↓')
```

Figure 16. Grammar to generate Dimensional Flowcharts

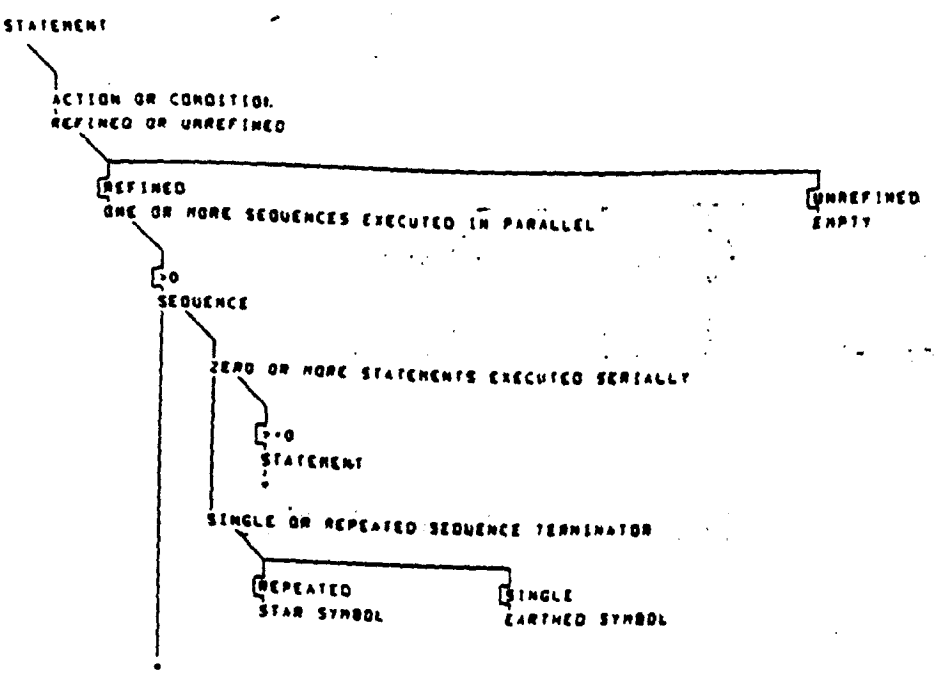


Figure 17. Dimensional representation of grammar in figure 16

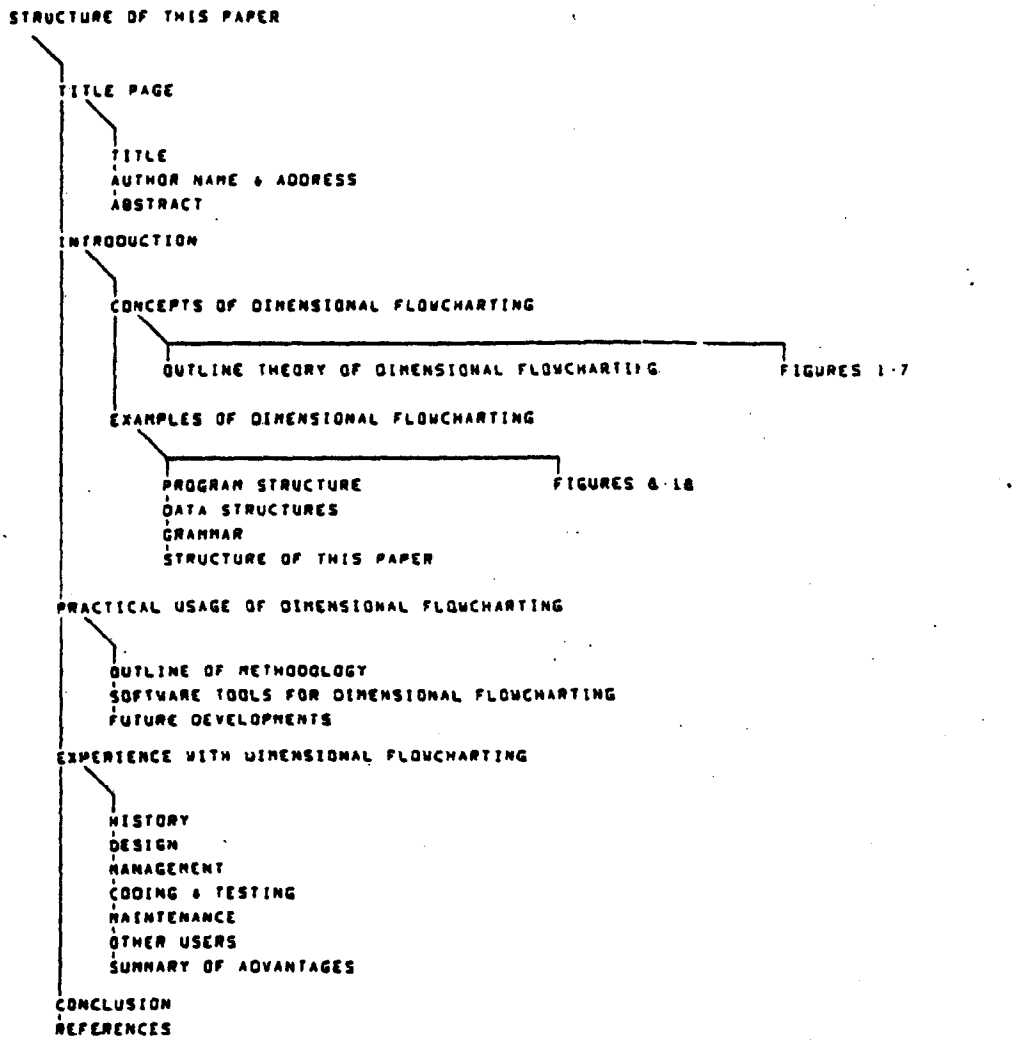


Figure 18. Dimensional representation of this paper's structure

2. PRACTICAL USAGE OF DIMENSIONAL FLOWCHARTING

2.1 Outline of Methodology

This section outlines how Dimensional Flowcharting is used in practice by the author and his colleagues. Suppose a new program is to be built from a specification. The initial design work is done using Dimensional Flowcharts hand drawn on the 'backs of envelopes'. Step-wise Refinement of the initial design sketches causes a proliferation of small Dimensional Flowcharts as the work progresses. As the design firms up the set of small, scappy drawings is combined into a single, unified drawing on one large sheet of paper up to four feet square. A full sized draftsman's drawing board is employed on this task. Programs of between 500-2000 lines can be expressed in one such drawing. For larger systems one drawing represents one major module such as a device handler.

Refinement of this design drawing continues until a complete program is derived ie when all the terminal nodes of the tree structure are compilable source code. At this stage the Dimensional Flowchart is coded up into one of two special languages, DRIL [5] and FOREST [6], which support 3-D hierarchical structuring. Coding is the process of reducing a 3-D structure to a 1-D machine readable form. This is achieved by 'tree-walking' the Dimensional Flowchart ie going through the flowchart in a particular order encoding each line and statement as it is 'visited'. The DRIL and FOREST compilers can internally recreate the Dimensional Flowchart as they 'know' the ordering algorithm which is based on the grammar in Figure 16.

The machine readable source code version of the Dimensional Flowchart is then compiled. The output from the compilation system is a binary program and, instead of the conventional source listing, a neat, well laid out Dimensional Flowchart. Thus the visible output from the compiler is an exact, but neater, replica of the program's design. Figure 19 illustrates this circle from hand-drawn design to machine-drawn documentation.

2.2 Software Tools for Dimensional Flowcharting

Figure 20 illustrates the software tools currently associated with Dimensional Flowcharting. Dimensional Flowcharts may be encoded into one of two languages, DRIL and FOREST. FOREST is a 'Structured FORTRAN' and DRIL is a 'Structured Assembler (Figures 21 and 22). Both languages essentially provide modern well disciplined control constructs. Both support hierarchical structuring by means of Scoped Comments [1] which allow Refinement, the third dimension, to be expressed in a linear fashion. The DRIL compiler is produced via the Tree-Meta compiler-compiler [4] and the FOREST compiler is written in itself. Each of these two compilers has twin code generators. One code generator produces Assembler (or FORTRAN) and the other produces

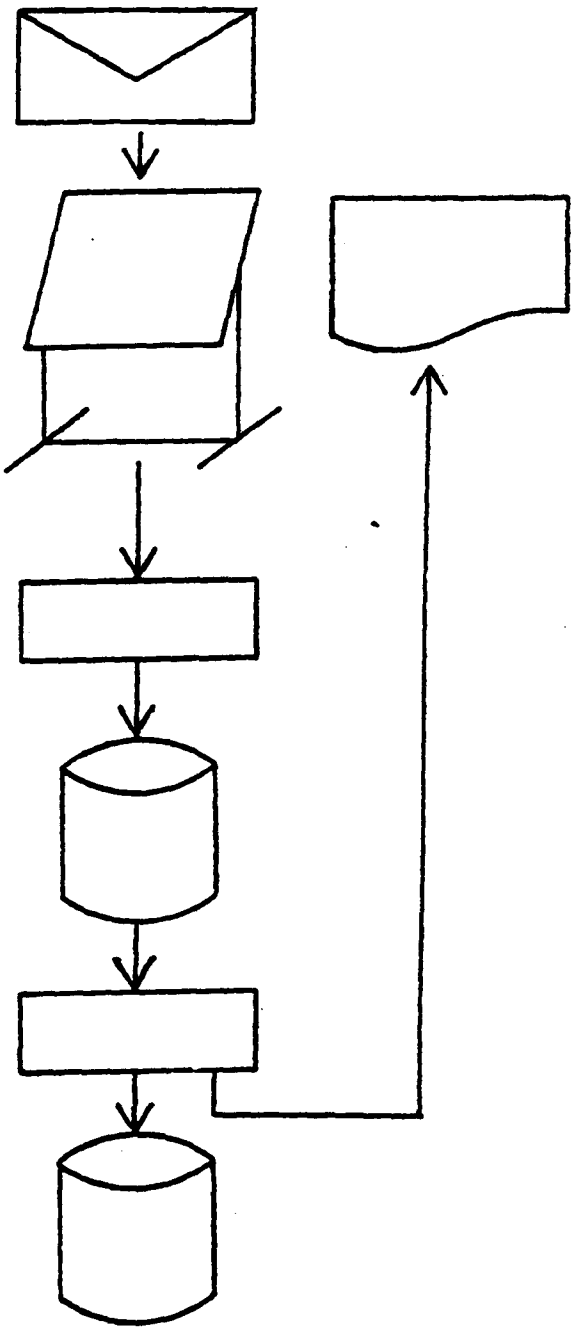


Figure 19. Outline of use of Dimensional Flowcharting

a common linear machine readable Dimensional Flowchart language which is input to a drawing program. The Dimensional Flowchart drawing program is outlined in Figures 23-25 which show the input language syntax, the formatting algorithm and an example input file.

Output from the drawing program can be produced on a lineprinter and a storage tube for a 'quick look', or the output can be directed to an FR80 Microfilm Recorder which produces top quality flowcharts on microfilm, microfiche and photographic paper. Several of the diagrams in this paper were produced on the FR80 using the above software tools.

2.3 Future Developments

The tools described in section 2.2 form a batch compilation system. To change an existing program the programmer must edit the machine readable version in the conventional manner. It is intended, when time and effort permit, to build an interactive refresh graphics Dimensional Flowchart editor so that the designer and programmer may both interact exclusively with the graphical 3-D form of flowchart, in other words to replace the draftsman's drawing board and the linear source code by interactive computer graphics. As a first step in this direction the current drawing program has the facility to 'turn-off' the refinement of any given instruction so that drawings may be produced which have selected refinements missed out. This enables large programs to be drawn on small pieces of paper on which only selected features are refined in detail.

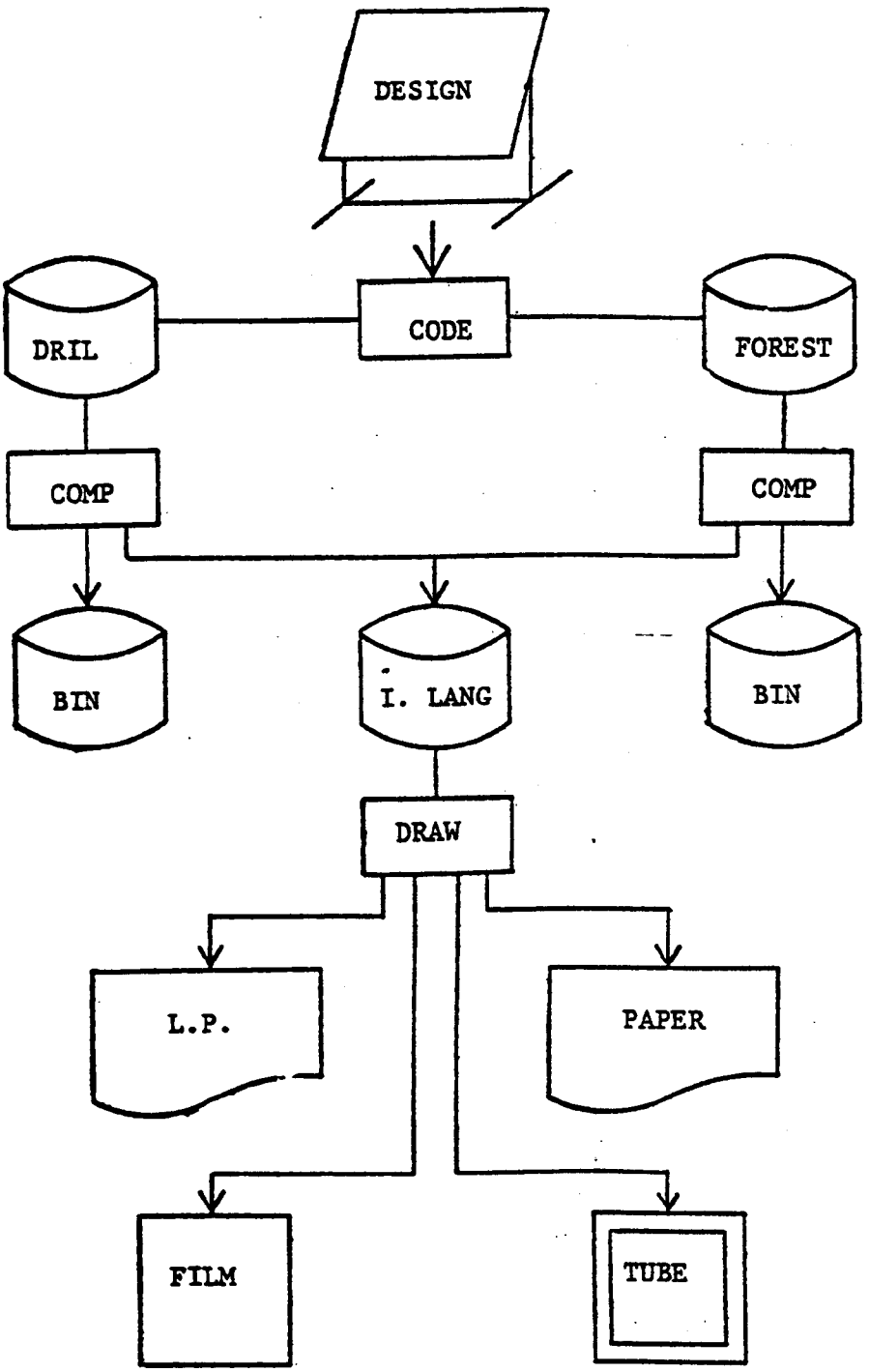


Figure 20. Dimensional Flowcharting Software Tools

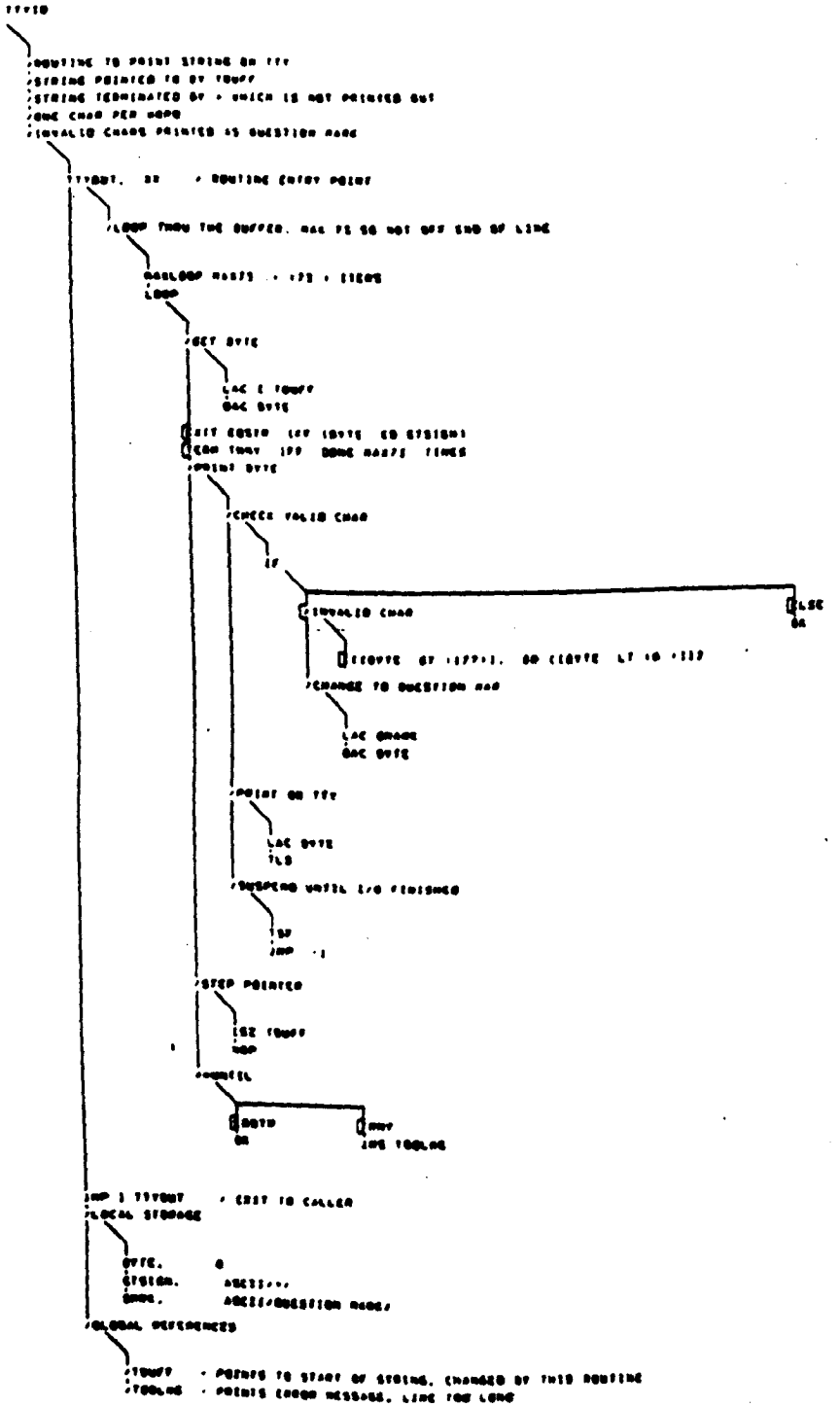


Figure 21. Design of Teletype printing routine

```
.PROG TTY3D
.SCALE 2000
      \
"ROUTINE TO PRINT STRING ON TTY"
"STRING POINTED TO BY TBUFF"
"STRING TERMINATED BY > WHICH IS NOT PRINTED OUT "
"ONE CHAR PER WORD"
"INVALID CHARS PRINTED AS QUESTION MARK "
      \
'TTYOUT,  XX     / ROUTINE ENTRY POINT'
      \
"LOOP THRU THE BUFFER, MAX 73 SO NOT OFF END OF LINE"
      \
.MAXLOOP MAX73  := 73 .ITERS
.LOOP

    "GET BYTE"
        \
'       LAC I TBUFF'
'       DAC BYTE'
         #

    .TERM EOSTR .IFF BYTE .EQ GTSIGN .ENDE

    .TERM TMNY .IFF .DONE MAX73 .TIMES .ENDE

    "PRINT BYTE"
        \
       "CHECK VALID CHAR"
           \
          .IF "INVALID CHAR" \  [BYTE .GT (177)] .OR [BYTE .LT 0] #
             .THEN
                "CHANGE TO QUESTION MARK "
                   \
              '       LAC QMARK'
              '       DAC BYTE'
                       #
             .ELSE
                .OK
          .ENDI
           #

       "PRINT ON TTY"
           \
'       LAC BYTE'
'       TLS'
         #
```

```
        "SUSPEND UNTIL I/O FINISHED"
            \
    '       TSF'
    '       JMP .-1'
             #
          #

     "STEP POINTER"
         \
'       ISZ TBUFF'
'       NOP'
         #

.REPEAT
     .SIT EOSTR .CAUSES .OK .ENDS
     .SIT TMNY  .CAUSES '       JMS TOOLNG' .ENDS
.ENDL
       #
    #

'       JMP I TTYOUT     / EXIT TO CALLER'


"LOCAL STORAGE"
       \
     'BYTE,       0'
     'GTSIGN,     .ASCII/>/ '
     'QMRK,       .ASCII/QUESTION MARK/ '
       #


"GLOBAL REFERENCES"
       \
    "TBUFF    - POINTS TO START OF STRING, CHANGED BY THIS ROUTINE"
    "TOOLNG   - PRINTS ERROR MESSAGE, LINE TOO LONG"
       #


       #
       #
.ENDP TTYOUT
```

Figure 22. DRIL source coding of Teletype printing routine

```
Machine readable flowchart = scalefactor statement

statement = (action/condition) (refinement/.empty)

action =    singlequote text singlequote

condition = query text query

refinement = sequence $(sequence)

sequence =  '\' $(statement) ('*'/'#')
```

Figure 23. Syntax of machine readable Dimensional Flowchart

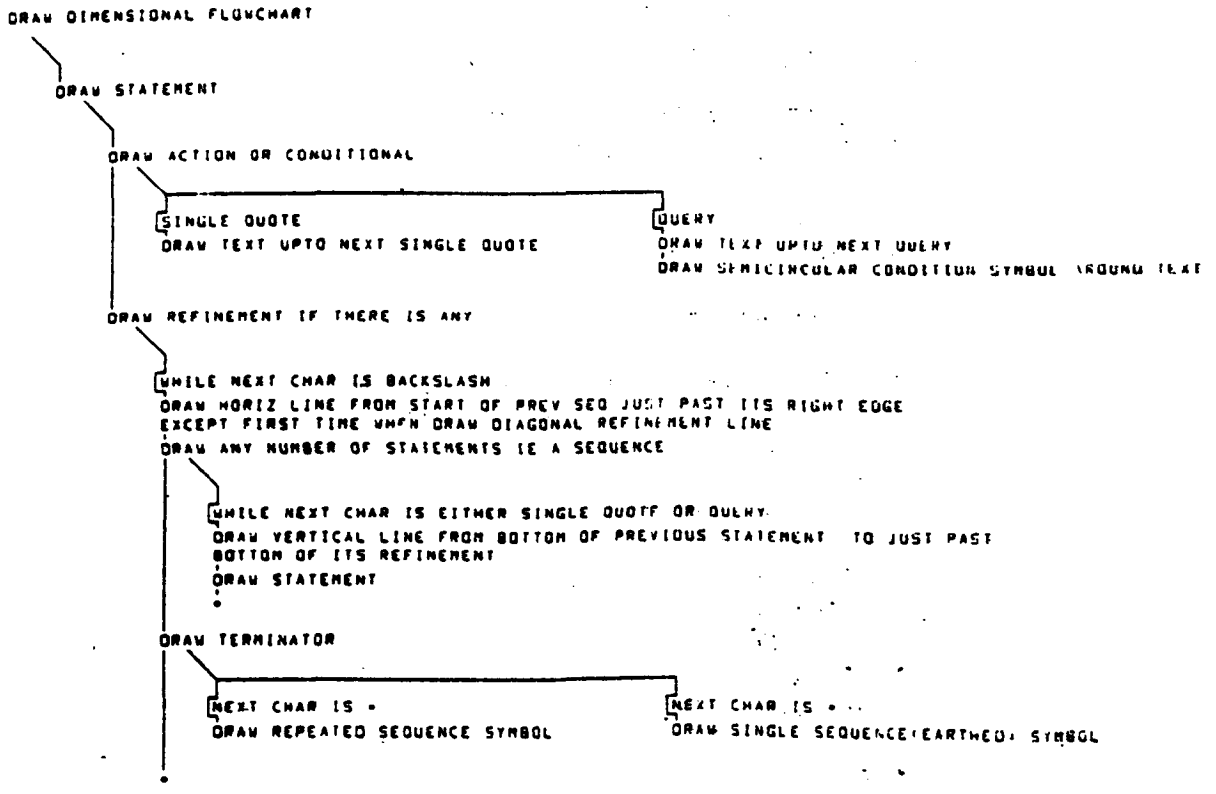


Figure 24. Algorithm to draw Dimensional Flowcharts

```
1500
'SOLVE QUADRATIC EQUATION'
  \
'INPUT DATA'
  \
'READ(A)'
'READ(B)'
'READ(C)'
  #
'COMPUTE ROOTS BY FORMULA'
  \
'IF'
  \
?DESCRIMINANT<0?
  \
'(B*B-4*A*C)<0'
  #
'PRINTCH(IMAGINARY ROOTS)'
'POSROOT:=NEGROOT:=0'
  #
  \
?DESCRIMINANT>=0?
  \
'(B*B-4*A*C)>=0'
  #
'POSROOT:=(-B+SQRT(B*B-4*A*C))/2*A'
'NEGROOT:=(-B-SQRT(B*B-4*A*C))/2*A'
  #
  #
'OUTPUT RESULTS'
  \
'PRINT(A)'
'PRINT(B)'
'PRINT(C)'
'PRINT(POSROOT)'
'PRINT(NEGROOT)'
  #
'STOP'
  #
```

Figure 25. Example of machine readable Dimensional Flowchart

3. EXPERIENCE WITH DIMENSIONAL FLOWCHARTING

3.1 History

Dimensional Flowcharting has been in use since May 1975 and has been used to produce three major programs. These are FR80 SYSLOG[7], a program to log accounting data about user jobs, hand coded in Assembler, FINGS[8], a graphics package, coded in FOREST, and FR80 DRIVER[9], a multi-tasking control system for a microfilm recorder, coded in DRIL.

Dimensional Flowcharting was developed during the design of FR80 SYSLOG. It was decided to use Step-wise Refinement and Structured Programming principles for this program. Applying these techniques during the design stage brought forth the problem of representation. Conventional flowcharts were used initially but were found to have the following disadvantages.

1. They cannot be drawn naturally and easily; programming languages are often easier to use because they follow the normal lexical directions of written English and do not need special symbols and boxes.
2. They do not conveniently represent modern high level well-disciplined control constructs.
3. They do not clearly distinguish between Sequence and Parallelism.
4. They fail to show how a final, detailed design has been achieved through Step-wise Refinement.
5. They do not represent data structures.
6. They are difficult to store and manipulate in a computer and are difficult to draw on a plotter.

Design representation by a very high level language was tried next. This was more successful as it overcame objections 1, 2 and 5 but objections 3, 4 and 6 remained. Why did these three objections remain? Programming languages are essentially linear strings of symbols designed as much for machine consumption as human. They are one dimensional. Programmers work two dimensions into their programs by placing one statement per line (the Sequence dimension) and by indenting sequences to simulate a second dimension, Refinement. This is an ad-hoc, informal technique which is limited, as is so much of programming language design, by the Procrustean limitations of the 80 column card and the 132 column lineprinter. Indentation is usually halted after only a few levels by the inadequate width of the input and output media. Most programming languages do not adequately support Refinement or hierarchical structuring. Their major structuring

mechanism is the subroutine, a mechanism for saving instruction storage within a computer rather than a mechanism for representing logical decomposition. Thus well structured programs can incur unnecessary subroutine overheads. This illustrates a point learnt from experience, namely that every program has a Logic Design (the algorithmic solution) and a Physical Construction Design (the source code program) analogous to the hardware engineer's Logic Diagram and Wiring Diagram or Chip Mask (see reference [1]). Subroutines are a Physical Construction mechanism whereas Refinement is a Logic Design feature.

It was from this background that Dimensional Flowcharting was developed to overcome objections 3, 4 and 6, particularly 4. FR80 SYSLOG was eventually designed using Dimensional Flowcharting, the design being refined right down to the bit level as Assembler was then the only available language. The design was 'treewalked' into Assembler and the finished program of 8,000 executable instructions has been successfully running unmodified since September 1975. Based on the success of FR80 SYSLOG it was decided to construct the DRIL compiler and flowchart drawing program to take some of the hard work out of drawing and coding. A FORTRAN-based equivalent, FOREST, was also undertaken, mainly for the development of a new graphics package called FINGS. DRIL has been used to build the multi-tasking control system FR80 DRIVER. This program currently contains over 20,000 lines of DRIL. It is represented on 10 Dimensional Flowcharts each of which contains 1,000-3,000 lines of code. As each drawing is around four feet square it is impossible to reproduce real examples of Dimensional Flowcharts in this paper. All the remarks in this paper about Dimensional Flowcharts should be related to such large drawings of 2,000 lines of code as Dimensional Flowcharting was primarily invented to represent the logic of large complex modules.

3.2 Design

During the initial design phase of a project Dimensional Flowcharts have proved quick and easy to draw. They provide detailed control constructs even at the most abstract level of design which is an advantage over some design notations. The explicit representation of Refinement has been of great help in creating clean designs.

It has been easy to produce formal designs from the design sketches via a draftsman's drawing board because Dimensional Flowcharts contain no boxes, have statements of any length, require few special symbols, need no templates to draw - only pencil and ruler and their simple formatting algorithm really is easy to use. Using one sheet of paper to hold the design of a large module with a unified notation for both program and data structures has helped greatly to improve the quality of software design. Each sheet is an 'engineering drawing' containing many macros, subroutines and data structures, amounting to around the 2,000 lines of code necessary to

implement a complex function like a sophisticated device handler. Because of the explicit Refinement comprehension is aided by individual statements generally being refined into less than 10 statements each. This is recursively true for any level of the hierarchy.

The large single sheet approach to design enables the designer to view his design at any desired level of abstraction. He may investigate the tiniest detail or take a global view. The global view has enabled messy parts of a design to be spotted easily. A proliferation of nested If-Then-Elses is very easy to spot and turn into a Case statement. Experience has shown that good, clean designs actually look good, simple and clean when represented as Dimensional Flowcharts. The global view also helps in deciding which pieces of logic to extract as macros and subroutines as similar shapes are a due to similar functions. The converse is also true. In DRIVER two sections of code which were meant to be functionally identical (but which were not worth making into a subroutine) were seen to have slightly different shapes. Finding and fixing a bug in one made the two shapes identical.

The global view afforded by the single sheet design can be shared by several people simultaneously. Discussions, design inspections and walkthroughs are helped by this accessability. Contrast this with say four people looking at the same program listing whilst keeping half a dozen fingers in it to mark the set of subroutines currently of interest.

As a large design progresses it must be periodically redrawn. This can be tedious and time-consuming. Thus large, partially refined designs are 'treewalked' into their target source code and compiled. A binary program cannot be produced but a neat machine drawn flowchart can be. The large design is thereafter altered and expanded by editing a conventional source file; ideally a computer graphics terminal should replace the drawing board, source file and editor - see section 2.3. Small changes do not now mean a lot of redrawing by hand, and different versions of the same design are easily created and stored. This means that neat, accurate, up-to-date design drawings can be easily copied and circulated amongst project members and managers.

3.3 Management

Dimensional Flowcharting makes programming progress visible. Managers are now able to see, and appreciate, how an initial design has progressed as they can see the expansion that has taken place during refinement. This helps to prevent the premature rush towards coding as the only evidence of progress. Presentations to managers can now be by way of neat machine drawn flowcharts rather than probably, to them, unreadable program listings. Dimensional Flowcharts make it easier for a project leader to see how far there is

still to go and help him estimate the amount of work to completely refine a partial design. It is easier for him to see how to split up the work and for any individual programmer to see where 'his bit' fits into the whole system as he can see the higher level structure above 'his bit'.

One representational technique now spans the whole process from initial design right through to detailed coding and maintenance. The same representational technique is used by the author and his colleagues in the design and coding of programs in different languages and on different projects (DRIVER and FINGS). This unity simplifies communication and discussion between projects and helps in the easy transfer of staff from one project to another.

3.4 Coding and Testing

Coding of a completely refined design is almost a mechanical process as all the executable instructions are contained in the design. Indeed, if the design itself has been build in the machine (section 3.2) then no coding stage is necessary. A major help in checking the coding of a design is the fact that the 'compilation listing' is actually a Dimensional Flowchart which should be exactly the same as the hand drawn version which was encoded. Many coding and punching errors have been quickly spotted by noticing differences in the shapes of these two documents (figure 19). Any design represented as a Dimensional Flowchart cannot be coded up into anything else but a hierarchically structured program which conforms to the principles of Structured programming.

Having Dimensional Flowcharts around at testing time is a great help as the programmer needs to appreciate the overall design when looking for bugs, wandering up and down the logic hierarchy until descending onto the fault part. Having the whole design available helps the tester to see how the section under test interacts with sections he constructed a long time ago and has by now forgotten the details of, or interacts with a section that someone else built which he must now learn about. Dimensional Flowcharting naturally encourages Top-down Testing.

3.5 Maintenance

Typically the maintenance programmer has little or none of the original program design to work from. What little he has is usually out of date in that many small changes have probably been made since the documentation was issued. Often it is these small changes which cause the bugs. Dimensionally Flowcharted programs which contain the design as an integral part of the program help to overcome this problem as every time a change is made to an executable instruction the logic

hierarchy above it should also be corrected, as any changes to the terminal nodes of the hierarchical structure imply that the design has been altered. This is even more important if the program undergoes development rather than correction. Crude fixes, development patches and bad maintenance tend to be easily spotted as they cause aberrations in the shape of the Dimensional Flowchart.

3.6 Other users of Dimensional Flowcharting

Several people outside the author's organisation have also been using Dimensional Flowcharting. In general their experience has been similar to the author's, with one exception. They have tended to make small changes, developments and adaptions to the notation in section 1 to suit their own particular problems, eg adding a variation of the conditional statement representation to explicitly indicate semaphores. Adaptability is perhaps a strength in any representational technique.

3.7 Summary of the advantages of Dimensional Flowcharting

1. They are quick and easy to draw by hand, typewriter or computer because of their simple formatting algorithm, their lack of boxes, templates and special symbols.

2. They are easily stored and manipulated by computer.

3. They are easy to encode by the treewalking method.

4. They are programming language independent.

5. They model high level language control and data structures.

6. The technique is easy to adapt and extend.

7. They explicitly show Sequence, Parallelism and Refinement.

8. They encourage and facilitate well-structured program design and well-disciplined code. They facilitate design by Step-wise Refinement.

9. They facilitate design discussions and walk throughs.

10. They help the maintenance programmer by facilitating the learning and understanding of a program's design by retaining the hierarchical design as an integral part of the program.

11. They allow management to monitor the progress during the design stage. They allow programmers to appreciate where their work fits into the overall scheme.

4. CONCLUSION

Dimensional Flowcharts help the designer to produce hierarchically structured software, they help the programmer to produce well-disciplined code, they help the manager to monitor progress and they help the maintenance programmer to understand the finished product.

5. REFERENCES

1. R W Witty, 'Dimensional Flowcharting', Software - Practice & Experience, Vol 7 No 5, Sept/Oct (1977).

2. N Wirth, 'Program Development by Step-wise Refinement', Communs Ass Comput Mach, Vol 14 No 4, April (1971).

3. E W Dijkstra, 'Guarded Commands', Communs Ass Comput Mach, Vol 18 No 8, August (1975).

4. F R A Hopgood, 'The 1906A Tree-Meta Manual', Atlas Computer Laboratory, April (1974).

5. R W Witty, 'FR80 DRIVER Software Construction', Atlas Computer Laboratory, December (1975).

6. D A Duce, 'FOREST - a Structured Fortran Preprocessor', Prime User Note 5, Rutherford Laboratory, February (1977).

7. J Rushby and R W Witty, 'FR80 Logging System', Atlas Computer Laboratory, September (1975).

8. D A Duce and A H Francis, 'FINGS Primer and Manual', Prime User Note 17, Rutherford Laboratory, July (1977).

9. R W Witty, 'FR80 DRIVER', Rutherford Laboratory, April (1977).

10.1.2. "ROOTS".[22]

SCIENCE RESEARCH COUNCIL

RUTHERFORD LABORATORY
ATLAS COMPUTING DIVISION

P R I M E U S E R N O T E 7 2

ROOTS

issued by
D A Duce
R W Witty

June 6, 1979

DISTRIBUTION:

C J Pavelin	F R A Hopgood	R W Witty
L O Ford	R E Thomas	Miss J Sullivan
P E Bryant	D C Sutcliffe	Mrs E Krauesslar
D C Toll	J Brown	P J Newton
D A Duce	H K F Yeung	M P Keane
J R Gallop	R Brandwood(2)	I Benest
G W Robinson	C Greenough	K Robinson
P A Dewar	M J Newman	M Geary
S K Chanda	A D Bryden	Mrs J Hutchinson
D Boyd	J McLean	

W Swindells - UMIST
D N Kennedy - Nottingham
G Kendall - PRIME

1 INTRODUCTION

ROOTS is an extended FORTRAN language which is translated into pure FORTRAN by a preprocessor. The language was designed for use in the context of a design methodology which allows a single, automated representational technique to be used throughout the design, construction and maintenance of hierarchically structured software [1]. Section 2 below contains an outline description of this methodology.

ROOTS can however be used outside this context as an extended FORTRAN providing enhanced control structures, but the authors suggest that the reader pays some attention to section 2 and the possible benefits of using the software tools to the full.

ROOTS also provides performance monitoring, control and data flow tracing facilities which are integrated into the representational technique described below. These facilities can be used with existing FORTRAN programs, though this does require some effort and the results are not as easy to display as are the results when used with a true ROOTS program.

2 THE DESIGN OF HIERARCHICALLY STRUCTURED SOFTWARE

2.1 Representation of hierarchically structured software

One technique for the creation of hierarchically structured sofrtware is the Step-Wise Refinement method [2] in which a specification is broken down or refined into a sequence of instructions. Each instruction may itself be further refined. Such software has three major structural features, sequence, parallelism and refinement. Dimensional Flowcharting [1] is an automated graphical technique to represent these three features.

Figure 1 shows how in conventional Step-Wise Refinement there is no way of showing which instructions are derived from which specifications whereas in the Dimensional Flowchart of figure 2 the relationship is obvious. Refinement is denoted by a diagonal line connecting a specification to its refinement, ie to a sequence of instructions.

A sequence of instructins is represented as a vertically ordered list which is exectued from top to bottom. If a sequence is to be executed only once then it is called a single sequence and is indicated by the earthed symbol which is usually omitted for brevity. If a sequence is to be executed one or more times then it is called a Repeated Sequence and is indicated by the '*' symbol which cannot be omitted.

Conditional statements are represented by the 'C' (conditional) symbol which, unless the specified condition is met, causes execution of the sequence to be terminated and the next (higher level) sequential instruction to be executed. The next sequential instruction is the one following the specification of the terminated sequence, see figure 3. Reaching an 'earthed' symbol causes termination of the sequence.

In general, any specification may be broken down into one or more sequences of instructions, with each sequence being executed in parallel (figure 4). A specification which is refined into several pararallel sequences terminates if and when all its constituent sequences terminate.

Figures 2-4 show that hierarchically structured software can be represented by tree diagrams in which the three major structural features of such software are represented by lines in perpendicular directions (Figure 5). Dimensional flowchart drawings are 2-D projections of these 3-D tree structures.

```
SOLVE QUADRATIC EQUATION                level 1 - specification

INPUT COEFFICIENTS
COMPUTE ROOTS BY FORMULA                level 2 - refinement of level 1
OUTPUT RESULTS

READ (A)
READ (B)
READ (C)                                level 3 - refinement of 2 level 2
COMPUTE ROOTS BY FORMULA                instructions
PRINT (A,B,C)
PRINT (POSROOT,NEGROOT)
```

Figure 1. Step-wise Refinement

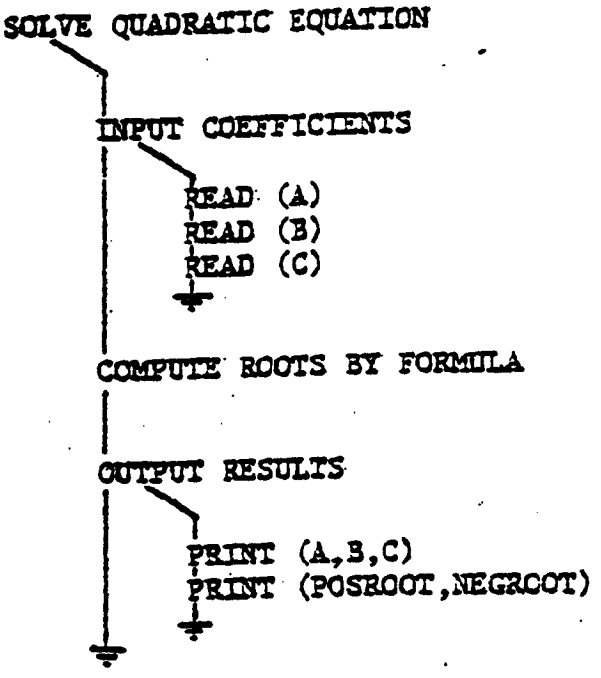


Figure 2. Dimensional Flowchart of Figure 1

Figures 6-8 show how certain ROOTS constructs are represented as flowcharts. The direction labelled 'parallelism' in figure 5 is used also to represent the selection of alternatives (see the IF and EPILOG examples).

2.2 Outline of design methodology

Suppose a new program is to be built from scratch. The initial design work is done with flowcharts hand drawn on the 'backs of envelopes'. The step-wise refinement method of the initial design sketches causes a proliferation of small drawings which are combined into larger drawings as the work progresses and the design firms up. Refinement of the design drawing continues until a complete program is derived, ie when all the terminal nodes of the tree structure are compilable source code. At this stage the flowchart is coded up into ROOTS. This is done by 'tree-walking' the flowchart, ie going through the flowchart in a particular order encoding each line and statement as it is 'visited'. The technique is easily acquired by studying the example of the ROOTS source of a program and the corresponding flowchart given later in this paper.

The machine readable source code is then presented to the ROOTS preprocessor. The output from the preprocessor is an equivalent FORTRAN program and an encoded form of the program which may be fed into a second software tool (the DFC generating program) which generates a dimensional flowchart on one of a variety of output devices. Output devices currently available include Tektronix storage tubes and the Benson 1302 pen plotter.

The output from the preprocessor is thus an exact, but neater, replica of the program's design. This is a major help in checking the coding of a design as the flowchart should be exactly the same as the hand drawn version encoded.

Flowcharts have also been found to be a major aid to program testing as they help the programmer to appreciate the overall design when looking for bugs. As mentioned above and discussed in more detail below, program execution traces can be obtained in the same representation and program performance data can be displayed on the flowchart representing the static program structure.

They are a help to the maintenance programmer as they facilitate the learning and understanding of a program's design by retaining the hierarchical design as an integral part of the program.

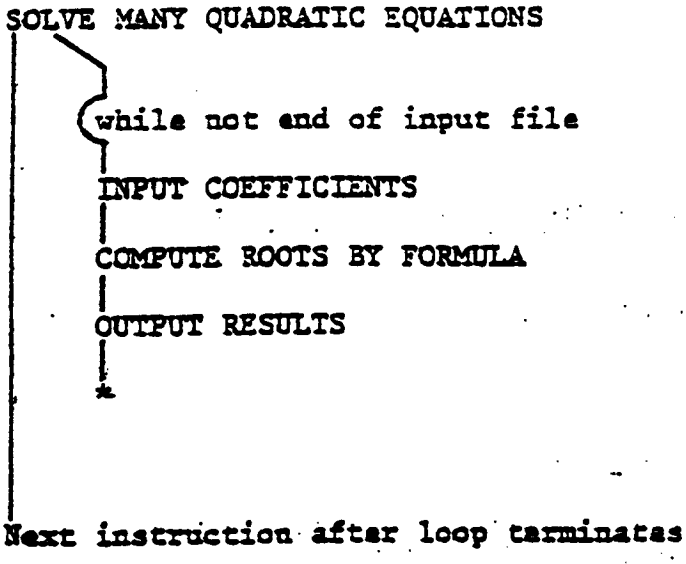


Figure 3

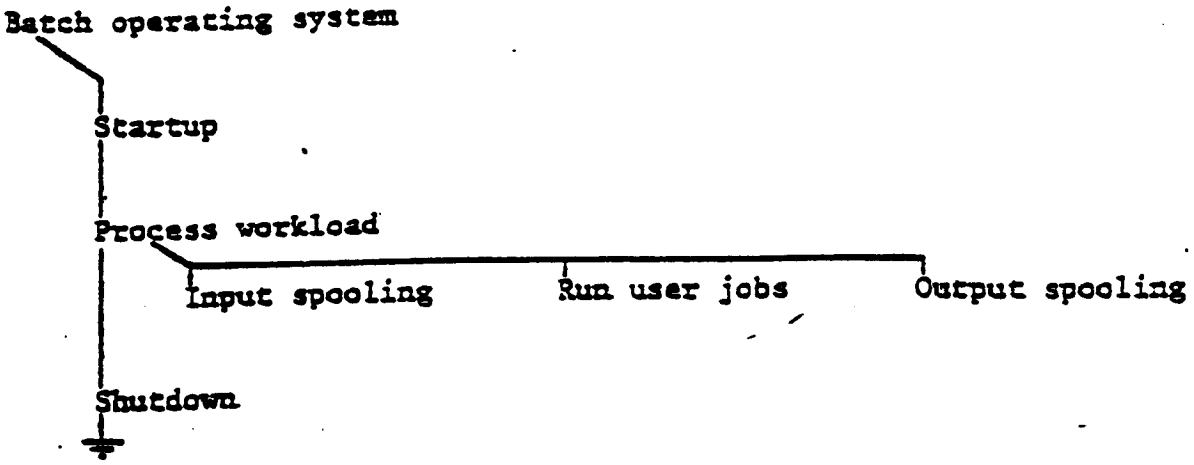


Figure 4

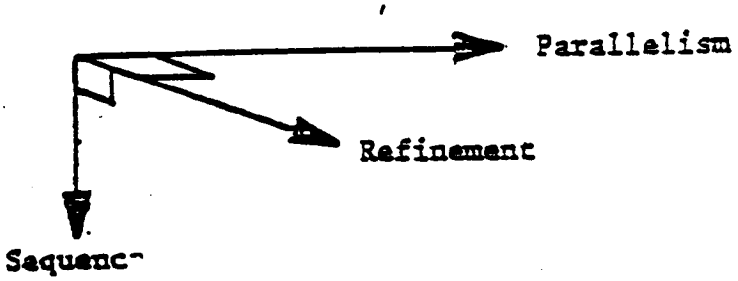


Figure 5

3 ROOTS - A STRUCTURED FORTRAN LANGUAGE

The remainder of this paper describes the run-time monitoring facilities provided by the ROOTS preprocessor, which gives a flavour for the control structures provided in ROOTS and the appearance of ROOTS programs. The remaining sections describe the language in detail and the mechanics of running ROOTS programs and of flowchart generation.

4 RUN-TIME MONITORING

It is a weakness of many language implementations that they provide no facilities for debugging and performance measurement in programs. Discovering which sections of a program consume the most resources can be exceedingly laborious and hence is rarely undertaken.

The ROOTS translator aims to provide users with facilities to measure the CPU time and IO time taken by nominated statements in the program, to generate an execution trace of nominated sections of the program and to monitor control and data flow in the program. At the language level, this is achieved by inserting monitoring statements at the start of the program which define the type of monitoring required, and then tagging those statements in the program for which monitoring is desired.

It is a further weakness of most run-time monitoring systems that the results produced are in the form of a sequential line printer listing and relating the results to the program can be exceedingly tedious. The ROOTS monitoring system is closely linked to the DFC system and by feeding the output from the monitoring system along with the flowchart code generated by the ROOTS translator into the DFC program static performance data and snap-shots of data will be displayed on the flowchart, as will be seen in figure 9.

The run-time execution trace generated by the monitor is also in the form of input data for the DFC program and this also is intended to be represented as a flowchart, see for example figure 10. There are several mechanisms provided to limit the volume of tracing output generated.

The facilities provided are best described by way of examples and for this the program below will be considered. The program converts binary representations of integers to character string representations of their decimal values by iterative and recursive algorithms. This is used to illustrate the point that the monitoring system is capable of handling recursion. Recursion is permitted as an option in the PRIME FORTRAN system.

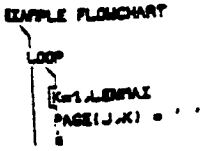


Figure 6

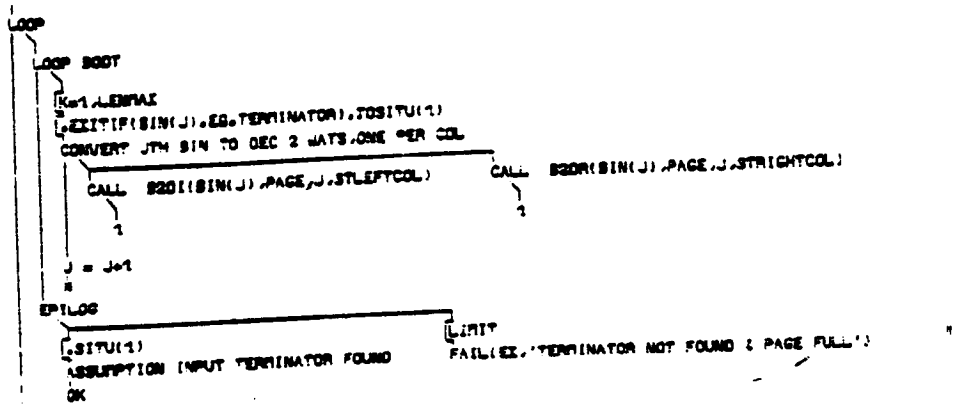


Figure 7

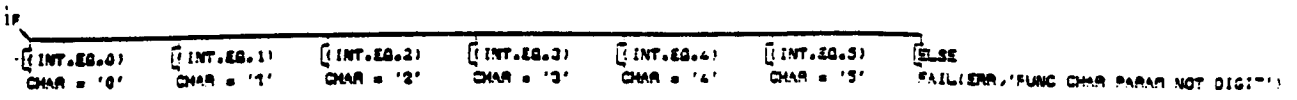


Figure 8

```
.PROG
.MONITOR SNAPS,PERFORMANCE,HISTORY,CONTROL
.TRACE
    .T1: DEP(20,9),DET(3) ,RF(1) .ET
    .T2: DEP(0,0),DET(10) .ET
    .T3: DEP(4,4),DET(3) .ET
.ENDTRACE
.SNAP-SHOT
    .SS1:  DET(20),FORMAT(200),SIZE(80)  .ESS
    .SS2:  DET(25),FORMAT(210),SIZE(80)  .ESS
    .SS3: DET(25),FORMAT(220),SIZE(80) .ESS
.ENDSNAP
.FILTERS
.BF1: (MOD(IT(N),2).EQ.0) .EBF
.ENDFILTERS
.ENDMONITOR
.MASTER
 .C DECLARATIONS
    INTEGER IN,EX,NUM,J,K
    INTEGER BIN(100)
    INTEGER PAGE(30,100)
    INTEGER ISTRNG(80)
    .EC
.T1: .BEGIN
.C INITIALISE
    .C I/O CHANNELS
        IN = 1
        EX = 1
        .EC
    .PARSEP
    .C CLEAR OUTPUT PAGE BUFFER TO SPACES
        .FOR J=1,30 .DO
            .FOR K=1,100 .DO
                PAGE(J,K) = ' '
            .ENDFR
        .ENDFR
        .EC
    .EC
 .C READ INPUT DATA INTO ARRAY BIN
    READ(IN,100)NUM,(BIN(J),J=1,NUM)
    .C FORCE LAST DATUM TO BE 9999 TO ENSURE TERMINATION
        BIN(NUM) = 9999
        .EC
    .EC
 .C CONVERT INPUT SET OF BIN NUMS TO DEC IN PAGE BUFFER
    J = 1
    .T1: .CYCLE K=1,100 .TILL(1) .DO
    .SS2: NUM,(BIN(J1),J1=1,NUM)
```

```
210   FORMAT('MEASURE OF SET',I4,1X,'MEMBERS',100(I10,1X))
            .EXITIF(BIN(J).EQ.9999).TOSITU(1)
            .T1: .C CONVERT JTH BIN TO DEC 2 WAYS,ONE PER COL
                    .T1: .CALL (1) B2DI(BIN(J),PAGE,J,1)
                .PARSEP
                    .T1: .CALL (1) B2DR(BIN(J),PAGE,J,30/2+1)
                .EC
            J = J+1
        .REPEAT
            .SITU(1)
                .ASSUMPTION INPUT 9999 FOUND
                    .OK
            .LIMIT
                .FAIL(EX,'9999 NOT FOUND & PAGE FULL')
        .ENDCY
        .EC
    .C OUTPUT CONTENTS OF PAGE BUFFER
        WRITE(EX,110)(BIN(K),(PAGE(J,K),J=1,30),K=1,NUM)
        .EC
    .STOP
    .C I/O FORMATS
100     FORMAT(I10)
110     FORMAT(I10,2X,30 A1)
        .EC
    .ENDM
    .LEVEL 1
    .SUBROUTINE B2DR(BINARY,PAGE,LINE,START)
    .C DECLARATIONS
        .C PARAMETERS
            INTEGER BINARY,PAGE(30,100),LINE,START
            .EC
        .C LOCALS
            INTEGER NCP,BIN
            .EC
        .C FUNCTIONS
            .IG NONE
            .EC
        .EC
    .BEGIN
    .C OUTPUT SIGN & MAGNITUDE OF BINARY
    .C OUTPUT SIGN OF BINARY
        .IF(BINARY.LT.0).THEN
            .CALL(3) PRINT('-',PAGE,LINE,START)
        .ELSE
            .CALL(3) PRINT('+',PAGE,LINE,START)
        .ENDIF
        .EC
    .PARSEP
    .C OUTPUT MAGNITUDE IN DECIMAL CHARSFROM MOST SIG DIGIT TO LEAST
        BIN = IABS(BINARY)
        NCP = START+1
        .T1: .CALL (2) PB2DR(BIN,PAGE,LINE,NCP)
        .EC
```

```
.EC
.RETURN
.END
.SETSEP
.SUBROUTINE B2DI(BINARY,PAGE,LINE,START)
.C DECLARATIONS
    .C PARAMETERS
        INTEGER BINARY,PAGE(30,100),LINE,START
        .EC
    .C LOCALS
        INTEGER NCP,BIN,I,REM
        .EC
    .C FUNCTIONS
        INTEGER DIGITS
        .EC
    .EC
.BEGIN
.C OUTPUT SIGN & MAGNITUDE OF BINARY
.C OUTPUT SIGN OF BINARY
    .IF(BINARY.LT.0).THEN
        .CALL(3) PRINT('-',PAGE,LINE,START)
    .ELSE
        .CALL(3) PRINT('+',PAGE,LINE,START)
    .ENDIF
    .EC
.PARSEP
.C OUTPUT MAGNITUDE IN DECIMAL CHARSFROM LEAST SIG DIGIT TO MOST
     BIN = IABS(BINARY)
    NCP = START+DIGITS(BIN)
    .CALL(2) PB2DI(BIN,PAGE,LINE,NCP)
    .EC
.EC
.RETURN
.END
.ENDLEV
.LEVEL 2
.SUBROUTINE PB2DR(BINARY,PAGE,LINE,NCP)
.C DECLARATIONS
    .C PARAMETERS
        INTEGER BINARY,PAGE(30,100),LINE,NCP
        .EC
    .C LOCALS
        INTEGER LSTDIG,BIN,ISTRNG(80),CHR
        .EC
    .C FUNCTIONS
        INTEGER CHAR
        .EC
    .EC
.T1: .BEGIN
.IF(BINARY.GE.10).THEN
    BIN = BINARY/10
    .T1: .CALL (*) PB2DR(BIN,PAGE,LINE,NCP)
.ELSE
```

```
        .NULL
    .ENDIF
    .C DETATCH LEAST SIG DIGIT
        LSTDIG = MOD(BINARY,10)
        .EC
    .C PRINT LEAST SIG DIGIT
        CHR = CHAR(LSTDIG)
        .SS1: CHR,LINE,NCP
        .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
        NCP = NCP+1
        .EC
200 FORMAT('CHARACTER',1X,A1,1X,'LINE',I4,1X,'COLUMN',I4)
    .RETURN
    .END
    .SETSEP
    .SUBROUTINE PB2DI(BINARY,PAGE,LINE,NCP)
    .C DECLARATIONS
        .C PARAMETERS
            INTEGER BINARY,PAGE(30,100),LINE,NCP
            .EC
        .C LOCALS
            INTEGER LSTDIG,J,BIN,ISTRNG(80),CHR
            .EC
        .C FUNCTIONS
            INTEGER CHAR
            .EC
        .EC
    .BEGIN
    BIN = BINARY
    .T2: .CYCLE J=1,30 .TILL(1) .DO
        .C DETATCH JTH LEAST SIG DIGIT
            LSTDIG = MOD(BIN,10)
            BIN = BIN/10
            .EC
        .T3: .C PRINT JTH LEAST SIG DIGIT
            CHR = CHAR(LSTDIG)
            .SS1: CHR,LINE,NCP
            .T2: .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
            NCP = NCP-1
            .EC
        .EXITIF(BIN.EQ.0).TOSITU(1)
    .REPEAT
        .SITU(1)
            .OK
        .LIMIT
            .FAIL(1,'PB2DI LOOP GUARD ERROR')
    .ENDCY
200 FORMAT('CHARACTER',1X,A1,1X,'LINE',I4,1X,'COLUMN',I4)
    .RETURN
    .END
    .ENDLEV
    .LEVEL 3
    .SUBROUTINE PRINT(CHCODE,PAGE,LINE,WIDPOS)
```

```
.C DECLARATIONS
    .C PARAMETERS
        INTEGER CHCODE,PAGE(30,100),LINE,WIDPOS
        .EC
    .EC
.T2: .BEGIN
.T2: PAGE(WIDPOS,LINE) = CHCODE
.RETURN
.END
.SETSEP
.INTEGER FUNCTION CHAR(INT)
.C DECLARATIONS
    .C PARAMETERS
        INTEGER INT
        .EC
    .EC
.BEGIN
.IF(INT.EQ.0).THEN
    CHAR = '0'
.ELIF(INT.EQ.1).THEN
    CHAR = '1'
.ELIF(INT.EQ.2).THEN
    CHAR = '2'
.ELIF(INT.EQ.3).THEN
    CHAR = '3'
.ELIF(INT.EQ.4).THEN
    CHAR = '4'
.ELIF(INT.EQ.5).THEN
    CHAR = '5'
.ELIF(INT.EQ.6).THEN
    CHAR = '6'
.ELIF(INT.EQ.7).THEN
    CHAR = '7'
.ELIF(INT.EQ.8).THEN
    CHAR = '8'
.ELIF(INT.EQ.9).THEN
    CHAR = '9'
.ELSE
    .FAIL(1,'FUNC CHAR PARAM NOT DIGIT')
.ENDIF
.RETURN
.END
.SETSEP
.INTEGER FUNCTION DIGITS(POSINT)
.C DECLARATIONS
    .C PARAMETERS
        INTEGER POSINT
        .EC
    .C FUNCTIONS
        INTEGER DIGTSR
        .EC
    .EC
.BEGIN
```

```
.IF(POSINT.GE.10).THEN
    DIGITS = 1+DIGTSR(POSINT/10)
.ELSE
    DIGITS = 1
.ENDIF
.RETURN
.END
.ENDLEV
.LEVEL 4
.INTEGER FUNCTION DIGTSR(INT)
INTEGER INT,DIGITS
.BEGIN
DIGTSR = DIGITS(INT)
.RETURN
.END
.ENDLEV
.ENDP
```

The type of run-time monitoring required is requested by the inclusion of a .MONITOR section at the head of the program as shown below.

```
.MONITOR PERFORMANCE,HISTORY,CONTROL,SNAPS
.TRACE
  .T1: DEP(20,7),DET(3),RF(1) .ET
  .T2: DEP(0,0),DET(10),RF(1) .ET
.ENDTRACE
.SNAP-SHOT
  .SS1: DET(20),FORMAT(200),SIZE(80) .ESS
  .SS2: DET(25),FORMAT(210),SIZE(80) .ESS
.ENDSNAP
.FILTERS
  .BF1: (MOD(IT(N),2).EQ.0) .EBF
.ENDFILTERS
.ENDMONITOR
```

The monitoring categories required are specified by the keywords following .MONITOR. The statement above represents all available monitoring categories. TRACE-ENDTRACE, SNAP-SHOT - ENDSNAP and FILTERS - ENDFILTERS are optional sections which control the details of monitoring.

4.1 Tracing

The most general form of trace statement possible is indicated above. The number following .T is the tracing level being defined. When statements in the program are tagged for monitoring, the tag includes a tracing level, which indexes a set of tracing parameters defined in .TRACE.

Tracing parameters are set as follows:

Detail level	DET(<number>)
Pruning data	DEP(<number>,<number>)
Repetition filter	RF(<number>)

The association of a level of detail with each statement tagged for tracing provides a mechanism whereby the user may select at run-time a subset of these statements to be actually traced. The user specifies at run-time a level of detail, and only those statements with

detail $\leq$ run-time detail

will appear in the tracing output.

The specification of a repetition filter is a method for selecting the iterations of a loop to be traced. The filter to be associated with a particular tracing level is defined by an index into the set of filters (specified in the .FILTERS section). Thus in the example above, the filter

```
(MOD(IT(N),2).EQ.0)
```

is associated with tracing level 1. Statements are only traced when the value of the associated filter is .TRUE. (and some other conditions are satisfied). Associated with each loop in the program is an iteration counter, assigned the value zero before the loop is entered and incremented by one at the start of each iteration of the loop. The function IT(N) is a function of type INTEGER*4 provided by the monitor which returns the current value of the iteration counter. Thus the repetition filter above will cause all odd numbered iterations not to be traced.

Note that any routine in which IT(N) is invoked must include a declaration:

```
INTEGER*4 IT
```

this is NOT included automatically by the translator. Failure to include such a declaration will lead to spurious results (the filter will always return the value 0).

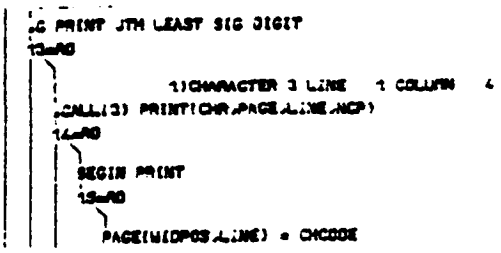
The pruning data specified by DEP allow the user to control the depth to which the trace is allowed to grow and how much of the trace is to be retained as levels of refinement are terminated.

The motivation behind this is that if the execution of a refinement terminates unexpectedly (for example a .FAIL statement is executed or the condition specified in a .ASSERTION statement yields the value .FALSE.) one can have a very detailed execution trace to work from. If the execution is successful, one is not in general interested in the detail of the execution so this can be discarded from the trace.

Consider the section of code:

```
.T3:  .C PRINT JTH LEAST SIG DIGIT
.T2:      CALL PRINT(CHAR(LSTDIG),PAGE,LINE,NCP)
      .EC

          ..........

.SUBROUTINE PRINT(CHCODE,PAGE,LINE,WIDPOS)
.T2: .BEGIN
.T2:  PAGE(WIDPOS,LINE) = CHCODE
      .RETURN
      .END
```

Execution of this code will produce the trace



Depths of refinement are indicated at the head of each refinement in the text <number>=RD, thus the statment 'BEGIN PRINT' is at refinement depth 9. Suppose the statment

.C PRINT JTH LEAST SIG DIGIT

is tagged with the tag .T3 where tracing level 3 is defined by:

.T3: DEP(2,1),DET(3) .ET

The first parameter specified in DEP (DOWN) is the number of refinements of the current refinement which are to be traced. This value is used to calculate a new value of the maximum depth of refinement to be traced (MDRT), which is given by:

MDRT = MAX(MDRT,MDRT + DOWN)

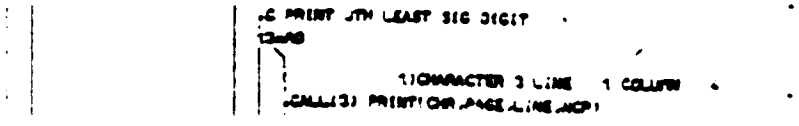
Thus in this example, since the value of DOWN associated with tracing level 2 is zero, the statements CALL PRINT and BEGIN PRINT would be traced, but the refinements of BEGIN PRINT would not.

The second parameter specified in DEP (UP) is the number of refinements of the current refinement to be saved. This value is used when a refinement terminates to decide whether the refinement should be saved in the tracing buffer or discarded. The maximum depth of refinement to be saved (MDRS)

is calculated as for MDRT, ie:

```
MDRS = MAX(MDRS,MDRS + UP)
```

Thus in this example, if MDRS has the value 7 when the statement PRINT JTH LEAST SIG DIGIT is obeyed, the new value calculated will be 8. Thus on termination of all refinements with depths greater than 8, any statements traced will be discarded. The above example will appear as:



4.2 Snap-shots

The SNAP-SHOT section specifies how current data values are to be recorded in the tracing buffers and the circular snap-shot buffers. The parameter specified by DET is the level of detail which controls whether anything will be recorded in the buffer. FORMAT specifies the label of a FORMAT statement which will be used as a template for writing the data values into the tracing buffer. SIZE specifies how many characters of the string produced by writing the values into a buffer according to FORMAT are to be saved. Snap-shots are invoked by a program statement of the form:

```
.SS<number>: <name list>
```

for example:

```
.SS1: I,J,K
```

<number> is an index into the set of snap-shot definitions in .SNAP-SHOT. Thus with the snap-shot definitions given earlier, 80 characters of the string produced by writing the values of I, J, K, into a buffer according to FORMAT statement 200 will be recorded in the tracing buffer if the detail level specified at run-time is greater than 20. Statement 200 might be:

```
200 FORMAT('VALUES OF I, J, K',2X,3(I6,2X))
```

Every routine producing snap-shots should include the declaration of an array:

```
INTEGER ISTRNG(<n>)
```

where the dimension <n> is large enough to hold the number of characters specified by the SIZE parameter of the .SS statement defining the snap-shot. Thus for a maximum SIZE of 80 characters, the array should be dimensioned at 40, as 2 characters are stored per word.

The run-time monitor will produce an output file containing snap-shot information in the format:

```
STATEMENT NUMBER          1
ENTRY     ITERATION AND SNAP-SHOT
   -2     (          1)MEASURE OF SET   3 MEMBERS      -123       5436
    9999
   -1     (          2)MEASURE OF SET   3 MEMBERS      -123       5436
    9999
    0     (          3)MEASURE OF SET   3 MEMBERS      -123       5436
    9999


STATEMENT NUMBER          2
ENTRY     ITERATION AND SNAP-SHOT
   -6     (          1)CHARACTER 1 LINE   1 COLUMN  17
   -5     (          1)CHARACTER 2 LINE   1 COLUMN  18
   -4     (          1)CHARACTER 3 LINE   1 COLUMN  19
   -3     (          2)CHARACTER 5 LINE   2 COLUMN  17
   -2     (          2)CHARACTER 4 LINE   2 COLUMN  18
   -1     (          2)CHARACTER 3 LINE   2 COLUMN  19
    0     (          2)CHARACTER 6 LINE   2 COLUMN  20


STATEMENT NUMBER          3
ENTRY     ITERATION AND SNAP-SHOT
   -6     (          1)CHARACTER 3 LINE   1 COLUMN   4
   -5     (          2)CHARACTER 2 LINE   1 COLUMN   3
   -4     (          3)CHARACTER 1 LINE   1 COLUMN   2
   -3     (          1)CHARACTER 6 LINE   2 COLUMN   5
   -2     (          2)CHARACTER 3 LINE   2 COLUMN   4
   -1     (          3)CHARACTER 4 LINE   2 COLUMN   3
    0     (          4)CHARACTER 5 LINE   2 COLUMN   2
```

ENTRY records the position of the snap-shot in the circular snap-shot buffers (least recent first). The value of the current iteration counter is recorded with each entry which is of value in tying up snap-shots and program execution.

If HISTORY tracing is invoked, snap-shots will also appear in the history trace file and will be displayed in context if a flowchart is drawn of this data. The mechanics of this operation are discussed in section 14 on running the flowchart generation program.

4.3 Tracing Modes

The tracing mechanism operates in one of two modes, either dumping information to the tracing file as it is generated (history buffer) or by maintaining a circular buffer of information and dumping the buffer when the monitor is terminated. The latter mode is useful during later stages of debugging as it produces significantly less output than the history buffer, typically giving an execution trace of the last 100 statements executed. Tracing mode is selected at run-time.

Tracing mode can also be changed dynamically at execution time by including statements

```
.HISTORICAL TRACE
.CIRCULAR TRACE
```

in the ROOTS source code. Execution of these statements switches the trace mode to history buffer or circular buffer respectively. Examples of traces generated in each mode are appended (see figures 11 and 12). Statements .HISTORICAL TRACE or .CIRCULAR TRACE were inserted after the statement:

```
J = J+1
```

in the main loop of the example program.

4.4 Control Tracing

If CONTROL monitoring is requested, the file produced by the monitoring package will contain information of the form:

CIRCULAR CONTROL BUFFER

EXEC. ORDER	STMT. NO.	ITERATION
-98	3	1
-97	4	1
-96	5	1
-95	6	1
-94	24	1
-93	25	1
-92	15	1
-91	16	1
-90	17	1
-89	18	1
-88	19	1
-87	24	1
-86	25	1
-85	20	1
-84	17	2
-83	18	2
-82	19	2
-81	24	2

-80	25	2
-79	20	2
-78	17	3
-77	18	3
-76	19	3
-75	24	3
-74	25	3
-73	20	3
-72	21	1
-71	22	1
-70	7	1
-69	8	1
-68	24	1
-67	25	1
-66	12	1
-65	13	1
-64	14	1
-63	13	1
-62	14	1
-61	13	1
-60	24	1
-59	25	1
-58	24	1
-57	25	1
-56	24	1
-55	25	1
-54	4	2
-53	5	2
-52	6	2
-51	24	2
-50	25	2
-49	15	2
-48	16	2
-47	17	1
-46	18	1
-45	19	1
-44	24	1
-43	25	1
-42	20	1
-41	17	2
-40	18	2
-39	19	2
-38	24	2
-37	25	2
-36	20	2
-35	17	3
-34	18	3
-33	19	3
-32	24	3
-31	25	3
-30	20	3
-29	17	4
-28	18	4

-27	19	4
-26	24	4
-25	25	4
-24	20	4
-23	21	2
-22	22	2
-21	7	2
-20	8	2
-19	24	2
-18	25	2
-17	12	2
-16	13	2
-15	14	2
-14	13	2
-13	14	2
-12	13	2
-11	14	2
-10	13	2
-9	24	2
-8	25	2
-7	24	2
-6	25	2
-5	24	2
-4	25	2
-3	24	2
-2	25	2
-1	9	1
0	10	1

The monitoring package stores the statement numbers and associated iteration counter of the last 100 statments executed by the program. Statement numbers can be ascertained from the flowchart of the program. Numbers used to identify performance, control and snap-shot data are given in parentheses after each tagged statement, for example:

```
CONVERT JTH BIN TO DEC 2 WAYS, ONE PER COL
(PF NO 3 CL NO 3)
```

The performance (PF NO) identifier of this statement is 3, the control statement number (CL NO) is also 3.

4.5 Performance Measurement

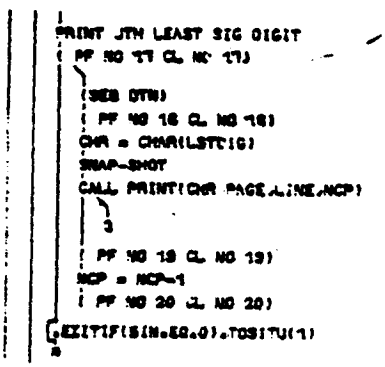
If PERFORMANCE monitoring is requested, the file produced by the monitoring package will contain information of the form:

PERFORMANCE MONITOR

STMNO	CPU-TIME	IO-TIME	FREQUENCY	MAX.REC.DEP	CURR.REC.DEP
1	855	1	1	1	0
2	799	1	1	1	0
3	793	1	1	1	0
4	780	1	2	1	0

5	409	0	2	1	0
6	407	0	2	1	0
7	363	1	2	1	0
8	360	1	2	1	0
9	3	0	1	1	0
10	0	0	1	1	0
11	0	0	0	0	0
12	332	1	2	1	0
13	326	1	7	4	0
14	189	1	5	3	0
15	359	0	2	1	0
16	347	0	2	1	0
17	322	0	7	1	0
18	199	0	7	1	0
19	43	0	7	1	0
20	3	0	7	1	0
21	6	0	2	1	0
22	0	0	2	1	0
23	0	0	0	0	0
24	58	0	18	1	0
25	10	0	18	1	0

Statement numbers can be related to statements as discussed above. PRIME routines CTIM$A and DTIM$A (see PRIME Manual for descriptions of these) are used to measure the cpu times and io times used by statements. CPU times and IO times are recorded in units of 1/100th of a second. Consider the program fragment:



The execution time for the statement PRINT JTH LEAST SIG DIGIT will be given by the difference in CPU time used at the start of the statement

.EXITIF(BIN.EQ.0).TOSITU(1)

and the CPU time used at the point before the statement

PRINT JTH LEAST SIG DIGIT

As far as possible resources used by subroutine calls to the monitor are subtracted from the CPU-TIME figures reported, but some proportion of the monitor overheads will be used, because some instructions must be executed before CPU time monitoring can be turned off. Thus the figures reported should be treated with some caution and fine grain effects should be ignored.

Monitor overhead is measured by the monitor and is printed out at the end of the program run (see attached example).

The column headed FREQUENCY records the execution frequencies of the statements monitored. MAX.REC.DEP reports the maximum depth of recursion of the statement and CURR.REC.DEP records the depth of recursion of the statement when monitoring terminated.

In general performance monitoring information will be displayed as a flowchart of the program under consideration, in preference to direct examination of the file produced by the monitor. The technique for doing this is described in the section on running the flowchart generation program. Bizarre results can be produced if statements tagged for tracing are not included on the flowchart drawn (ie are tagged to be abstracted).

5 GENERAL LAYOUT FEATURES OF ROOTS PROGRAMS

All reserved words in ROOTS are preceeded by the symbol '.'. ROOTS source text must start in or after column 7 and must not go beyond column 72 of the line. FORTRAN source may be intermingled with ROOTS constructs as desired, but only 'pure' FORTRAN source may be continued to a second or subsequent line.

If the FORTRAN code generated by a ROOTS statement exceeds 72 characters, it will be automatically continued to the next line. This should cause no problems.

ROOTS reserved words may not contain embedded spaces. Thus:

```
.I F(.TRUE.).T H E N
```

is illegal.

The ROOTS preprocessor generates labels in the range 20000 - 29999. Variable names KT<n> where <n> is in the range 0001 to 9999 are generated by the preprocssor in the translation of the .CYCLE statement and hence should not be used in the source fed to the preprocessor.

6 NOTATION

We introduce the notation :

<be>

to mean any valid FORTRAN boolean expression,and:

<ls>

to be any sequence of FORTRAN or ROOTS statements.

<text>

is any sequence of characters such as would appear in a FORTRAN comment statement,

<lines of text>

is any number of lines of <text>.

7 SELECTION STATEMENTS

7.1 .IF-.THEN-.ELIF-.ELSE

The syntax is:

```
.IF(<be>).THEN                    IF(.NOT.(<be>))GOTO20000

    <ls>                              <ls>

                                      GOTO 20001
.ELIF(<be>).THEN                  20000 IF(.NOT.(<be>))GOTO 20002

    <ls>                              <ls>

                                      GOTO 20001
.ELIF(<be>).THEN                  20002 IF(.NOT.(<be>))GOTO 20003

    <ls>                              <ls>

                                      GOTO 20001
.ELSE                             20003 CONTINUE

    <ls>                              <ls>

.ENDIF                            20001 CONTINUE
```

and generates the code shown. The .ELSE clause is mandatory. The statement may contain any number (including zero) of .ELIF clauses.

Further syntatic restrictions are:

(i) .THEN must appear at the end of a line

(ii) .IF-.THEN must appear on one line, continuation lines are not allowed.

(iii) .ELSE and .ENDIF must appear alone on separate lines.

The conditionals are evaluated serially until one is TRUE, the associated <ls> is obeyed and control passes to the end of the statement.

7.2 .SWITCH

In the syntax below <int> is an INTEGER variable, and <nc> is the number of cases in the statement. The syntax and code generated are:

```
                                    E.G. <no of cases>=3

.SWITCH(<int>,<nc>)                 IF(<int>.LT.1.OR.<int>.GT.<nc>)GOTO 20004
                                    GOTO(20001,20002,20003),<int>

.CASE(1)                    20001 CONTINUE

    <ls>                            <ls>

                                    GOTO 20005
.CASE(2)                    20002 CONTINUE

    <ls>                            <ls>

                                    GOTO 20005
.CASE(3)                    20003 CONTINUE

    <ls>                            <ls>

                                    GOTO 20005
.OUT-OF-RANGE               20004 CONTINUE

    <ls>                            <ls>

                                    GOTO 20005
.ENDSW                      20005 CONTINUE
```

The .OUT-OF-RANGE statement is mandatory.

8 ITERATION STATEMENTS

8.1 LOOPS

The language contains three loop statements, .FOR-.ENDFR is a simple iteration statement akin to the FORTRAN DO statement. .CYCLE is a general Zahn loop construct, .WHILE will be familiar from ALGOL.

8.1.1 .WHILE-.ENDWH

The syntax and code generated are:

```
.WHILE(<be>).DO                    20000 IF(.NOT.(<be>))GOTO 20001
    <ls>                                     <ls>
                                           GOTO 20000
.ENDWH                             20001 CONTINUE
```

Note that .WHILE (<be>) and .ENDWH must occupy single lines.

8.1.2 .FOR-.ENDFR

The syntax of this statement is:

```
.FOR I = N1,N2,N3 .DO                  DO 20000 I = N1,N2,N3
    <ls>                                     <ls>
.ENDFR                             20000 CONTINUE
```

8.1.3 .CYCLE-.ENDCY

All reserved words with the exception of .UNTIL in this statement are mandatory. In line with the syntax of .IF and .SWITCH statements, .LIMIT must be the last clause of the epilog before .ENDCY.

The .UNTIL construct has been introduced to provide a mechanism for representing the statements of which .EXITIF is a refinement. For example:

```
.UNTIL(End of file).IE
.UNTIL(Top bit set).IE
.EXITIF(REPLY.LT.0).TOSITU(2)
```

In the description below, <ns> is the number of distinct exits from the loop, ie. the number of .SITU clauses to follow.

```
.CYCLE I=N1,N2,N3 .TILL(<ns>).DO

    <ls>

.UNTIL(<text>).IE
.EXITIF(<be>).TOSITU(1)

    <ls>

.EXITIF(<be>).TOSITU(2)

    <ls>

.REPEAT

.SITU(1)

    <ls>

.SITU(2)

    <ls>

.LIMIT

    <ls>

.ENDCY
```

e.g. <ns>=2

```
      DO 20000 I = N1,N2,N3

           <ls>

           ASSIGN 20003 TO KT0001
      IF(<be>) GOTO 20002

           <ls>

      ASSIGN 20004 TO KT0001
      IF(<be>) GOTO 20002

           <ls>

20000 CONTINUE
      ASSIGN 20005 TO KT0001
20002 CONTINE
      GOTO KT0001

20003 CONTINUE

           <ls>

      GOTO 20001
20004 CONTINUE

           <ls>

      GOTO 20001
20005 CONTINUE

           <ls>

20001 CONTINUE
```

9 REFINEMENT STATEMENTS

9.1 Refinement

A refinement is introduced by either the statement

```
.C <text>
```

or the statements

```
.N
   <lines of text>
.EN
```

and is terminated by the statement

```
.EC
```

For example:

```
.C  ADVANCE OUTPUT BUFFER POINTER
    OBUFPR = OBUFPR + 1
.EC
```

Parallel sequences may be represented by the syntax:

```
.C                or .N   ....   .EN
    <ls>
.PARSEP
    <ls>
.EC
```

Any number of .PARSEP ... <ls> items are allowed.

9.2 .OK and .NULL

These statements have been found to be extremely useful in the language DRIL. The code generated by both will be CONTINUE.

9.3 .FAIL

The syntax of this statement and code generated are:

```
.FAIL(<channel number>,'<message>')
                                    WRITE(<channel number>,<label>)
                              <label> FORMAT('<message>')
                                      IF(.TRUE.)STOP
```

The last line is necessarily contorted to get round the problem of 'no path to statement' in the PRIME FORTRAN compiler.

10 SUBPROGRAM STATEMENTS

ROOTS recognises the statements

```
.BLOCK DATA
.SUBROUTINE
.FUNCTION
.INTEGER FUNCTION
.INTEGER*4 FUNCTION
.REAL FUNCTION
.DOUBLE PRECISION FUNCTION
.LOGICAL FUNCTION
.COMPLEX FUNCTION
.RETURN
.STOP
.END
```

Note that single spaces must appear in these statements where shown above.

The code generated by .STOP is:

```
IF(.TRUE.)STOP
```

To facilitate program monitoring ROOTS recognises statements of the form:

```
.CALL .....
```

Two forms of the statement are allowed:

```
.CALL <name>(<argument list>)
```

ie the addition of a period (.) to the front of a FORTRAN CALL statement. The second form allows the insertion of the routine level after CALL:

```
.CALL(<level>)<name>(<argument list>)
```

Recursive calls are denoted by the symbol '*' in place of <level>.

The notion of subroutine level comes from a consideration of call graphs, a routine at level n can be called by routines at all levels $m \leq n$ and may call routines at all levels $k \geq n$.

There is a need to delimit the MASTER segment (in 1900 terminology) of a program and hence we introduce:

```
.MASTER
.ENDM
```

It is also useful to know where the code of a routine begins and hence we introduce

```
.BEGIN
```

The statements

```
.PROG <text string>
.ENDP
```

delimit the entire source.

To facilitate the representation of the level concept in flowcharts, the statements:

```
.LEVEL <number>
.SETSEP
.ENDLEV
```

are introduced. The structure of a ROOTS program is then represented by the syntax:

```
.PROG
.MASTER
    .
    .
    .
.ENDM
.LEVEL 1
.SUBROUTINE
    .
    .
    .
.END
.SETSEP
.SUBROUTINE
    .
    .
    .
.END
.SETSEP
.SUBROUTINE
    .
    .
    .
.END
.ENDLEV
.LEVEL2
    .
    .
```

```
.ENDLEV
.ENDP
```

11 INPUT DIRECTION STATEMENTS

11.1 .ADD

It is a frequent source of annoyance that many software tools do not have an equivalent of the $INSERT Facility of the PRIME FORTRAN compiler. ROOTS has such a facility which will increase the utility of tools such as PFORT for investigation of the intermediate FORTRAN code generated by ROOTS.

The construct is:

.ADD <filename>	adds the text in <filename> and generates a flowchart of the entire source text, unless preceded by an edit tag (see below).

.ADD can be nested to a finite depth.

12 MONITOR STATEMENTS

12.1 .MONITOR

Monitoring levels are defined by statements enclosed by the reserved words:

```
.MONITOR
.ENDMONITOR
```

Trace, filter and snap-shot levels are defined within this section:

```
.MONITOR
.TRACE
    .T1: DEP(20,7),DET(3) ,RF(1) .ET
    .T2: DEP(0,0),DET(10),RF(1) .ET
.ENDTRACE
.SNAP-SHOT
    .SS1:  DET(20),FORMAT(200),SIZE(80)  .ESS
    .SS2:  DET(25),FORMAT(210),SIZE(80)  .ESS
```

```
.ENDSNAP
.FILTERS
.BF1: (MOD(IT(N),2).EQ.0) .EBF
.ENDFILTERS
.ENDMONITOR
```

Statements to be monitored are tagged as follows:

```
.T <level>:  <statement>
```

where <level> indexes a tracing level defined in the .TRACE section of the MONITOR statement.

Snap shots are produced by the statement:

```
.SS <index>:  <list of variables>
```

and <index> refers to a snap shot set up in the SNAP-SHOT section of the MONITOR statement.

To invoke monitoring, the .MONITOR statement must be followed by keywords (in any order) indicating the monitoring categories required. The most general form is:

```
.MONITOR PERFORMANCE,HISTORY,CONTROL,SNAPS
```

12.2 ASSUMPTION

The syntax of this statement is:

```
.ASSUMPTION <number>: (<text>)
```

the code generated is as for a comment statement.

12.3 .ASSERTION

The syntax of this statement is:

```
.ASSERTION <number>: (<be>)
```

If the statement is <u>not</u> tagged by a monitoring tag (eg .T1:) the code generated is as for a comment statement.

If the statement is tagged, for example:

```
.T1: .ASSERTION 1: (N.GT.M)
```

then the code generated is:

```
        IF(<be>) GOTO 20000
            STOP <number>
20000 CONTINUE
```

13 EDITING STATEMENTS

It is often the case that one does not wish to produce the entire flowchart of a program, merely the sections of current interest. An editing mechanism is defined within the syntax.

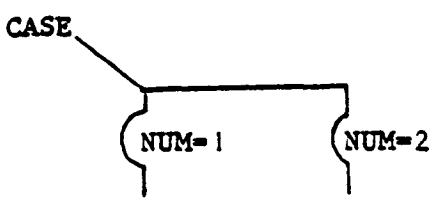
ROOTS statements may be tagged to indicate that the refinement of the statement is not to be drawn by tagging them with .K <number>: for example:

```
.SWITCH (NUM,3)
.K 1:  .CASE(1)
.K 2:  .CASE(2)
```

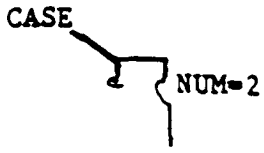
Each tag .K <number> may take the value REF (refined) meaning draw the sequence, or ABS (abstracted) meaning omit the sequence. Values are assigned within the .EDIT statement which must appear after the .PROG statement and before the tags defined are used. The maximum number of tags that may be defined is 20. The syntax of the statement is:

```
.EDIT
  .K <number>:(REF or ABS) .EK
  .K <number>:(REF or ABS) .EK
          ........
.ENDEDIT
```

For example, if K1 and K2 are defined as REF, the program fragment above would be drawn as:



If however K1 is defined as ABS, the fragment will be drawn as:



14 COMPLETE EXAMPLES

14.1 Running the Preprocessor

This section describes the operation of the ROOTS preprocessor. ROOTS is invoked by the macro command:

$ROOTS <input file>,<code file>,<flowchart file>,<options>

where:

<input file> is the name of the file containing ROOTS source

<code file> is the name of a file into which the FORTRAN code generated by the preprocessor will be written

<flowchart file> is the name of a file into which the code to drive the flowchart generation program will be written

<options> specify whether FORTRAN and/or flowchart code is to be produced

FORTRAN code is generated by specifying the boolean FORT. Flowchart code is generated by specifying the boolean FLOW.

Default names of

C-<input file>

F-<input file>

are provided for <code file> and <flowchart file> respectively.

Thus, for example if a ROOTS program is contained in file SOURCE, FORTRAN and flowchart code can be generated by the command:

```
    $ROOTS SOURCE,,,FORT,FLOW

The FORTRAN code will be written in file C-SOURCE,
the flowcharting code in file F-SOURCE.
See the terminal dialogue below (commands typed by user underlined,
text following a '['
is an explanatory comment).

OK, $ROOTS SOURCE...FORT,FLOW

OK, I SOURCE
OK, O C-SOURCE 3 2
OK, O F-SOURCE 4 2
OK, SEG PROGRAM>#ROOTSTRANS
GO
FOREST V 2.2
DO YOU WANT FLOWCHART CODE (Y/N)
Y
DO YOU WANT FORTRAN CODE (Y/N)
Y
MAIN
B2DR
B2DI
PB2DR
PB2DI
PRINT
CHAR
DIGITS
DIGTSR
 *** OK ***

**** STOP

OK, CO TTY
```

14.2 Invocation of the run-time monitor

Run-time monitoring code is inserted in the FORTRAN code generated by the ROOTS preprocessor by the insertion of a .MONITOR statement as described above. The FORTRAN code should be compiled in the normal way, but when the program is loaded, commands:

```
    LO PROGRAM>B_ROOTSMON
    LI VFLIB
    LI VAPPLB
```

should be included. To continue the example above:

```
OK, FTN C-SOURCE -DYNM        [ -DYNM is used because source contains
                                  [ recursive routines
GO
0000 ERRORS [<.MAIN.>FTN-REV15.2]
0000 ERRORS [<B2DR  >FTN-REV15.2]
0000 ERRORS [<B2DI  >FTN-REV15.2]
0000 ERRORS [<PB2DR >FTN-REV15.2]
0000 ERRORS [<PB2DI >FTN-REV15.2]
0000 ERRORS [<PRINT >FTN-REV15.2]
0000 ERRORS [<CHAR  >FTN-REV15.2]
0000 ERRORS [<DIGITS>FTN-REV15.2]
0000 ERRORS [<DIGTSR>FTN-REV15.2]

OK, SEG
GO
# VLOAD #C-SOURCE
$ LO B C-SOURCE
$ LO PROGRAM>B_ROOTSMON
$ LI VFLIB
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ SA
$ QU
```

When the program is run, a dialogue will be initiated by the run-time monitor as shown below:

```
OK, SEG #C-SOURCE
GO
BUFFER TYPE(1=HIST,2=CIRC)
1
PLEASE INPUT CHANNEL NO FOR OUTPUT
11
PLEASE INPUT PERFORMANCE FILENAME
SOURCE-P
PLEASE INPUT CONTROL PATHS FILENAME
SOURCE-C
PLEASE INPUT SNAP-SHOT FILENAME
SOURCE-S
PLEASE INPUT TRACE FILENAME
SOURCE-T
TRACE ACTIVE?(T OR F)
T               [ TRACE is suppressed by replying F
PLEASE INPUT LEVEL OF DETAIL OF TRACE
30              [ sets level of detail for this run
PLEASE INPUT MAXIMUM REFINEMENT TRACED
```

```
20              [refinements deeper than this are not traced
PLEASE INPUT LENGTH OF STATEMENT
IN CHARACTERS REQUIRED IN TRACE
80              [the first 80 characters of each statement
                    [ traced are recorded in the buffer
3               [ remaining input typed is data for program
-123
5436
1
      -123  -123           -123
      5436  +5436          +5436
      9999
                  [ output below is summary of resources
                  [ used by monitor itself
PERFORMANCE MONITOR FOR MONITORING SUBROUTINE
TOTAL CPU-TIME=          980
TOTAL I/O-TIME=            2
FREQUENCY=          995

**** STOP

OK,
```

14.3 Production of Flowcharts from run-time output

This section describes, by way of the above example the production of flowcharts from the flowchart code files produced by the ROOTS preprocessor and the run-time monitor.

The example below shows the commands necessary to display the run-time performance and snap-shot data on a flowchart of the program source. The detailed description of the parameters to the flowchart program is deferred to a later section.

The flowchart generation program is invoked by the command:

```
SEG PROGRAM>#ROOTSFLOW
```

The user is then asked to specify those parameters for this task which differ from the default parameters set up by the flowchart generation program, as shown below:

```
OK, SEG PROGRAM >#ROOTSFLOW
GO
DIMENSIONAL FLOWCHARTING PROGRAM, VERSION RECURS 7.0
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
Y
```

PARAM WANTED	DEFAULT	INPUT	!	PARAM WANTED	DEFAULT	INPUT
ACTDEL	"	1	!	FRAME WIDTH	6000	14
CONDEL	?	2	!	FRAME HEIGHT	6000	15
REFBEG	!	3	!	CHAR WIDTH	6	16
ABSBEG		4	!	CHAR HEIGHT	4	17
PARBEG	=	5	!	SPACING	2	18
REFEND	#	6	!	DIAG	8	19
RPTEND	*	7	!	DTB	3	20
PAREND	#	8	!	DMLP	40	21
DRWDYN	+	9	!	DEVICE	PLOT	22
NOTDYN	-	10	!	TRACE LEVEL	0	23
FOREST	T	11	!	DYNAMIC INFO	F	24
STMCUT	25	12	!	GRID WANTED	F	25
FRMSIZ	100	13	!	DIVISIONS	10	26
INFILE	INFOFC					27
DYNFIL	DYNFO					28
OUTFIL	FLOWIT					29
DEBUG	DEBUG					30
SNPFIL	SNAPS					31
PLTFIL	BPLOTTER					32
SNPSHT	F	33	!	NO OF DYNAMS	2	34

DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
12
WHAT DO YOU WANT TO USE INSTEAD ? -I4-
100
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
27
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
F-SOURCE [the flowchart code from ROOTS
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
28
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
SOURCE-P [run-time performance file
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
31
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
SOURCE-S [run-time snap-shots file
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-

```
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
24     [ flag to display performance data
WHAT DO YOU WANT TO USE INSTEAD ? -L1-
T
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
33     [ flag to display snap-shots
WHAT DO YOU WANT TO USE INSTEAD ? -L1-
T
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
32
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
P-SOURCE       [ file to contain plotter code
                   [ for Benson 1302 - default output device
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
Y
```

PARAM WANTED	DEFAULT	INPUT	!	PARAM WANTED	DEFAULT	INPUT
ACTDEL	"	1	!	FRAME WIDTH	6000	14
CONDEL	?	2	!	FRAME HEIGHT	6000	15
REFBEG	!	3	!	CHAR WIDTH	6	16
ABSBEG		4	!	CHAR HEIGHT	4	17
PARBEG	=	5	!	SPACING	2	18
REFEND	#	6	!	DIAG	8	19
RPTEND	*	7	!	DTB	3	20
PAREND	#	8	!	DMLP	40	21
DRWDYN	+	9	!	DEVICE	PLOT	22
NOTDYN	-	10	!	TRACE LEVEL	0	23
FOREST	T	11	!	DYNAMIC INFO	T	24
STMCUT	100	12	!	GRID WANTED	F	25
FRMSIZ	100	13	!	DIVISIONS	10	26
INFILE	F-SOURCE					27
DYNFIL	SOURCE-P					28
OUTFIL	FLOWIT					29
DEBUG	DEBUG					30
SNPFIL	SOURCE-S					31
PLTFIL	P-SOURCE					32
SNPSHT	T	33	!	NO OF DYNAMS	2	34

```
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
N
O.K.
"ADVANCE FRAMES       0"-
DO YOU WANT THE HISTOGRAMS?(Y OR N -A1-)
```

```
N
"THE TOTAL CPU TIME USED BY THIS PROGRAM =     1872
THE TOTAL I/O TIME USED BY THIS PROGRAM =      106
 IN HUNDREDTHS OF A SECOND
THE NUMBER OF RECORDS READ WAS :      354
THE CHAR POSN ON THE RECORD WAS :       27
THE NUMBER OF CHARACTERS WAS    :    42714"-

**** STOP

OK,
```

The output from the flowchart may be sent to the Benson plotter by the command

```
PLOT P-SOURCE
```

The next example describes the commands necessary to plot the run-time trace of the program as a flowchart on the Benson plotter.

```
OK, SEG PROGRAM>#ROOTSFLOW
GO
DIMENSIONAL FLOWCHARTING PROGRAM, VERSION RECURS 7.0
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
12
WHAT DO YOU WANT TO USE INSTEAD ? -I4-
100
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
27
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
SOURCE-T          [ trace file produced by monitor
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
32
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
P-SOURCE-TRACE     [ plotter code output file
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
```

```
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
34      [ it is imperative this parameter be changed
              [ the program will fail illegal dyn-char if this
              [ is omitted
WHAT DO YOU WANT TO USE INSTEAD ? -I4-
1
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
N
O.K.
"ADVANCE FRAMES        0"-
"ADVANCE FRAMES        1"-
DO YOU WANT THE HISTOGRAMS?(Y OR N -A1-)
N
"THE TOTAL CPU TIME USED BY THIS PROGRAM =    2292
THE TOTAL I/O TIME USED BY THIS PROGRAM =      15
 IN HUNDREDTHS OF A SECOND
THE NUMBER OF RECORDS READ WAS :      465
THE CHAR POSN ON THE RECORD WAS :      13
THE NUMBER OF CHARACTERS WAS    :   56146"-

**** STOP

OK,
```

15 THE FLOWCHART GENERATION PROGRAM

This section gives a complete description of the program for drawing Dimensional Flowcharts. The program is used to draw a dimensional flowchart on the TEKTRONIX 4010 and 4014, the line-printer, the Benson 1302 plotter or the FR80 microfilm recorder. The input file contains flowcharting code as generated by the FORTRAN preprocesser FOREST.

The program may be loaded by:

```
SEG  PROGRAM>#ROOTSFLOW
```

The program then asks the user if he wishes to see the parameter table. If the answer 'Y' is given to this question, the current parameter table will be displayed. Parameters may be changed as shown below. After each parameter change the user is given the opportunity to display the current parameter table.

Parameters may be changed as follows (the characters in brackets indicate the format of the response):

```
OK, SEG PROGRAM>µROOTSFLOW
GO
DIMENSIONAL FLOWCHARTING PROGRAM, VERSION RECURS 7.0
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
Y
```

PARAM WANTED	DEFAULT	INPUT	!	PARAM WANTED	DEFAULT	INPUT
ACTDEL	"	1	!	FRAME WIDTH	6000	14
CONDEL	?	2	!	FRAME HEIGHT	6000	15
REFBEG	!	3	!	CHAR WIDTH	6	16
ABSBEG	@	4	!	CHAR HEIGHT	4	17
PARBEG	=	5	!	SPACING	2	18
REFEND	#	6	!	DIAG	8	19
RPTEND	*	7	!	DTB	3	20
PAREND	#	8	!	DMLP	40	21
DRWDYN	+	9	!	DEVICE	PLOT	22
NOTDYN	-	10	!	TRACE LEVEL	0	23
FOREST	T	11	!	DYNAMIC INFO	F	24
STMCUT	25	12	!	GRID WANTED	F	25
FRMSIZ	100	13	!	DIVISIONS	10	26
INFILE	INFOFC					27
DYNFIL	DYNFO					28
OUTFIL	FLOWIT					29
DEBUG	DEBUG					30
SNPFIL	SNAPS					31
PLTFIL	BPLOTTER					32
SNPSHT	F	33	!	NO OF DYNAMS	2	34

```
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N)- -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
27
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
TESTFC
```

Once all the parameters are correct,type 'N' to the question 'DO YOU WISH TO CHANGE ANY OF THESE?' The flowchart will then be drawn in vertical strips for the appropriate device.Each of these strips is made up of a number of frames.

The program automatically collects data about the frequency of use of various flowcharting constructs such as the number of characters per statement and the number of statements per refinement.These histograms give a clue to the complexity of the program and areas of poor structure. When the program has completed the flowchart,or an error has occurred,the question

'DO YOU WANT THE HISTOGRAMS?'

is asked.If the histograms are required they will be written to the output file.

16 PARAMETER VALUES

16.1 Explanation of parameter table

(1) 'ACTDEL' - Action statement delimiter.
(2) 'CONDEL' - Conditional statement delimiter.
(3) 'REFBEG' - Beginning of refined sequence.
(4) 'ABSBEG' - Beginning of abstracted sequence.
(5) 'PARBEG' - Beginning of parallel sequence.
(6) 'REFEND' - Refined sequence terminator.
(7) 'RPTEND' - Repeated sequence terminator.
(8) 'PAREND' - Parallel sequence terminator.
(9) 'DRWDYN' - Dynamics information is required.
(10) 'NOTDYN' - Dynamics information is not required.
(11) 'FOREST'* - The above parameters are FOREST by default. For DRIL input this is set to false and all parameters are automatically set to their DRIL equivalents.
(12) 'STMCUT'* - Number of characters appearing for each statement.This should be given some large value , say 100 , if all the code is wanted. The shortened code lines produce a general outline of the structure of the flowchart.
(13) 'FRMSIZ'* - The coordinate system in which the grid will be drawn.This parameter has no effect on the size of the flowchart,but determines the scale of the grid i.e. the first frame will have top left-hand corner (0,0) and bottom right-hand corner (FRMSIZ,FRMSIZ). This means that large flowcharts need not have excessive scales.
(14) 'FRAME WIDTH' - Width of one output frame on selected device.
(15) 'FRAME HEIGHT' - Height of one frame.
(16) 'CHAR WIDTH' - Width of one output character.
(17) 'CHAR HEIGHT' - Height of half a character.
(18) 'SPACING' - Width of inter-character spacing.
(19) 'DIAG' - Length of diagonal refinement line.
(20) 'DTB' - Length of vertical line between consecutive statements.
(21) 'DMLP' - Gap between parallel sequences.
(22) 'DEVICE'** - Device on which flowchart is to be drawn. On inputting this parameter number the following list is given showing which value to input:

WHAT DO YOU WANT TO USE INSTEAD ? (I1)

DEVICE	INPUT
TEK 4010	1
TEK 4014	2
FR80	3
L.P.	4
BEN PLOTTER	5

(23) 'TRACE LEVEL'* - The depth of trace desired for the program e.g.
0=None.
10=Basic outline i.e. coordinates of frame.
20=Entry to and exit from the recursive subroutines which draw the sequences.
30=Information about each sequence i.e. refined or parallel,abstracted, intersecting or windowed out, repeated or earthed, sequence number and sequence table.
40=Statement level i.e. action or conditional.
50=Statement details.
80=Lines - diagonal,vertical and horizontal.
90=Next non-blank character on input, statistics updating.
100=Every character from input file, coordinates for writing text.
120=The function MAX,each line of input.
The tracing information will be written into the debug file which may then be passed through this program to produce a 3-d trace.

(24) 'DYNAMIC INFO' - This must be set to true if run-time performance analysis is wanted.

(25) 'GRID WANTED'* - It is possible to draw a grid around the flowchart by setting this value to true.

(26) 'DIVISIONS'* - Number of divisions in the grid for each frame.

(27) 'INFILE'** - Input file containing flowcharting code. The filename may be up to 32 characters long,and may include treenames.

(28) 'DYNFIL' - File for performance information as produced by the monitor The file will not be opened unless no.24 is set to true.

(29) 'OUTFIL'** - File in which flowchart for the line-printer and the histograms for all devices (if required) will appear.

(30) 'DEBUG'* - The debug file will contain the last piece of input and any error message produced. It will also hold the tracing output.

(31) 'SNAPS'* - File containing snap-shot values output by the monitor

(32) 'PLTFIL'* - File for plotter orders if output device is Benson plotter (5)

(33) 'SNPSHT'* - Set to true if snap-shots are to be displayed on the flowchart

(34) 'NO OF DYNAMS' - Set to 1 if drawing flowchart of runtime execution trace generated by monitor.

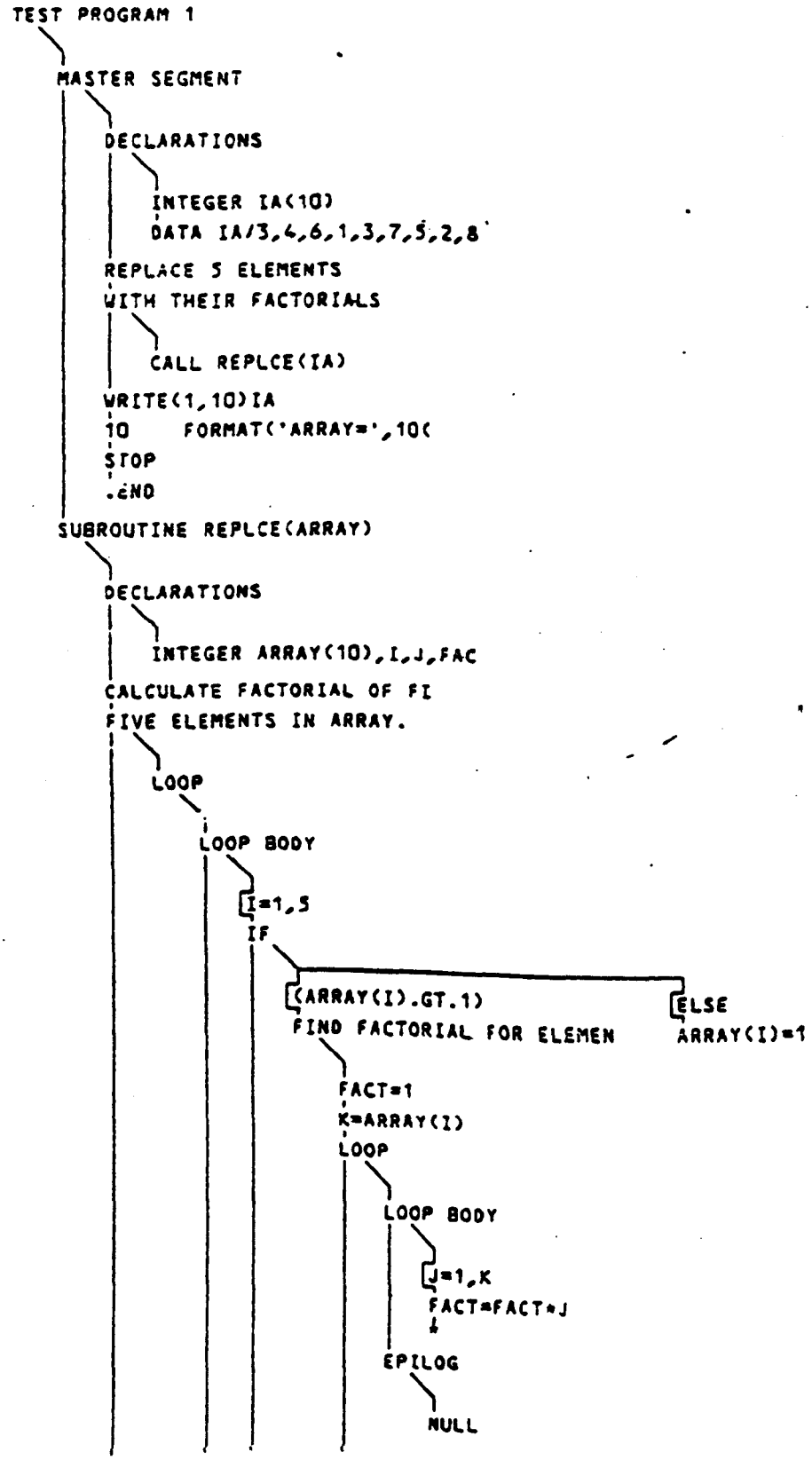
* - may be changed

** - most often changed

16.2 Example of use of grid and statement cut off

1)DEFAULT SIZE
i.e. STMCUT=25,FRMSIZ=100,DIVISIONS=10

```
10.        20.        30.        40.        50.        60.        70.        80.
```



2)STMCUT=100,FRMSIZ=250,DIVISIONS=25

```
30.  40.  50.  60.  70.  80.  90. 100. 110. 120. 130. 140. 150. 160. 170. 180. 190. 200. 210. 220
```

```
TEST PROGRAM 1
    MASTER SEGMENT
        DECLARATIONS
            INTEGER IA(10)
            DATA IA/3,4,6,1,3,7,5,2,8,9/
        REPLACE 5 ELEMENTS
        WITH THEIR FACTORIALS
            CALL REPLCE(IA)
        WRITE(1,10)IA
        10    FORMAT('ARRAY=',10(I6,','))
        STOP
        .END
    SUBROUTINE REPLCE(ARRAY)
        DECLARATIONS
            INTEGER ARRAY(10),I,J,FACT,K
        CALCULATE FACTORIAL OF FIRST
        FIVE ELEMENTS IN ARRAY.
            LOOP
                LOOP BODY
                    I=1,5
                    IF
                        (ARRAY(I).GT.1)                    ELSE
                        FIND FACTORIAL FOR ELEMENT I       ARRAY(I)=1
                            FACT=1
                            K=ARRAY(I)
                            LOOP
                                LOOP BODY
                                    J=1,K
                                    FACT=FACT*J
                                EPILOG
                                    NULL
```

17 ACTION ACCORDING TO DEVICE

17.1 Tektronix 4010 AND 4014

To draw a flowchart on the TEKTRONIX devices,first select the correct device number in the parameter table (NO.22 - 1 for TEK 4010 and 2 for TEK 4014).Other parameters which may need changing are the grid decision,since this will be necessary to help with focussing,and the statement cut off length.

The frames will be drawn starting.with the top·left-hand corner and then moving down the flowchart to the bottom.If the flowchart is wider than the screen a number of these vertical strips will be drawn. The pause after the first frame is drawn will always be longer than any other since the program must process the whole code at this time.

After each frame there is a choice of three possible alternatives:

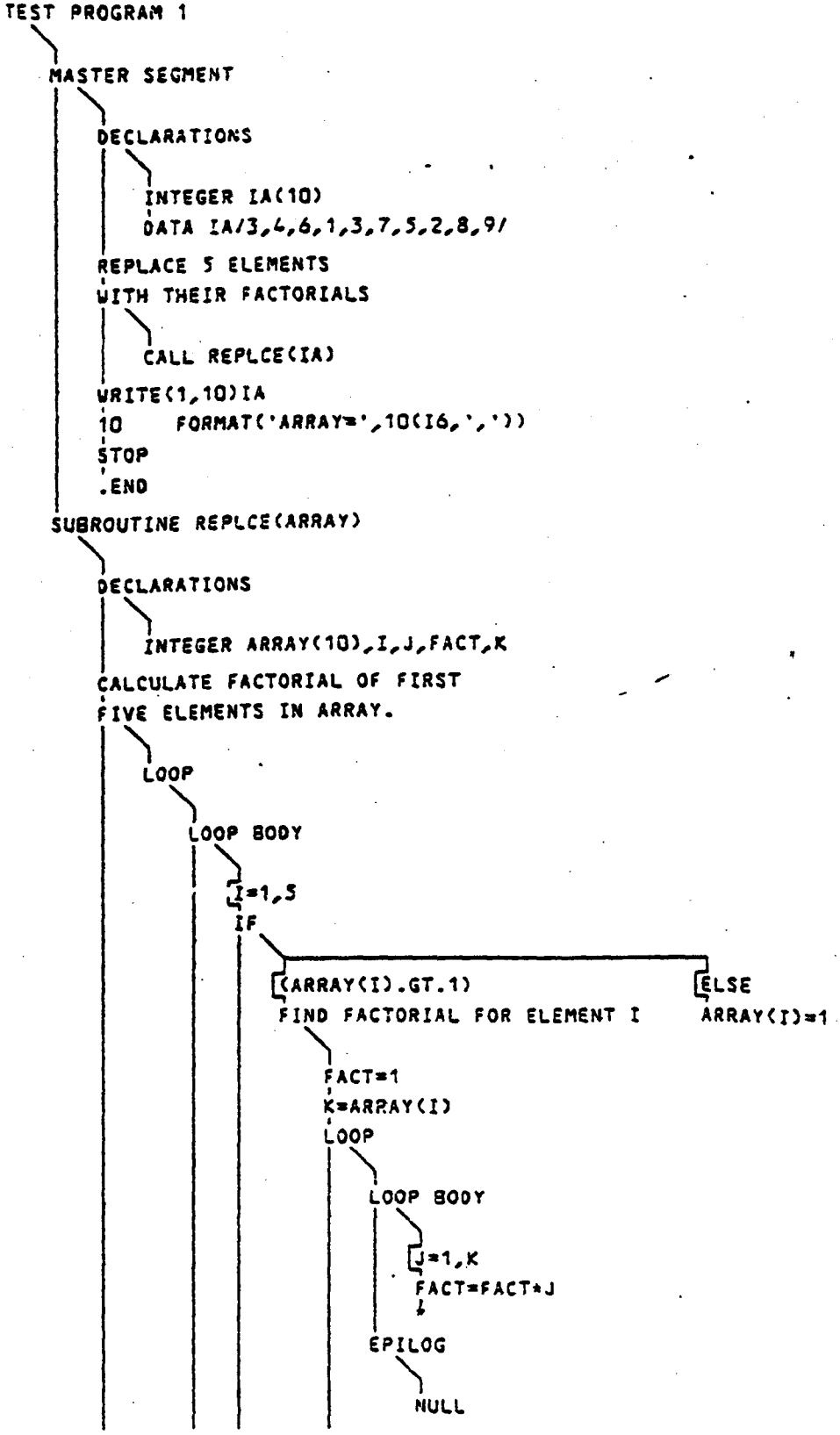
(i) To go on to the next frame in the sequence - type 'Y' to the question 'DO YOU WANT TO GO ON TO THE NEXT FRAME?'.

(ii) To break - type 'N' to the first question and then 'B' to the question: 'YOU CAN EITHER BREAK(B).OR FOCUS ON A SEQUENCE(F)'

(iii) To focus - type 'N' to the first and 'F' to the second question.Focussing means redrawing one frame from a user specified point.This point may be anywhere within the limits of the flowchart and need not necessarily be inside the frame which has just been drawn.The x- and y-coordinates of the desired point are input in the scale of the selected grid. It is possible to focus on any number of points and still return to the original position by typing 'Y' to the question 'DO YOU WANT TO GO ON TO THE NEXT FRAME?'.

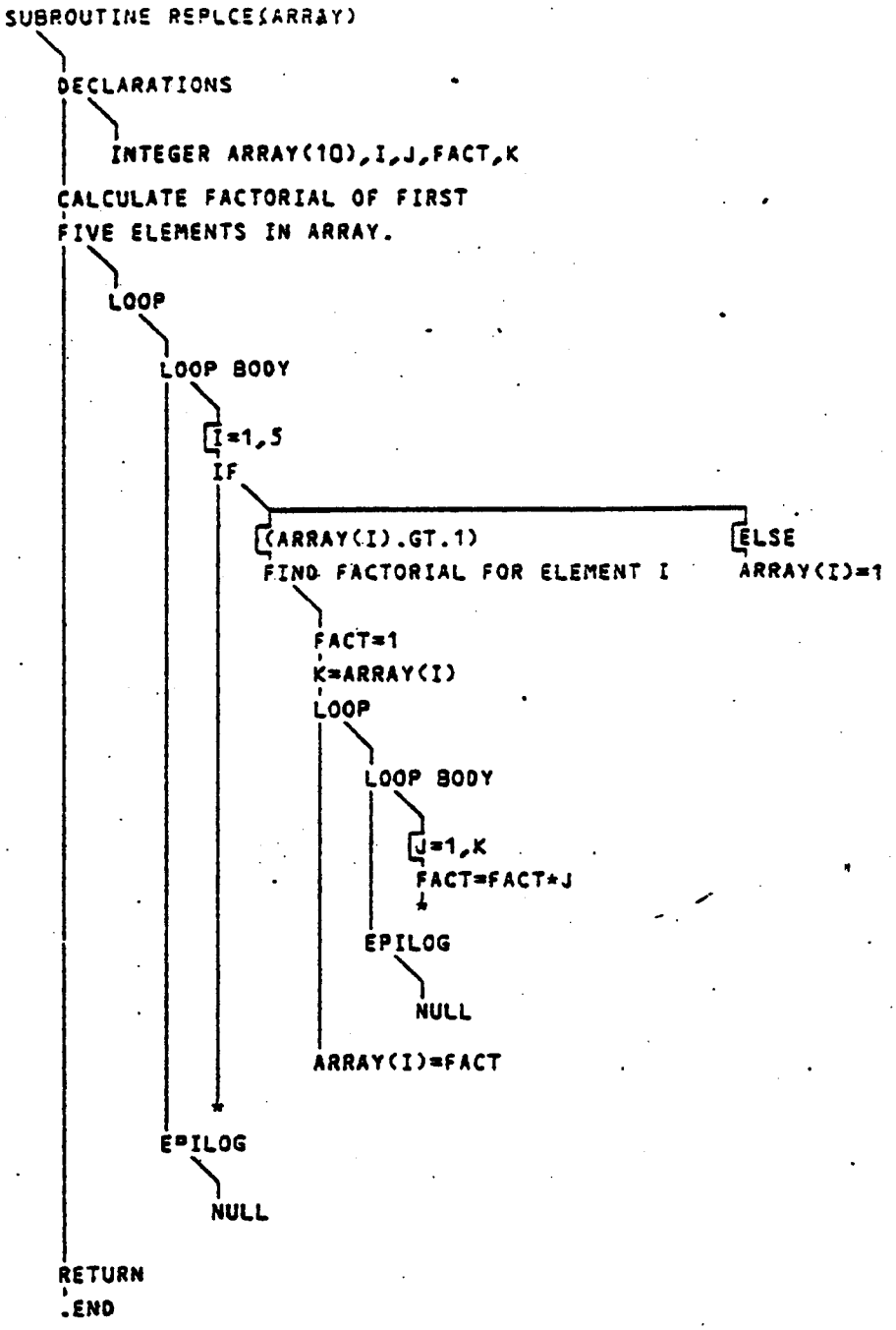
Example of focussing
1)FIRST FRAME

```
20.   30.   40.   50.   60.   70.   80.   90.   100.  110.  120.  130.  140.  150.  160.  170.
```



```
DO YOU WISH TO GO ON TO THE NEXT FRAME?(Y OR N -A1-)  N
YOU CAN EITHER BREAK(B),OR FOCUS ON A SEQUENCE(F)(-A1-)  F
PLEASE TYPE X-COORDINATE,MAX RIGHT=   126 (-I12-)  0
PLEASE TYPE Y-COORDINATE,MAX BOTTOM=   250 (-I12-)  85
```

2)FOCUSSED FRAME



When the flowchart has been drawn,or the user has broken in(by typing 'B'),the program asks 'DO YOU WANT THE HISTOGRAMS?'. If the response is 'Y' the histograms will be written to the output file.

17.2 FR80

To produce output for the FR80 choose device number 3 (parameter NO. 22),decide on the statement cut off (which is usually enough to give the whole code,since the FR80 output is smaller).It is also possible to draw a grid around the flowchart by setting parameter NO. 25 to true.

During the program run the number of frames being drawn is written to the screen(i.e. 'ADVANCE FRAMES 0')to give an indication of how long the program is taking , and the size of the finished flowchart.These numbers start at zero, the largest number advanced showing how long the flowchart will be,and the number of times this figure returns to zero saying how many strips will have been drawn.

When the program has finished and asked whether the histograms are required,the FR80 output will be found in the file FR80JB.This must be sent accross to the FR80 via the IBM 360/195.

E.G. HASP
GO
REV 15.02.C0
INIT DONE

>SEND FR80JB
>QU

If the flowchart is wider than one strip, the FR80 hard copy,which appears in a continuous strip,should be cut along the cut lines and joined together.

17.3 Line-printer

The device number for the line-printer is 4.The program will advance frames similar to the FR80 and the flowchart,which will be written to the output file,must be spooled with the FORTRAN control option.

17.4 Benson plotter

The device number for the Benson 1302 plotter is 5. The program will advance frames similar to the FR80. To plot the output on the Benson, issue the command:

PLOT <PLTFIL>

18 ERROR MESSAGES

18.1 Errors in input/output:

```
ERR = (I3) SEE PRIME USER NOTE 13- Error in opening file.
           IERR = 0 O.K.
                  1 Illegal unit
                  2 Illegal file name
                  3 Illegal key
                  4 File does not exist
                  5 File protected
                  6 File in use
                  7 File exists
                  8 File directory full
                  9 Disc full
                 99 Unusual error
ERR-NO.(I3)IN CLOSING INPUT FILE,SEE PRIME USER NOTE 13
ERROR IN CLOSING I/O FILES
           IERR = 0 O.K.
                  1 Illegal unit
                 10 Unit not open
                 99 Unusual error
ERR-IN RDLIN$, NO.=(I3) ACTREC=(I8)- Error in reading in a
  line of input.
ERR-GET NON BLANK CHARACTER (A1)- Null line in input.
ERR-E.O.F. REACHED IN READCH- End of file has been reached.
  Could be due to mismatched !'s and #'s.
```

18.2 Errors in flowcharting code:

```
ERR-BAD STATMT DELIM (A1)- Illegal character at beginning of
  statement.
ERR-IN SKIP SEQUENCE (A1)(I4)- This character is neither
  a refinement controller or statement delimiter.
ERR-BAD DYN CHAR=(A1)- Illegal character after statement
  i .e. not DRWDYN or NOTDYN.
ERR-MAX CHARS PER STATMT (A1)- More then 999 characters
  in one statement.
ERR-BAD 1ST CHAR IN DYN DATA REC- Dynamics information does
  not begin with a left bracket.
ERR-BAD SEQ TERM IN WINDOW(A1)(I8)- Sequence terminator is
  neither REFEND,PAREND or RPTEND.
ERR-MAX STMTS PER SEQ- More than 100 statements in one sequence.
ERR-MAX SEQS PER REF- More than 100 sequences in one refinement.
ERR-SEQUENCE TABLE OVERFLOW- More than 750 sequences.
ERR-MORE THAN 5 DIGITS IN NO.- Scale on grid has grown too
  large.Redraw flowchart with smaller FRMSIZ or without grid.
```

8.3

o simulate recursion a stack is used in calling the subroutines which aw the sequences.These may not recurse to a depth of more than 200.If his happens the following messages will be given:
RR-STACK OFLO IN CALLING DWSEQN
RR-STACK OFLO IN CALLING DSEQN
RR-STACK OFLO IN CALLING COMPND

8.4

he following error messages arise in the event of an internal error being etected by the program. Occurrence of such error messages should be eported to the authors of this program,after careful checking of the nput data to the program.
RR-MYSTERIOUS ERROR IN READCH
RR-CH OVER PAGE ARRAY BOUND(15I8)
RR-MYSTERIOUS ERROR IN DSTATM
RR-CHSIZE
RR-HISTGM LOOP GUARD
RR-FLNAME CONTAINS 34 CHARS NON-BLANK

A common error,which is not detected by the program,but which results the bottom of the flowchart being lost,is caused by mismatched !'s and .In this case the input data has indicated that the rest of the owchart continues over the leftmost boundary.However,once the flowchart s gone beyond this limit the program will stop.

REFERENCES

R W Witty, 'The design and construction of hierarchically structured software', 'Pragmatic Programming and Sensible Software', ONLINE Conference 1978.

N Wirth, 'Program Development by Step-Wise Refinement', CACM, Vol 7 No 5, Sept/Oct (1977).

Figure 9

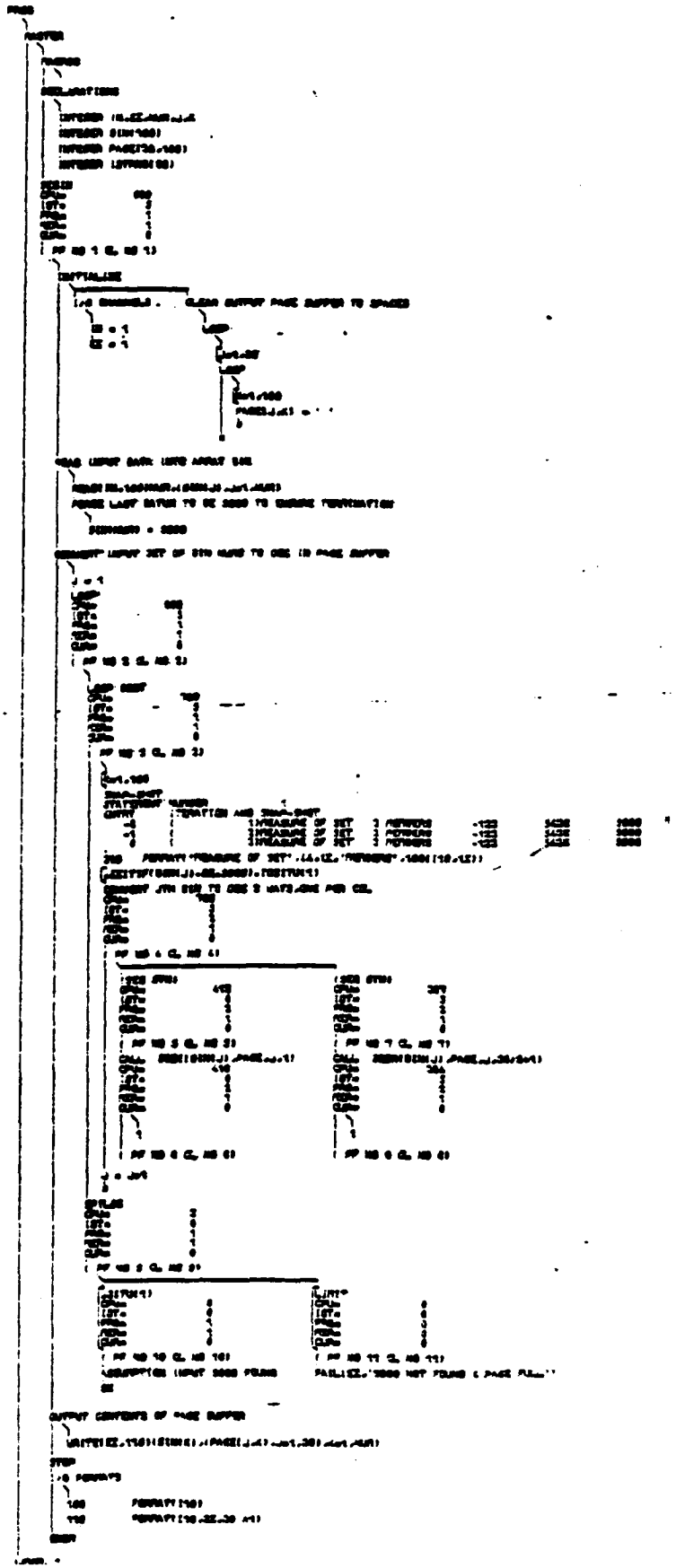
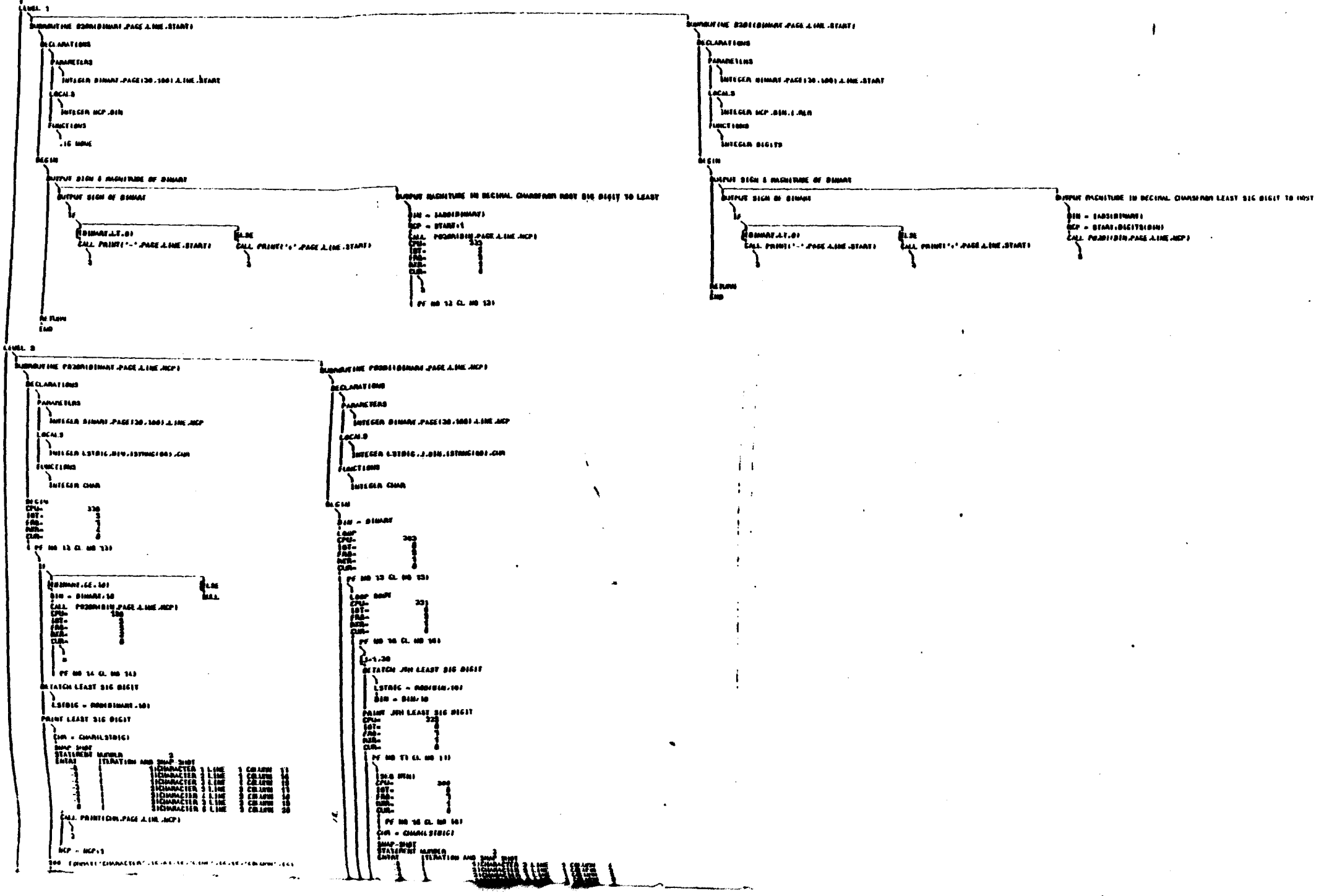


Figure 9 - continued



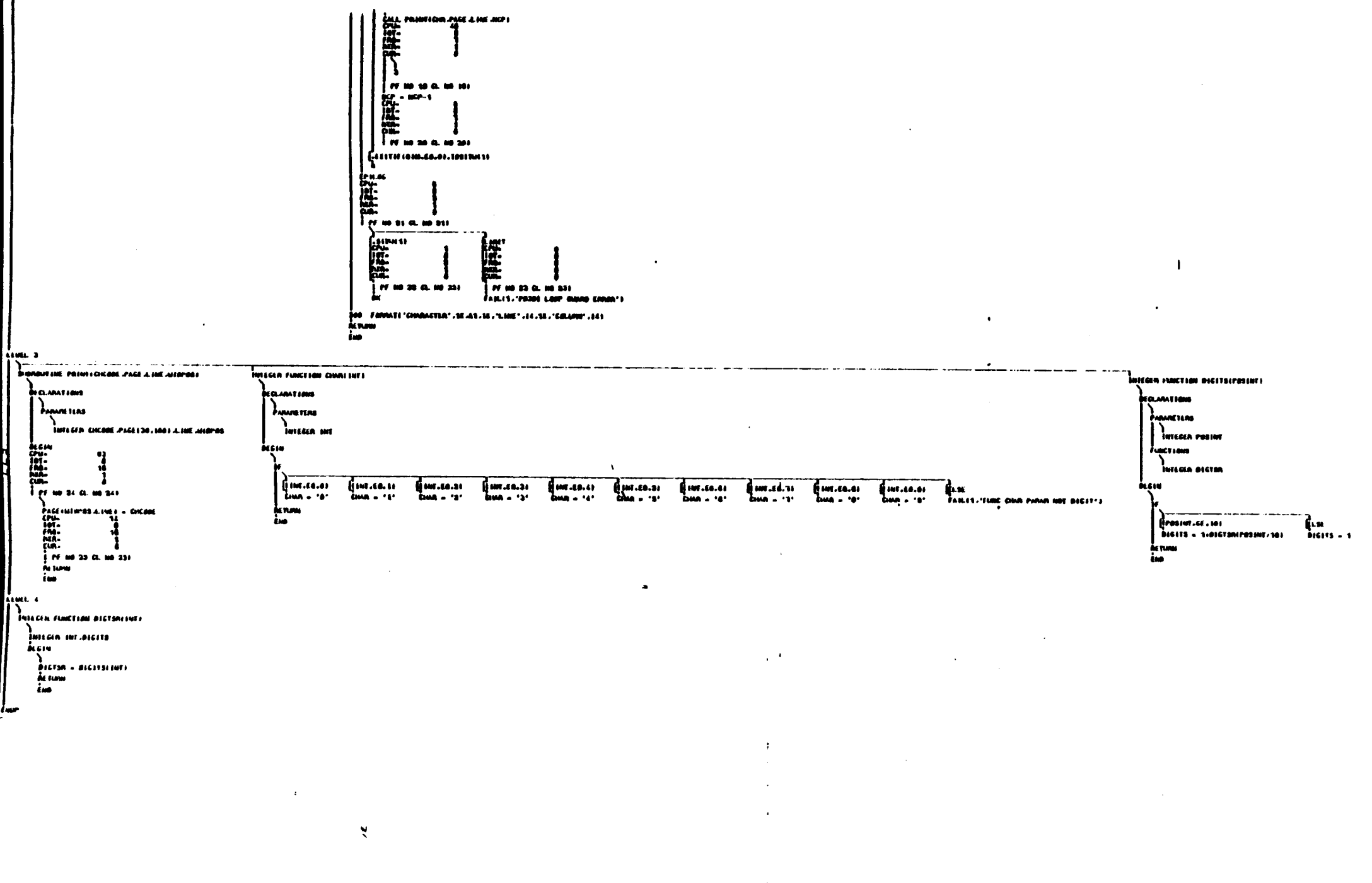
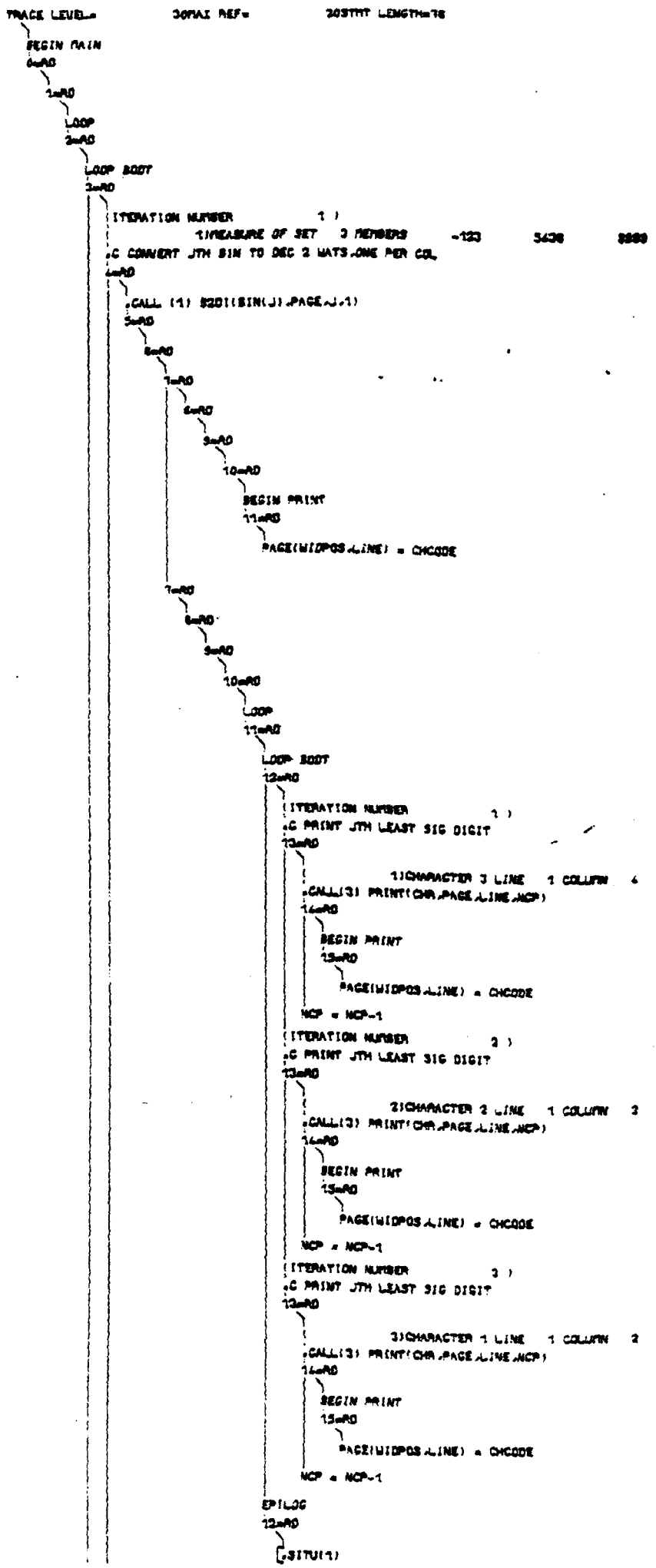


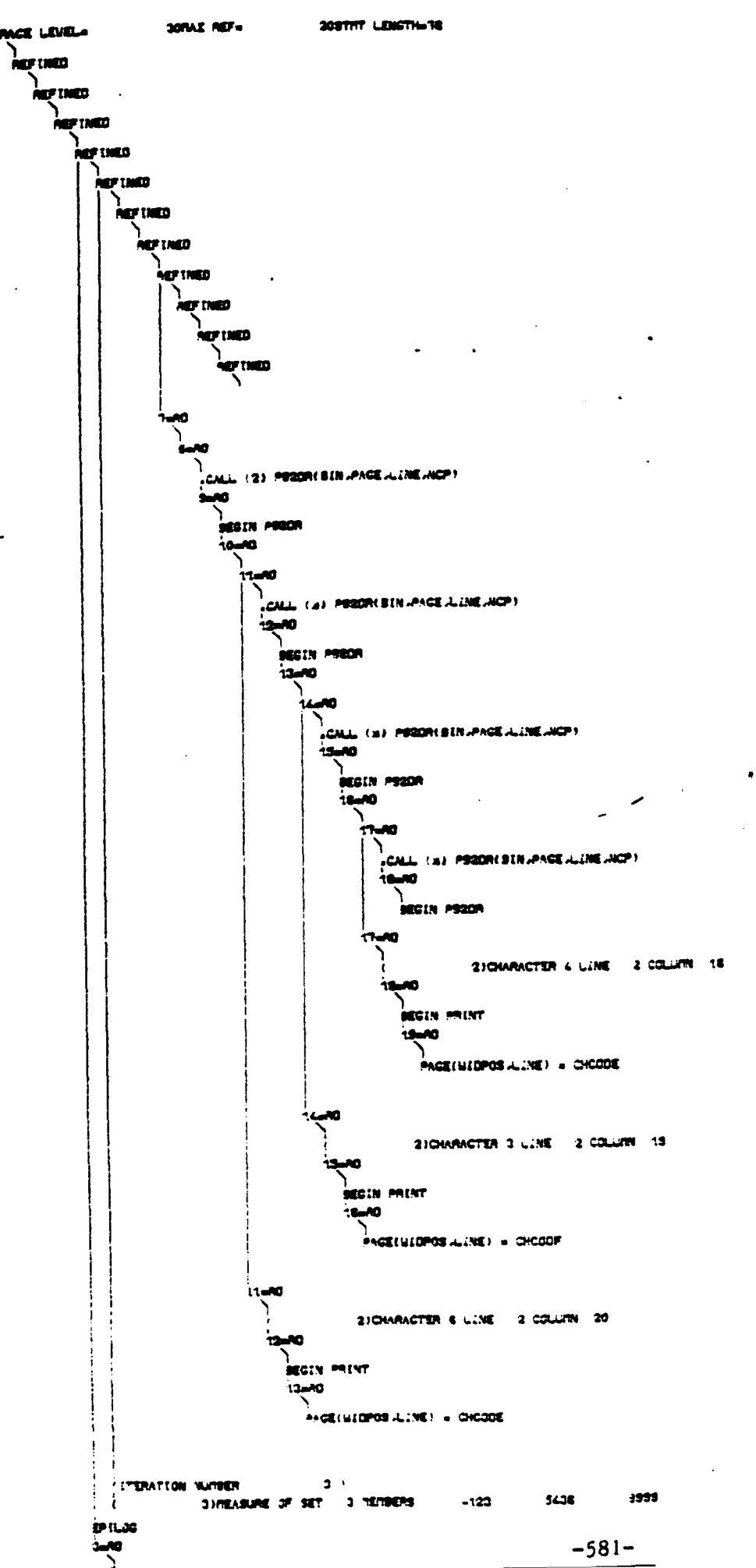
Figure 10


TRACE LEVEL=
MAX REF=
LENGTH=76
BEGIN MAIN
LOOP
LOOP BODY
ITERATION NUMBER 1
MEASURE OF SET 3 MEMBERS -123 5436 9999
CONVERT JTH BIN TO DEC 2 WAYS,ONE PER COL
CALL (1) BODY(BIN(J),PAGE,J,1)
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE
LOOP
LOOP BODY
ITERATION NUMBER 1
PRINT JTH LEAST SIG DIGIT
CHARACTER 3 LINE 1 COLUMN 4
CALL(3) PRINT(CHR,PAGE,LINE,NCP)
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE
NCP = NCP-1
ITERATION NUMBER 2
PRINT JTH LEAST SIG DIGIT
CHARACTER 2 LINE 1 COLUMN 3
CALL(3) PRINT(CHR,PAGE,LINE,NCP)
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE
NCP = NCP-1
ITERATION NUMBER 3
PRINT JTH LEAST SIG DIGIT
CHARACTER 1 LINE 1 COLUMN 2
CALL(3) PRINT(CHR,PAGE,LINE,NCP)
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE
NCP = NCP-1
EPILOG
SITU(1)

```
(ITERATION NUMBER          2 )
          2)MEASURE OF SET   3 MEMBERS      -123       9438       9999
.C CONVERT JTH SIN TO DEC 2 DATS.ONE PER COL
1=RO
  \
   .CALL (1) BCDI('SIN(J)',PAGE,J,1)
   2=RO
     \
      6=RO
        \
         7=RO
           |\
           | 8=RO
           |   \
           |    9=RO
           |      \
           |       10=RO
           |         \
           |          BEGIN PRINT
           |          11=RO
           |            \
           |             PAGE(WIDPOS,LINE) = CHCODE
           |
         7=RO
           |\
           | 8=RO
           |   \
           |    9=RO
           |      \
           |       10=RO
           |         \
           |          LOOP
           |          11=RO
           |            \
           |             LOOP BODY
           |             12=RO
           |               |\
           |               | (ITERATION NUMBER          1 )
           |               | .C PRINT JTH LEAST SIG DIGIT
           |               | 13=RO
           |               |   \
           |               |            1)CHARACTER 6 LINE   2 COLUMN   5
           |               |   .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
           |               |   14=RO
           |               |     |\
           |               |     | BEGIN PRINT
           |               |     | 15=RO
           |               |     |   \
           |               |     |    PAGE(WIDPOS,LINE) = CHCODE
           |               |   NCP = NCP-1
           |               | (ITERATION NUMBER          2 )
           |               | .C PRINT JTH LEAST SIG DIGIT
           |               | 13=RO
           |               |   \
           |               |            2)CHARACTER 3 LINE   2 COLUMN   4
           |               |   .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
           |               |   14=RO
           |               |     |\
           |               |     | BEGIN PRINT
           |               |     | 15=RO
           |               |     |   \
           |               |     |    PAGE(WIDPOS,LINE) = CHCODE
           |               |   NCP = NCP-1
           |               | (ITERATION NUMBER          3 )
           |               | .C PRINT JTH LEAST SIG DIGIT
           |               | 13=RO
           |               |   \
           |               |    14=RO
           |               |     |\
           |               |     | 15=RO
           |               |     |   \
           |               |            3)CHARACTER 4 LINE   2 COLUMN   3
           |               |   .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
           |               |   14=RO
           |               |     |\
           |               |     | BEGIN PRINT
           |               |     | 15=RO
           |               |     |   \
           |               |     |    PAGE(WIDPOS,LINE) = CHCODE
           |               |   NCP = NCP-1
           |               | (ITERATION NUMBER          4 )
           |               | .C PRINT JTH LEAST SIG DIGIT
           |               | 13=RO
           |               |   \
           |               |            4)CHARACTER 5 LINE   2 COLUMN   2
           |               |   .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
           |               |   14=RO
           |               |     |\
           |               |     | BEGIN PRINT
           |               |     | 15=RO
           |               |     |   \
           |               |     |    PAGE(WIDPOS,LINE) = CHCODE
           |               |   NCP = NCP-1
           |               IF(LOG
           |               12=RO
           |                 \
           |                  .3(TU(1)
```

```
4=R0
  .CALL (1) S2DR(SIN(J),PAGE,J,30,2+1)
  5=R0
    6=R0
      7=R0
        8=R0
          9=P0
            10=R0
              BEGIN PRINT
              11=R0
                PAGE(MIDPOS,LINE) = CHCODE

      7=R0
        8=R0
          .CALL (2) P92DR(SIN,PAGE,LINE,NCP)
          9=R0
            BEGIN P92DR
            10=R0
              11=R0
                .CALL (3) P92DR(SIN,PAGE,LINE,NCP)
                12=R0
                  BEGIN P92DR
                  13=R0
                    14=R0
                      .CALL (4) P92DR(SIN,PAGE,LINE,NCP)
                      15=R0
                        BEGIN P92DR
                        16=R0
                          17=R0
                            .CALL (4) P92DR(SIN,PAGE,LINE,NCP)
                            18=R0
                              BEGIN P92DR
                          17=R0
                                    2)CHARACTER 4 LINE   2 COLUMN  18
                            18=R0
                              BEGIN PRINT
                              19=R0
                                PAGE(MIDPOS,LINE) = CHCODE

                    14=R0
                                2)CHARACTER 3 LINE   2 COLUMN  19
                      15=R0
                        BEGIN PRINT
                        16=R0
                          PAGE(MIDPOS,LINE) = CHCODE

                11=R0
                            2)CHARACTER 6 LINE   2 COLUMN  20
                  12=R0
                    BEGIN PRINT
                    13=R0
                      PAGE(MIDPOS,LINE) = CHCODE


 (ITERATION NUMBER          3 )
          3)MEASURE OF SET   3 MEMBERS      -123       5436       9999
2=!LOG
3=R0
  .3(TU(1)
```

Figure 11 - Example of Circular Trace



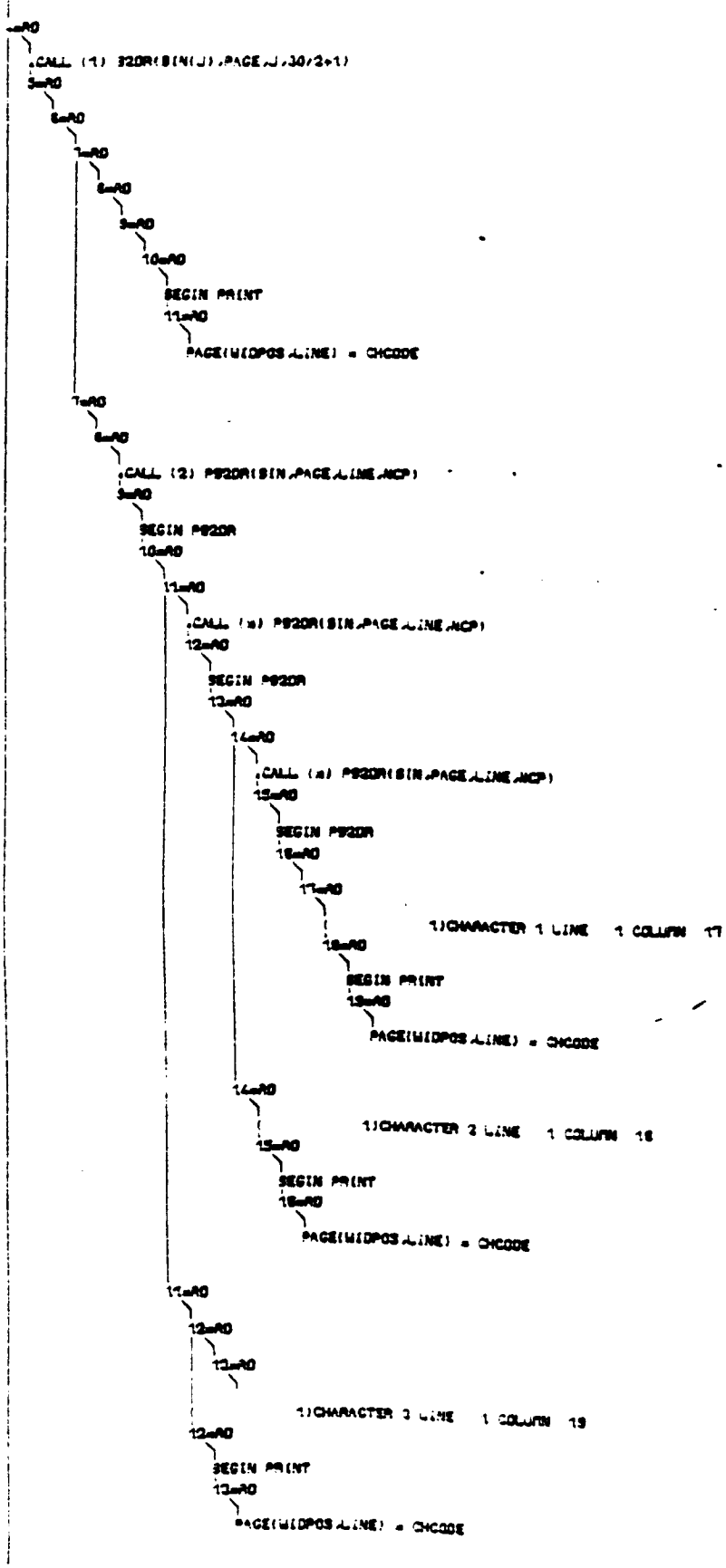
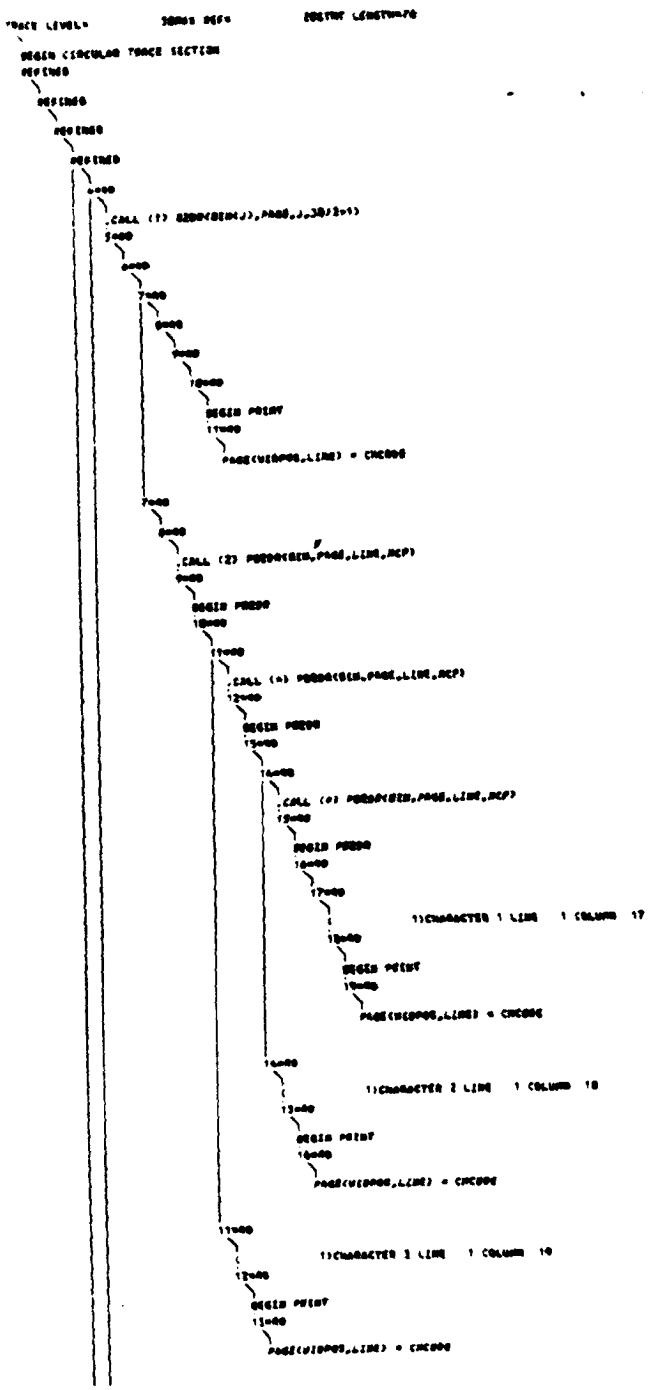

CALL (1) P20R(SIN(J),PAGE,J,30/2+1)
BEGIN PRINT
PAGE(MIDPOS,LINE) = CHCODE
CALL (2) P920R(SIN,PAGE,LINE,NCP)
BEGIN P920R
CALL (3) P920R(SIN,PAGE,LINE,NCP)
BEGIN P920R
CALL (4) P920R(SIN,PAGE,LINE,NCP)
BEGIN P920R
1)CHARACTER 1 LINE 1 COLUMN 17
BEGIN PRINT
PAGE(MIDPOS,LINE) = CHCODE
1)CHARACTER 2 LINE 1 COLUMN 18
BEGIN PRINT
PAGE(MIDPOS,LINE) = CHCODE
1)CHARACTER 3 LINE 1 COLUMN 19
BEGIN PRINT
PAGE(MIDPOS,LINE) = CHCODE

FIGURE 12 - EXAMPLE OF CIRCULAR TRACE CHANGED TO HISTORICAL TRACE



```
DECLAR TRACE SECTION
VIEW NUMBER          2 )
      2)MEASURE OF SET   3 MEMBERS      -123       5436       9999
HEAT JTH BIN TO DEC 2 WAYS,ONE PER COL

A (1) 6291(BIN(4),PAGE,,,,)


 +00
  \
  7+00
   \
    8+00
     \
      9+00
       \
       10+00
        \
         BEGIN POINT
         11+00
          PAGE(VIEWPOS,LINE) = CHCODE

 7+00
  \
   8+00
    \
     9+00
      \
      10+00
       \
        1000
        11+00
         \
         LOOP BODY
         12+00
          \
          (ITERATION NUMBER          1 )
          .C POINT JTH LEAST SIG DIGIT
          13+00
           \
           (          1)CHARACTER 6 LINE   2 COLUMN   5
           .CALL(3) POINT(CHR,PAGE,LINE,HCP)
           14+00
            \
            BEGIN POINT
            15+00
             \
              PAGE(VIEWPOS,LINE) = CHCODE
           HCP = HCP+1
          (ITERATION NUMBER          2 )
          .C POINT JTH LEAST SIG DIGIT
          13+00
           \
           (          2)CHARACTER 5 LINE   2 COLUMN   4
           .CALL(3) POINT(CHR,PAGE,LINE,HCP)
           14+00
            \
            BEGIN POINT
            15+00
             \
              PAGE(VIEWPOS,LINE) = CHCODE
           HCP = HCP+1
          (ITERATION NUMBER          3 )
          .C POINT JTH LEAST SIG DIGIT
          13+00
           \
           (          3)CHARACTER 4 LINE   2 COLUMN   3
           .CALL(3) POINT(CHR,PAGE,LINE,HCP)
           14+00
            \
            BEGIN POINT
            15+00
             \
              PAGE(VIEWPOS,LINE) = CHCODE
           HCP = HCP+1
          (ITERATION NUMBER          4 )
          .C POINT JTH LEAST SIG DIGIT
          13+00
           \
           (          4)CHARACTER 3 LINE   2 COLUMN   2
           .CALL(3) POINT(CHR,PAGE,LINE,HCP)
           14+00
            \
            BEGIN POINT
            15+00
             \
              PAGE(VIEWPOS,LINE) = CHCODE
           HCP = HCP+1
         EPILOG
         12+00
          \
          SITU(1)
```

TEXT BOUND INTO

THE SPINE

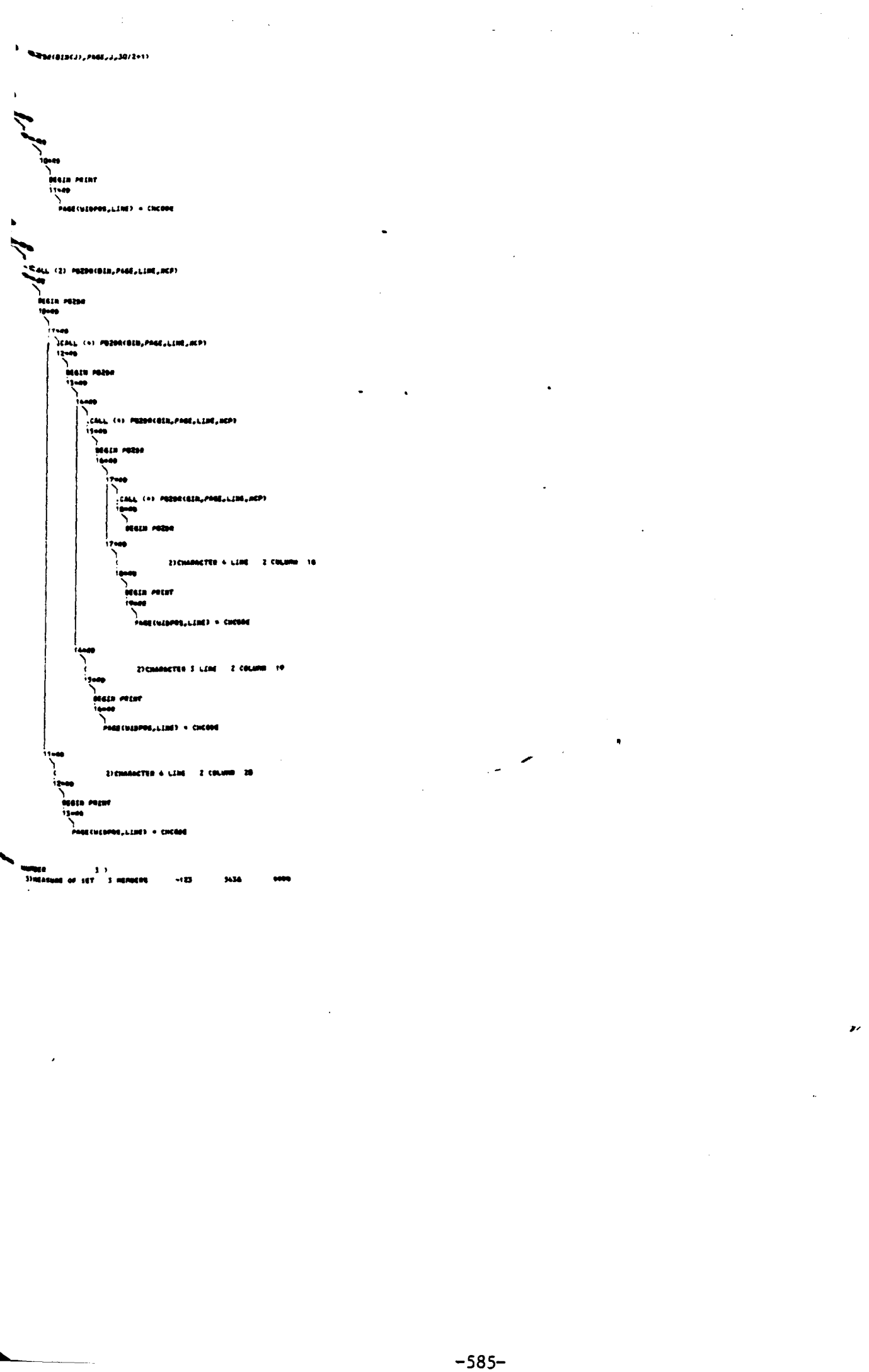
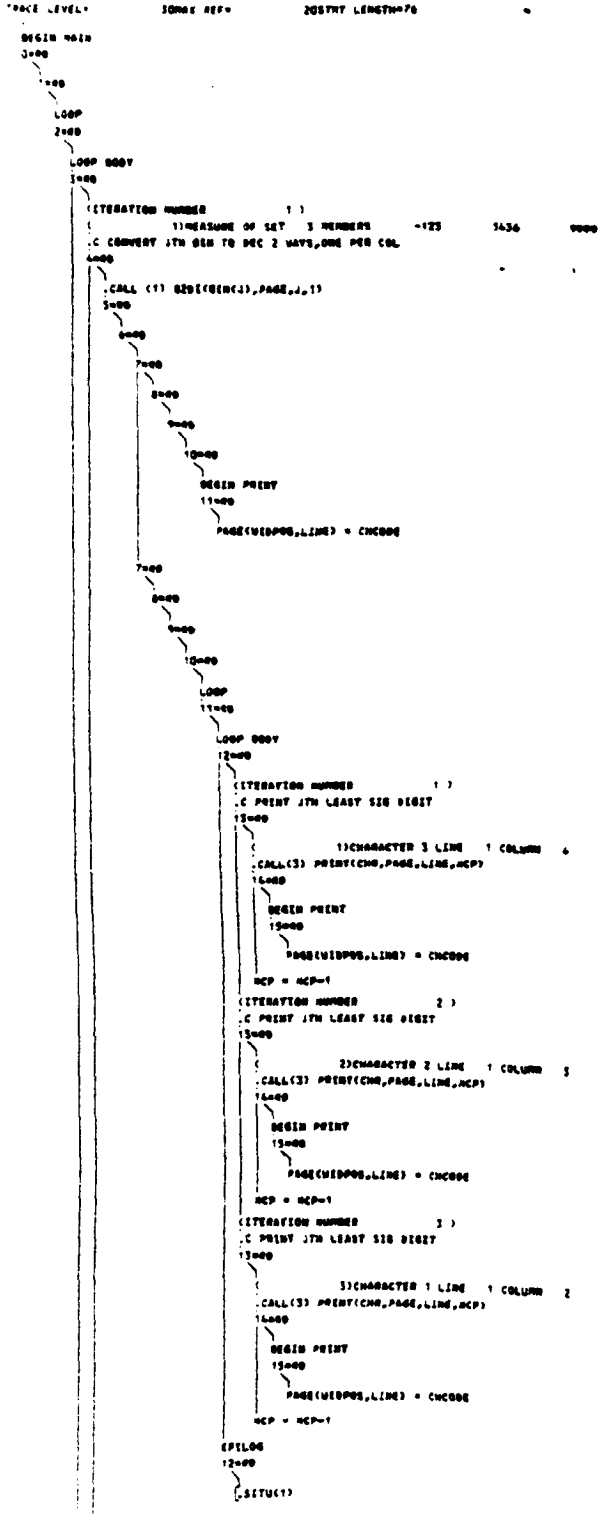
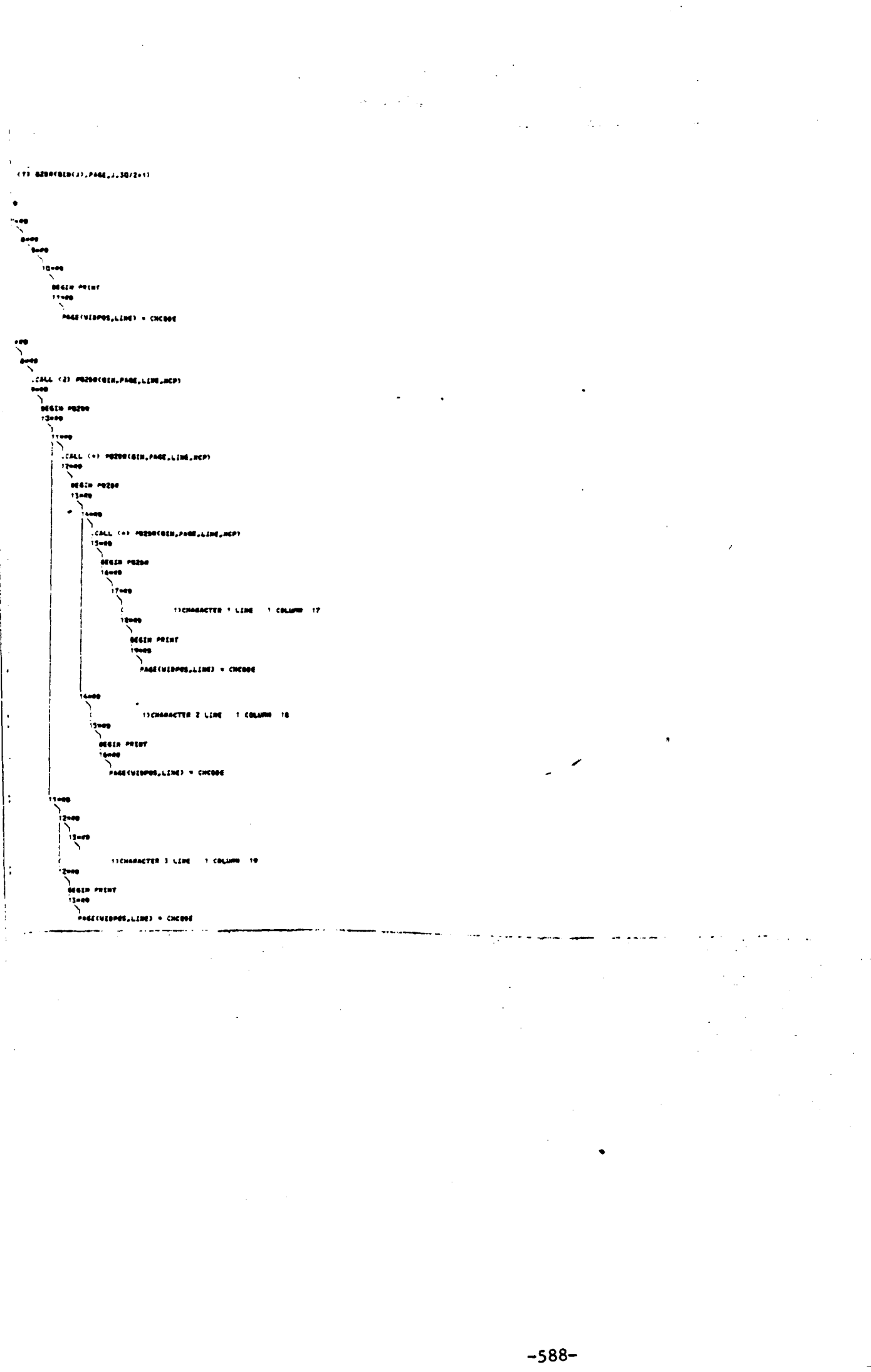

BEGIN POINT
PAGE(WINPOS,LINE) = CHCODE
CALL (2) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
CALL (*) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
CALL (*) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
CALL (*) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
2)CHARACTER 4 LINE 2 COLUMN 16
BEGIN POINT
PAGE(WINPOS,LINE) = CHCODE
2)CHARACTER 3 LINE 2 COLUMN 19
BEGIN POINT
PAGE(WINPOS,LINE) = CHCODE
2)CHARACTER 4 LINE 2 COLUMN 20
BEGIN POINT
PAGE(WINPOS,LINE) = CHCODE
3)READING OF SET 1 MEMBERS

FIGURE 13 - EXAMPLE OF HISTORICAL TRACE CHANGED TO CIRCULAR TRACE



```
TRACE LEVEL=         30MAX REF=        20STMT LENGTH=70
 BEGIN MAIN
 0=00
   1=00
     LOOP
     2=00
       LOOP BODY
       3=00
         (ITERATION NUMBER            1 )
         (            1)MEASURE OF SET   3 MEMBERS      -123       5436       9999
         .C CONVERT JTH BIN TO DEC 2 WAYS,ONE PER COL
         4=00
           .CALL (1) B2D(BIN(J),PAGE,J,1)
           5=00
             6=00
               7=00
                 8=00
                   9=00
                     10=00
                       BEGIN PRINT
                       11=00
                         PAGE(WIDPOS,LINE) = CHCODE

               7=00
                 8=00
                   9=00
                     10=00
                       LOOP
                       11=00
                         LOOP BODY
                         12=00
                           (ITERATION NUMBER            1 )
                           .C PRINT JTH LEAST SIG DIGIT
                           13=00
                             (            1)CHARACTER 3 LINE   1 COLUMN   6
                             .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
                             14=00
                               BEGIN PRINT
                               15=00
                                 PAGE(WIDPOS,LINE) = CHCODE
                             NCP = NCP-1
                           (ITERATION NUMBER            2 )
                           .C PRINT JTH LEAST SIG DIGIT
                           13=00
                             (            2)CHARACTER 2 LINE   1 COLUMN   5
                             .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
                             14=00
                               BEGIN PRINT
                               15=00
                                 PAGE(WIDPOS,LINE) = CHCODE
                             NCP = NCP-1
                           (ITERATION NUMBER            3 )
                           .C PRINT JTH LEAST SIG DIGIT
                           13=00
                             (            3)CHARACTER 1 LINE   1 COLUMN   2
                             .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
                             14=00
                               BEGIN PRINT
                               15=00
                                 PAGE(WIDPOS,LINE) = CHCODE
                             NCP = NCP-1
                         EPILOG
                         12=00
                           .SITU(1)
```

```
ACE SECTION

BEGIN PRINT
 9+00
  PAGE(VIDPOS,LINE) = CHCODE


(2) PD2D0(BIN,PAGE,LINE,NCP)

IN PD2D0

 7+00
  .CALL (+) PD2D0(BIN,PAGE,LINE,NCP)
 12+00
   BEGIN PD2D0
   13+00
    14+00
     15+00 (+) PD2D0(BIN,PAGE,LINE,NCP)
      BEGIN PD2D0
      16+00
       17+00
        .CALL (+) PD2D0(BIN,PAGE,LINE,NCP)
        18+00
         BEGIN PD2D0
       17+00
        (          2)CHARACTER 4 LINE   2 COLUMN  18
       18+00
         BEGIN PRINT
         19+00
          PAGE(VIDPOS,LINE) = CHCODE

    14+00
     (          2)CHARACTER 3 LINE   2 COLUMN  19
    15+00
      BEGIN PRINT
      16+00
       PAGE(VIDPOS,LINE) = CHCODE

 7+00
  (          2)CHARACTER 6 LINE   2 COLUMN  28
 12+00
   BEGIN PRINT
   13+00
    PAGE(VIDPOS,LINE) = CHCODE


E SECTION
          3 )
SUME OF SET   3 MEMBERS      =123       5416       9000
```


(1) PB200(BIN(1),PAGE,1.30/2+1)
.CALL (2) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
.CALL (*) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
.CALL (*) PB200(BIN,PAGE,LINE,NCP)
BEGIN PB200
(1)CHARACTER 1 LINE 1 COLUMN 17
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE
(1)CHARACTER 2 LINE 1 COLUMN 18
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE
(1)CHARACTER 3 LINE 1 COLUMN 19
BEGIN PRINT
PAGE(WIDPOS,LINE) = CHCODE

10.1.3. "Safe Programming".[3]

BIT 18 (1978), 1–8

SAFE PROGRAMMING

T. ANDERSON and R. W. WITTY

Abstract.

Safe specifications and programs are advocated as a simple way of enhancing the reliability of software. The behaviour of a safe program can be more easily certified as being correct with respect to its safe specification, which implies guaranteed termination. This paper describes the theory of safe programming, demonstrates the building of a safe program and summarises the experience gained from practical applications of safe programming.

Key Words: Bounded repetition, Correctness, Reliability, Termination.

Introduction.

The provision of a certification of the correctness of a program is intended to increase confidence that the behaviour of the executed program will conform to what is required of the program. Such a certification should consist of:

(a) a specification of the intended behaviour of the program, and
(b) an argument to show that, when executed, the program will always meet this specification.

The argument often breaks down into two parts; one part to show that all executions of the program terminate and the other to show that on termination the required results have been obtained. To be convinced that the program is indeed correct it is necessary to be satisfied that the specification is appropriate and that the argument (usually called a proof) is valid. Unfortunately, current experience indicates that correctness proofs constructed for even quite short programs can be lengthy, complex and (as demonstrated by Gerhard and Yelowitz [5] and Anderson [1]) invalid. For a proof to be of real value it should be clearer and simpler than the associated program.

To illustrate the above remarks a conventional proof of correctness will be presented for a small program. Then, in contrast, the specification will be altered to what is termed a safe specification, a safe program constructed, and a short, clear proof of safeness given. The practical application of safe programming is then discussed.

Received July 29, 1977. Revised October 31, 1977.

A Correct Program.

Consider the following specification:

Using only integer arithmetic, find the largest integer i less than or equal to the square root of a given non-negative integer constant n (find $i = \lfloor \sqrt{n} \rfloor$). That is, given $n \geqq 0$ find i such that

$$(i^2 \leqq n) \wedge ((i+1)^2 > n) .$$

The following solution is based on the Newton-Raphson root finding method.

```
SOLUTION1:
  begin
  i := ⌊(n+1)/2⌋ {initial value};
  while i² > n do
    begin
    i := ⌊(n+i²)/(2i)⌋ {next estimate};
    end;
  end SOLUTION1;
```

A proof of correctness can be given as:

PROOF OF TERMINATION. On entry to the loop body $i^2 > n$ which together with $i > 0$ implies that $i > \lfloor (n+i^2)/2i \rfloor$. The assignment $i := \lceil (n+i^2)/2i \rceil$ at each iteration ensures that the value of i must strictly decrease, and as it is always non-negative, only a bounded number of iterations can therefore occur.

PROOF OF CORRECT BEHAVIOUR.

1. After initialisation, $(i+1)^2 > n$ since $(\lfloor (n+1)/2 \rfloor + 1)^2 > n$.
2. For any positive i, $(\lfloor (n+i^2)/2i \rfloor + 1)^2 > n$ and so $(i+1)^2 > n$ after each assignment in the body of the loop.
3. Thus $(i+1)^2 > n$ always, and after termination $i^2 \leqq n$ also (from the **while** test).

The above proof is very informal, and some simple lemmas on integers have been omitted. However, it fails to inspire a great deal of confidence. If a proof is to be of real value it should be clearer and easier to understand than the associated program. For just as a simple program is more likely to be correct than a complex program, so a simple proof is more likely to be valid than a complex proof.

Adequate Programs.

Proof guided program design methodologies, as advocated by Dijkstra [3, 4], help to create simpler proofs. A variation of these techniques is possible; instead of attempting to prove the correctness of a program with respect to its original specification, some weaker criterion of acceptable behaviour is selected. That is, if the original specification is denoted by P then a specification Q is chosen such that:

(a) any program which conforms to P will also conform to Q, and
(b) Q prescribes an acceptable behaviour of the program.

The program is then designed and constructed in an attempt to conform to P, but so as to facilitate the provision of a much simpler proof of correctness with respect to Q than would be possible using P. Such a proof will be termed a proof that the program is *adequate*.

Safe Programs.

In the context of software reliability a special case of adequacy, termed safeness, is relevant. As a weaker specification for a program intended to satisfy P, take Q to be $P \vee$ "error", meaning that the program should either behave as was originally intended or should terminate with an explicit indication of the reason for failure. A proof of adequacy for this particular form of Q will be termed a proof that the program is *safe*.

Ideally, a program should be designed so that its proof of safeness can be substantially simpler than a corresponding correctness proof. One way of achieving this objective is shown in the following solution to the largest square root problem introduced above.

Safe specification:

Given $n \geq 0$ find i such that

$((i^2 \leq n) \wedge ((i+1)^2 > n)) \vee$ "error"

Program:

```
SOLUTION2:
  begin
  i := ⌊(n+1)/2⌋ {initial value};
  iteration_counter := 0;
  while (i² > n) and iteration_counter < iteration_limit do
    begin
      i := ⌊(n+i²)/(2i)⌋ {next estimate};
      iteration_counter := iteration_counter + 1;
    end;
  {safety check 1}
  if iteration_counter > iteration_limit then error("loop limit");
  {safety check 2}
  if not ((i² ≤ n) and ((i+1)² > n)) then error("wrong. answer");
  end SOLUTION2;
```

PROOF OF SAFENESS. *Termination*: Guaranteed by testing of iteration_counter.
Adequacy: After termination either an error will have been detected or a correct answer will have been calculated since an explicit test of correctness is included.

The simple nature of this proof leaves little opportunity for error which justifies a high level of confidence in the safeness of the program.

Note that the proof does not depend on the particular expressions used as "initial value" and "next estimate", or on the value of the iteration_limit (as yet unspecified). However, the requirement that the program be designed to also conform to the original specification demands that appropriate expressions are chosen (the proof of termination for *SOLUTION*1 suggests $\lfloor (n+1)/2 \rfloor$ as the iteration_limit). Except for this requirement, the program below would be an acceptable solution since it is safe for all specifications *P*.

begin error("wrong answer"); **end**;

An argument for safeness can be made independent of any assumptions about the input to a program, since any necessary assumptions can be checked at run time. Hence, safe behaviour can be guaranteed even with invalid input data, whereas correctness proofs conventionally assume valid input.

More generally, the adoption of a safe programming specification enables a programmer to introduce redundancy into a program specifically as a means of simplifying the proof of the program with respect to that specification. Redundancy included in a program for this purpose will often be in the form of **assert** statements (eg Algol-W[9]); a proof of safeness can rely on all such assertions holding when the program is executed since otherwise a failure indication would be generated.

It should be noted that an over stringent or otherwise ill-chosen assertion may generate a failure indication when the same program without the assertion would have executed correctly. Such an occurrence is indicative of a lack of understanding on the part of the programmer and, in practice, rectification of such an occurrence always leads to a deeper understanding of the program's specification and a more reliable program. The consequences of an over stringent assertion may be contrasted favourably with those resulting from the failure of a weak (or non-existent) assertion to detect an erroneous execution of the program.

By augmenting a safe program with routines which take corrective action in the event of an erroneous situation being detected, a significant enhancement of the reliability of the program can be obtained. The recovery block notation described by Randell [8] can be used to achieve this augmentation without increasing the complexity of the program. Anderson [1] has elaborated on these points.

Bounded Repetition.

The above program is atypical in that the explicit testing of the results of a program is rarely feasible in practice. However, it seems perfectly feasible to

eliminate the need for a proof of termination simply by programming in languages which ensure that all programs must halt, thereby greatly simplifying the overall proof.

Such languages do not provide explicit control transfer and impose constraints on all iterative and recursive facilities. Consequently they cannot be used to program all of the recursive (computable) functions, and are known as sub-recursive languages. The work of Constable and Borodin [2] indicates that such languages can provide all of the functions actually used in computing and this seems to be borne out in practice (see below). Indeed, these restrictions are an advantage of the subrecursive languages.

An iterative facility provided by many languages can be denoted by:

repeat *S* **possibly forever**

where *S* denotes a statement list which may or may not include conditional exits. *S* is repeatedly executed until an exit is taken whereupon the construct is terminated. The **while** loop is a typical example of this type of iteration.

Consider two special cases of the construct.

repeat *S* **forever**

S contains no exits and is repeated infinitely. This special case is rarely needed, and would deserve careful consideration if it were.

repeat *S* **exactly** *n* **times**

Here *n* denotes a non-negative integer value; *S* contains no exits and is executed precisely *n* times. This special case is frequently needed. Its termination is guaranteed.

The main criticism of the more powerful **possibly forever** construct is that it permits infinite repetition when in all probability the programmer did not intend this to occur. By analogy with the two special cases above an alternative version is suggested which prevents infinite repetition.

repeat *S* **upto** *n* **times**

S contains one or more conditional exits and is executed at most *n* times, the construct being terminated earlier if an exit is taken.

A sub-recursive language only provides bounded iteration constructs.

repeat *S* **upto** *n* **times** (*S* contains one or more exits)
repeat *S* **exactly** *n* **times** (*S* contains no exits)

If potentially infinite iteration is to be included in a programming language then a separate construct should be specially provided.

Recursive constructs may be constrained in a similar manner to the iterative constructs discussed above.

Luckham and Suzuki have reported [6] on work related to the proposals of this section. They advocate the use of repetition counters as a means of formally establishing termination within a weak logic of programs.

Experience with Adequate Programs.

An attempt has been made to demonstrate the possibility of writing a practical piece of software so as to obtain a simple proof of adequacy, by a postgraduate student at Newcastle University (M. S. Reynolds). A file system was implemented with specification P: "All user commands to the file system are correctly processed". A proof of adequacy was provided for the specification Q: "All user commands to the file system are either correctly processed, or if not, the user is sent a warning message and the integrity of all previously filed data is maintained". By means of isolating those routines which actually modified the file structures, and incorporating run time checks to verify their actions, a reasonably simple proof of Q was obtained. A large portion of the software could be ignored completely when establishing adequacy, a considerable benefit. Another encouraging feature of this experiment was that throughout the debugging phase, when the program was patently not correct, its behaviour was, however, always adequate.

Experience with Safe Programs.

The Rutherford Laboratory has a small, primitive mini-computer (used to control a graphics system) whose only software tools are an assembler, a loader and a debugging tool which allows the examination and alteration of the contents of absolute memory addresses. The machine has no supervisor program, no memory partitioning or protection hardware and no printer. A major difficulty in programming this machine is that erroneous programs generally overwrite themselves, thereby making debugging extremely difficult. It was therefore decided to construct all new software according to the principles of safe programming.

Several programs have been constructed including a multi-tasking system. The first program was built from 8000 lines of hand coded assembler statements and has never failed. It has been in constant use since September 1975 logging details about the resources consumed by the users of the graphics system. The multi-tasking program to actually control the graphics system was built by cross-compiling over 20,000 lines of a simple systems implementation language which included multiple exit loops based on Zahn's construct [10]. These proved a success as they eliminated the need to follow each loop by additional, redundant tests to determine which of the possible exit conditions actually terminated the loop (see below).

None of the safe programs written so far has overwritten itself or failed to terminate, even during development when bugs were obviously present. The need to place a bound on each loop proved beneficial rather than restrictive. The very act of determining the loop limit caught errors at the design stage. It was surprising how small most of the loop bounds were in practice, and how most loops had natural bounds anyway. For example, in the multi-tasking program

there is a routine which reads characters from a terminal until a carriage return character is encountered. This could have been coded as:

```
I := 0;
loop
  I := I+1;
  BUFFER(I) := readnextchar(teletype);
repeat unless
  BUFFER(I) eq CarriageReturn;
endloop;
```

Simple, elegant, efficient and lethal because the inputting of too many characters before the carriage return could have caused overwriting of the memory area after the end of the buffer. The natural loop limit here was the number of characters making up a line on the terminal, 72 on an ordinary teletype, which led to an easy proof of safeness. This gave rise to the version:

```
I := 0;
loop
  I := I+1;
  BUFFER(I) := readnextchar(teletype);
  terminate gotline if BUFFER(I) eq CarriageReturn;
  terminate snag if done 72 times;
repeat
  situation gotline causes OK;
  situation snag causes error("line too long");
endloop;
```

which utilises the form of Zahn loop contained in the systems implementation language mentioned above. The run time overheads associated with bounded repetition proved negligible as the "**if done** ⟨*limit*⟩ **times**" construct allowed very compact and efficient code to be generated. An extra add and test instruction per loop was a small price to pay for the increased reliability.

Meissner [7] has reported favourably on bounded loops and has suggested a template from which bounded loops may be constructed in FORTRAN.

Using Meissner's template, the above example would lead to the following FORTRAN solution:

```
   I=0
   DO 7 J=1,72
   I=I+1
   BUFFER(I) = READCH(TTY)
   IF (BUFFER(I).EQ.CRET) GOTO 8
 7 CONTINUE
   CALL ERROR("LINE TOO LONG")
 8 CONTINUE
```

The knowledge that a program will terminate safely whatever its input has greatly increased confidence in the programs; it has saved hours of debugging time and has increased enormously the programmers' peace of mind.

Conclusion.

Safeness directed program design and construction really works.

Acknowledgement.

The authors would like to acknowledge the support of the Science Research Council of Great Britain.

REFERENCES

1. T. Anderson, *Probably Safe Programs*, Tech. Rep. 70, Computing Laboratory, University of Newcastle upon Tyne (February 1975).
2. R. L. Constable and A. B. Borodin, *Subrecursive Programming Languages*, Part I: Efficiency and Program Structure, J. ACM 19 (July 1972), 526-568.
3. E. W. Dijkstra, *Concern for Correctness as a Guiding Principle for Program Composition*, Infotech State of the Art Report 1: The Fourth Generation (1971), 357-367.
4. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall (1976).
5. S. L. Gerhart and L. Yelowitz, *Observations of Fallibility in Applications of Modern Programming Methodologies*, IEEE Trans. on Software Engineering 2 (September 1976), 195-207.
6. D. C. Luckham and N. Suzuki, *Proof of Termination within a Weak Logic of Programs*, Acta Informatica 8 (1977), 21-36.
7. L. P. Meissner, *Bounded Loops*, FOR-WORD 3 (January 1977).
8. B. Randell, *System Structure for Software Fault Tolerance*, Current Trends in Programming Methodology 1, Prentice-Hall (1977), 195-219.
9. E. H. Satterthwaite, *Debugging Tools for High Level Languages*, Software – Practice & Experience 2 (July 1972), 197-217.
10. C. T. Zahn, *A Control Statement for Natural Top-Down Structured Programming*, Lecture Notes in Computer Science 19, Springer Verlag (1974), 170-180.

COMPUTING LABORATORY
UNIVERSITY OF NEWCASTLE UPON TYNE
ENGLAND

AND

ATLAS COMPUTING DIVISION
RUTHERFORD LABORATORY
CHILTON, DIDCOT
OXFORDSHIRE, ENGLAND

10.1.4. "Dimensional Flowcharting".[83]

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 7, 553–584 (1977)

Dimensional Flowcharting

ROBERT W. WITTY
Atlas Computing Division, Rutherford Laboratory, Chilton, Didcot, Oxon OX11 0QY, England

SUMMARY

By introducing the idea of axes Dimensional Flowcharting clarifies the representation of sequential and parallel operations. Step-Wise Refinement is added to give an improved method of representing and understanding the design of software. Many of the disadvantages of conventional flowcharting are removed. The dimensionality concept is carried through to the construction of working programs and a dimensional programming language is described.

KEY WORDS Structured programming Flowcharting Language design

BACKGROUND

Introduced here is a flowcharting technique which helps the software designer to use the Step-Wise Refinement (SWR) method[1] and which helps the programmer to produce well disciplined, sequential code. The technique forms the cornerstone of a method of constructing software[2] which has been successfully used to design and document a working system.[3] During the initial design of the system, conventional flowcharts were produced at successive levels of detail using the SWR method. This work showed that conventional flowcharts have some disadvantages.

1. They cannot be drawn 'naturally'; programming languages are often easier to use because they follow the normal lexical directions of English.
2. They do not conveniently represent modern high-level language control flow constructs.
3. They do not clearly represent parallelism because sequential actions can spread out in any direction.
4. They are difficult to convert into a machine-readable form.
5. They are difficult to format when drawn on a plotting device.
6. They fail to show how a final, detailed design has been achieved through Step-Wise Refinement.

To cope with these problems a variant of flowcharting, Dimensional Flowcharting, has been developed.

DIMENSIONAL FLOWCHARTING

Consider the CASE statement in Figure 1. This is a well disciplined construct having one entry point and one exit point. It shows how the conventional direction of sequential control is from the top to the bottom of a flowchart. It also reveals that the executable code

Received 24 April 1976

blocks E_1, E_2, E_3 exist as parallel alternatives, although only one is executed to the exclusion of the others. This is the selective CASE statement, the most common form. In a generalized CASE statement, whose semantics are that each and every true case is executed, the parallelism is more obvious.[4] From this example one can postulate that the 'dimensions' of the flowchart are Sequential Control Flow (SCF) and parallel execution (P) (Figure 2).

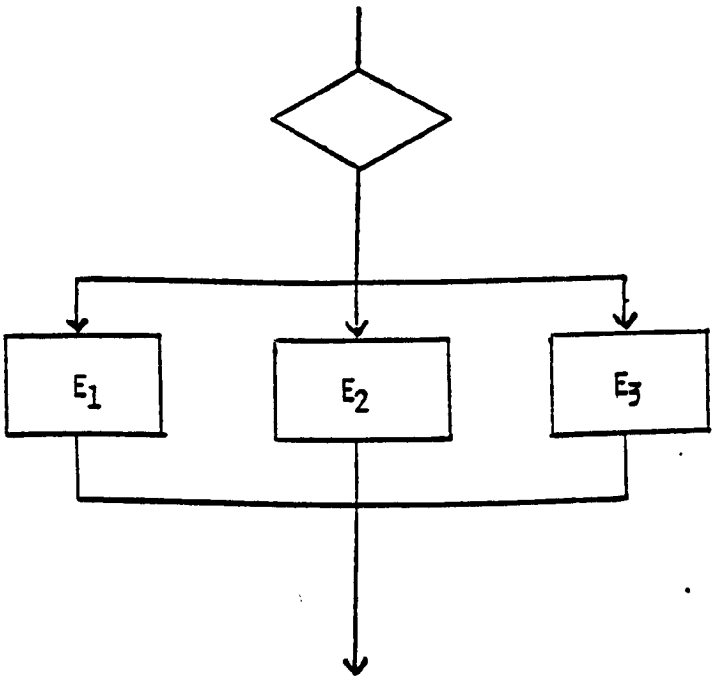


Figure 1. Conventional CASE statement representation

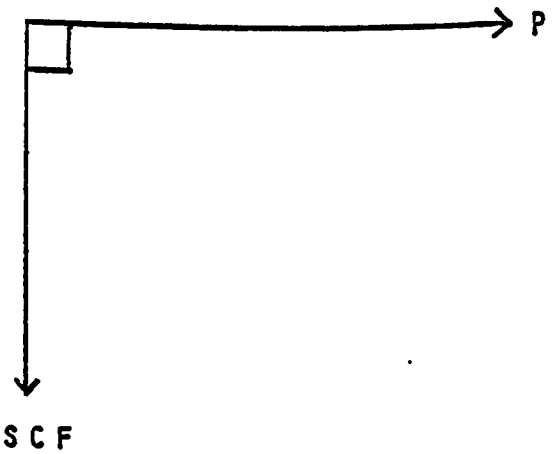


Figure 2. 2-D Axes—Sequential Control Flow and Parallelism

Sequential statements

To produce well disciplined, sequential programs the flow of control must *always* be top to bottom (SCF increasing); branch instructions are forbidden. Every program must be a sequence of statements having only one entry point and one exit point, such as IF-THEN-

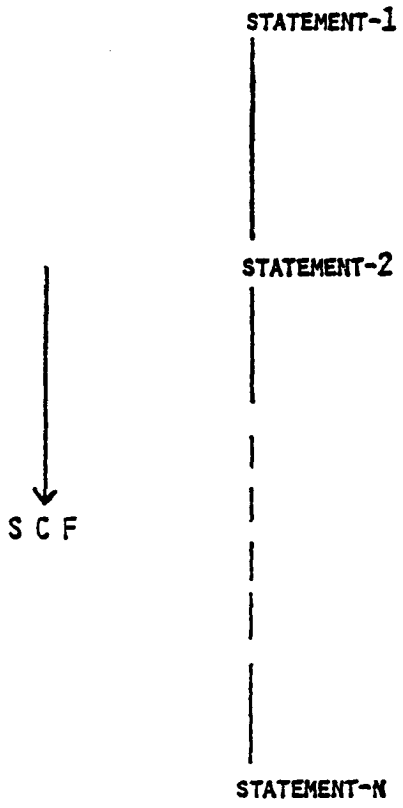


Figure 3. Sequential statements representation

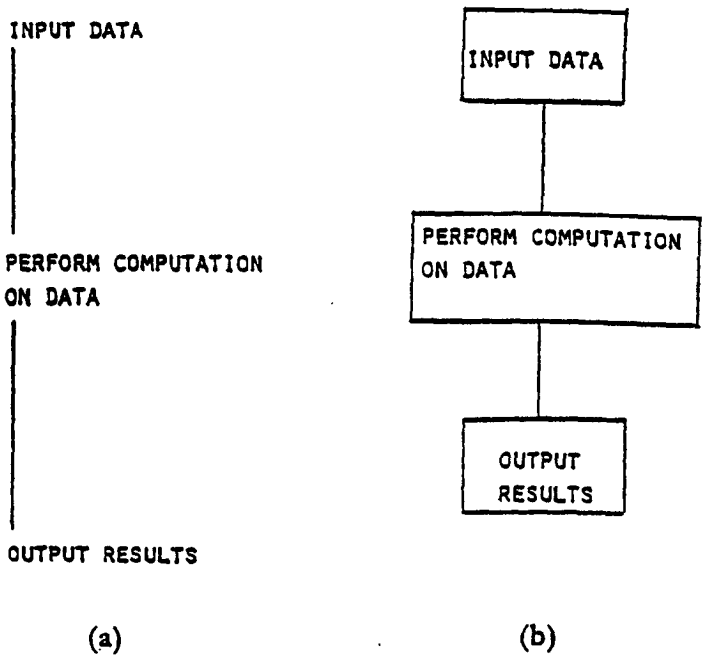


Figure 4. Contrasting representations of sequential statements, (a) dimensional, (b) conventional

ELSE, CASE and LOOP. Hence in Dimensional Flowcharting a sequence of statements is shown as in Figure 3. Any statement represents either program source text or an informal description of an action depending on the level of abstraction the of statement. There is no need to draw a box around a statement (see Figures 4 and 13(a)).

CASE

In Dimensional Flowcharting the CASE statement is drawn as in Figure 5. The selection is controlled by the Conditional statement (Figure 6). Its semantics are that if ⟨Boolean expression⟩ is true then control passes on down the vertical line. If ⟨Boolean expression⟩ is false then control is prevented from continuing down the vertical branch. This is a general mechanism which is used in other constructs (see LOOP).

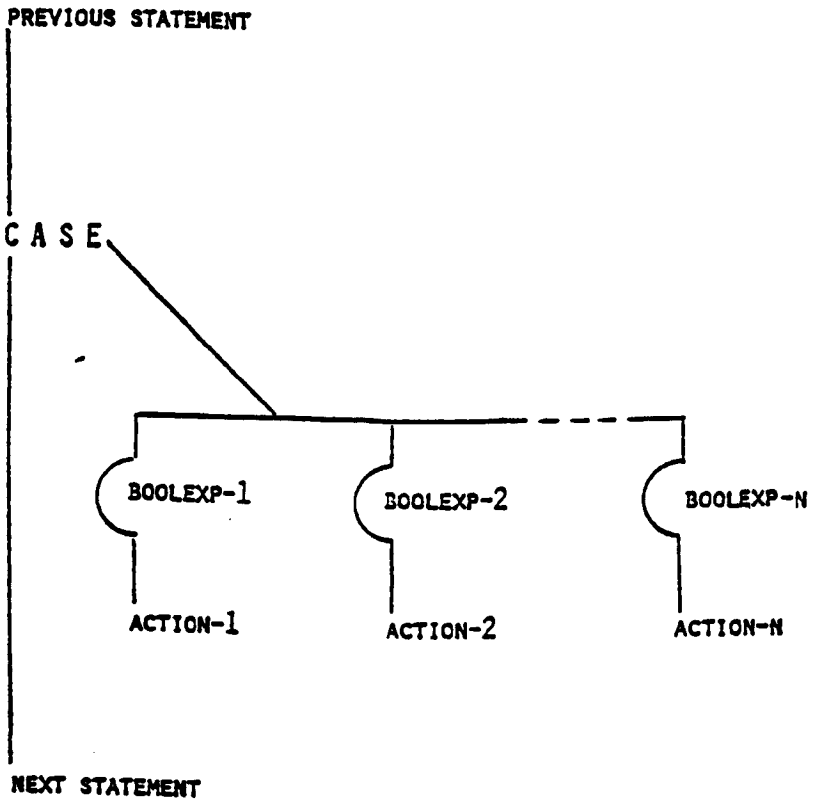


Figure 5. Dimensional CASE statement

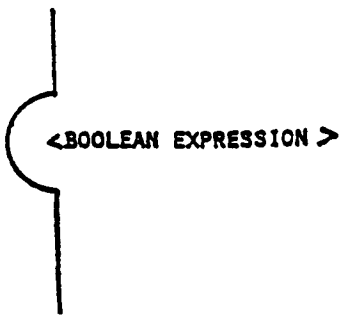


Figure 6. Conditional statement

Returning specifically to the CASE statement, its action is that if one or more vertical branches have true ⟨Boolexp-i⟩s then the corresponding ⟨action-i⟩s are executed. Control only passes to the statement after the CASE when all of these have finished execution. This shows the parallel dimension of the flowchart to its full advantage as control can be imagined to instantaneously flow along a horizontal line and down into each vertical line as though an infinite number of parallel processors were available to execute the statement. If all vertical branches are blocked by false ⟨Boolexp-i⟩s then the whole statement is terminated and control passes to the next sequential statement.

The simple Boolean blocking statement described above is good enough to represent purely serial algorithms (if the Boolean expressions are always mutually exclusive) or Dijkstra's guarded commands.[10] If a designer were to use this flowcharting method for constructing real parallel algorithms it is likely that he would add his own synchronization devices. The main point being made here is that the rigorous use of the two-dimensional page elegantly represents parallel code and sequential ordering.

Using the above scheme, IF-THEN-ELSE is drawn as in Figure 7. IF-THEN-ELSE is a simple CASE statement, the cases being '⟨Bool⟩ is true' and '⟨Bool⟩ is false'. Note that because of the explicit parallelism the ⟨then code⟩ and ⟨else code⟩ blocks must *both* be 'conditionally executed'.

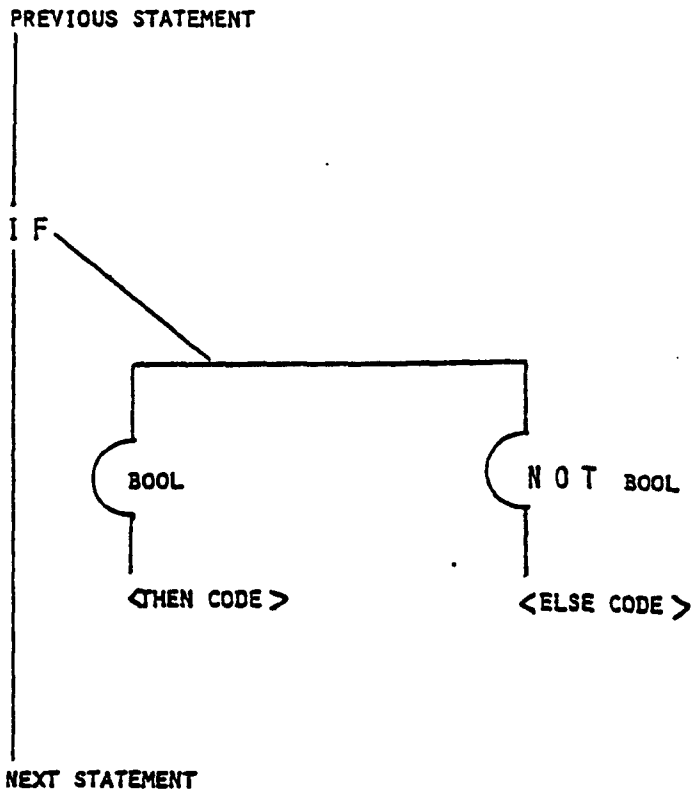


Figure 7. Dimensional IF-THEN-ELSE

This forces the programmer to consider fully all the implications of the ⟨else code⟩ operating on the (usually) large set of possibilities ⟨**not** Bool⟩, and is put forward as an advantage of the 'parallel' way of thinking.

Obviously, for normal, serial programming languages, the flowchart will be encoded using the conventional 'jump to ⟨else code⟩ if Boolean is false' technique.

LOOP

Figure 8(b) shows the Dimensional Flowcharting representations of a loop. Loops are repeated infinitely unless the loop body contains one or more UNTIL(WHILE) statements, one of which causes the loop to terminate when its Boolean expression is true (false). The UNTIL and WHILE constructs are simple adaptations of the Conditional Statement.

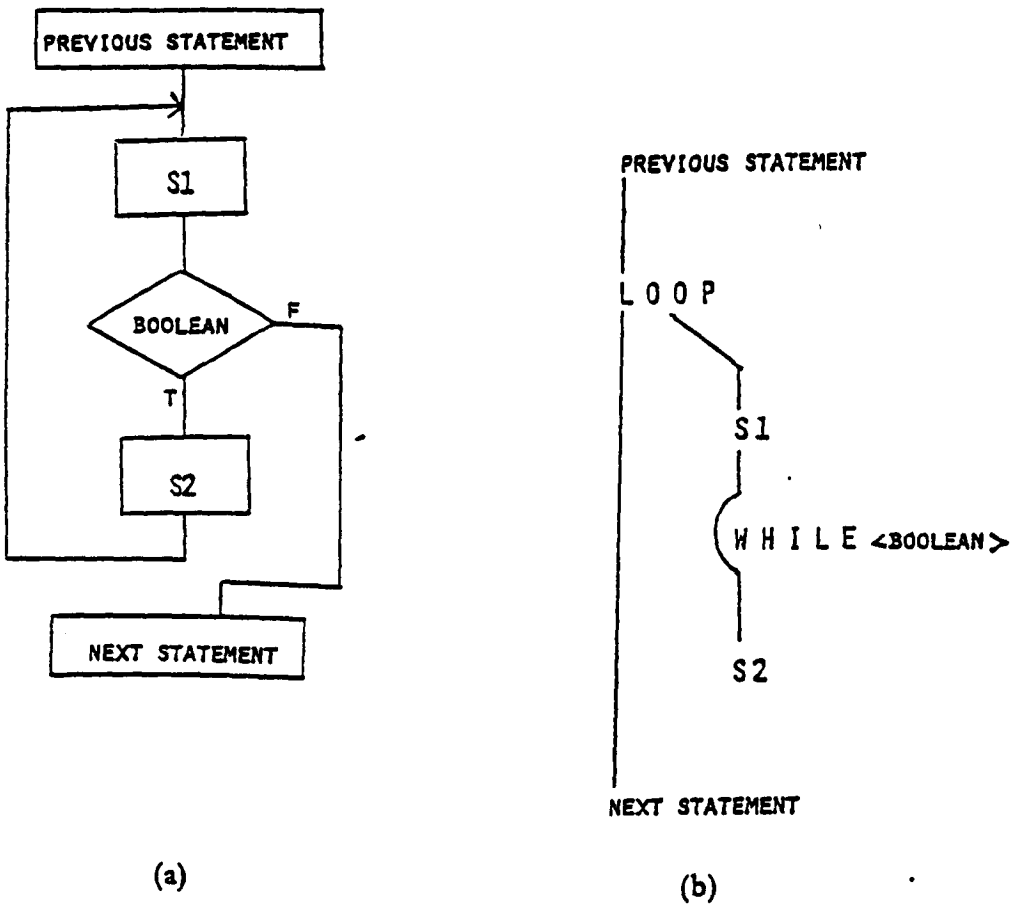


Figure 8. Contrasting representations of Dahl's loop,[6] (a) conventional, (b) dimensional

To help differentiate between a sequence of statements which is repeated (loop body) and one which is not it is sometimes useful to use the symbols '*' and '$\overline{=}$' which indicate repetition and completion respectively. Figures 9–13 and 16 illustrate their use. However, they can be omitted if the statement keyword (LOOP, CASE) is well defined.

Zahn's construct

In a reliable program all finite-by-design loops must be shown to terminate. A Termination Statement which specifically limits the number of iterations performed is a simple way to guarantee termination; with sensible repetition maxima this is also a valuable error checking

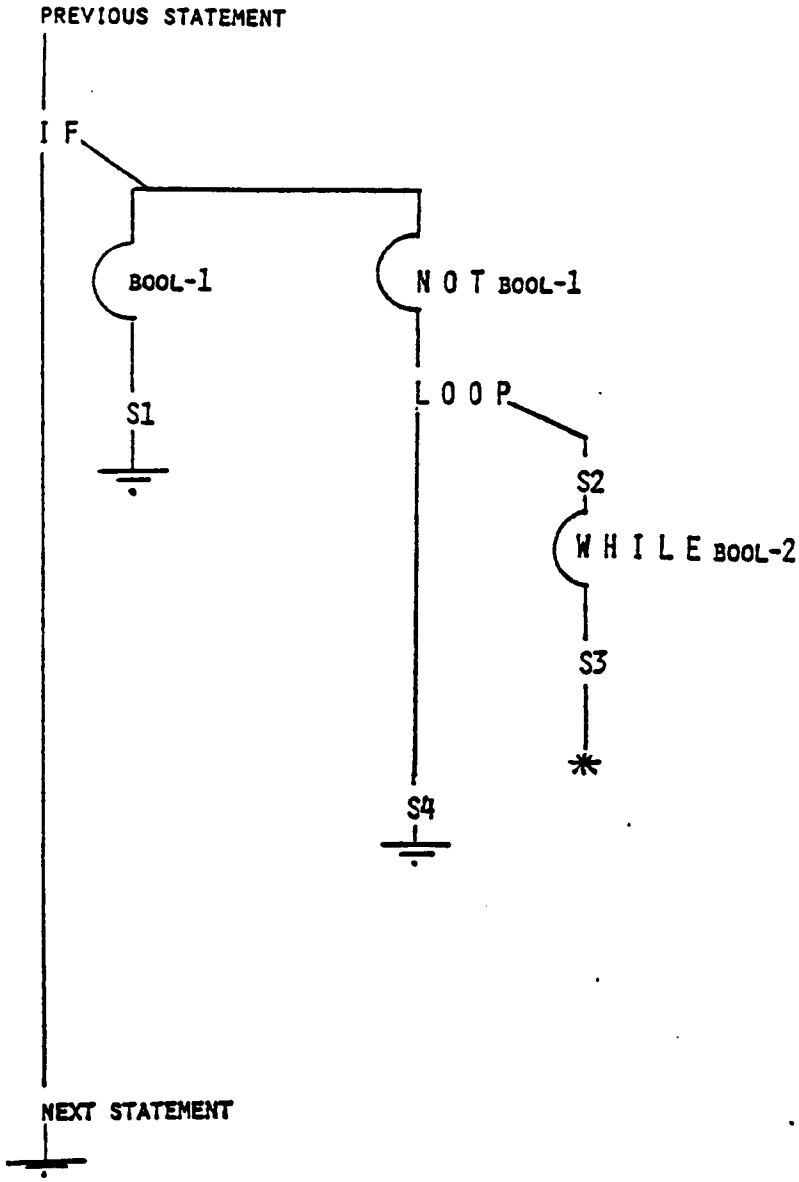


Figure 9. Repetition (*) and completion ($\overline{=}$) symbols

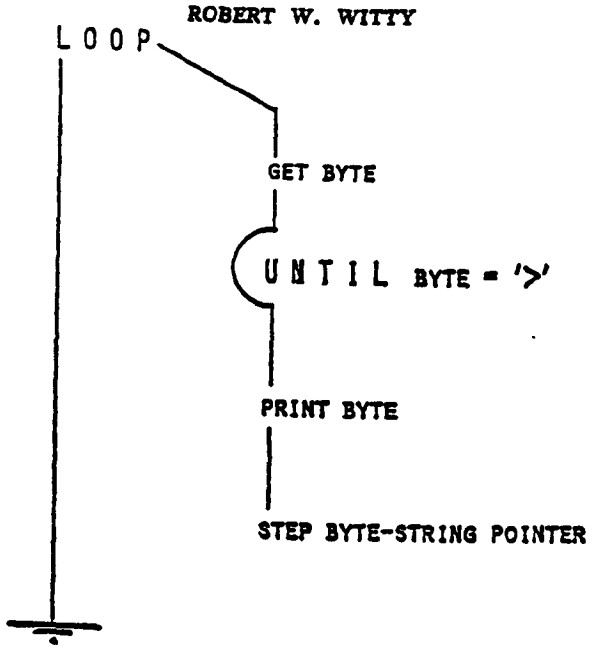


Figure 10. Print a line of text on a teletype

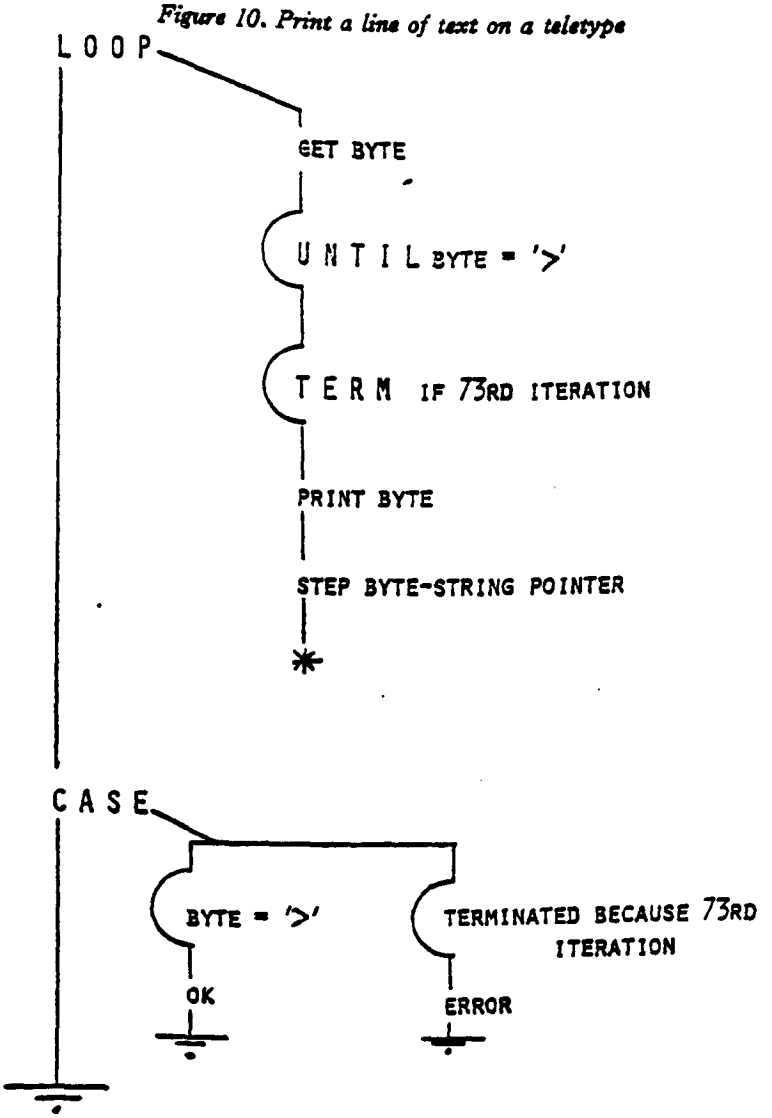


Figure 11. A reliable loop

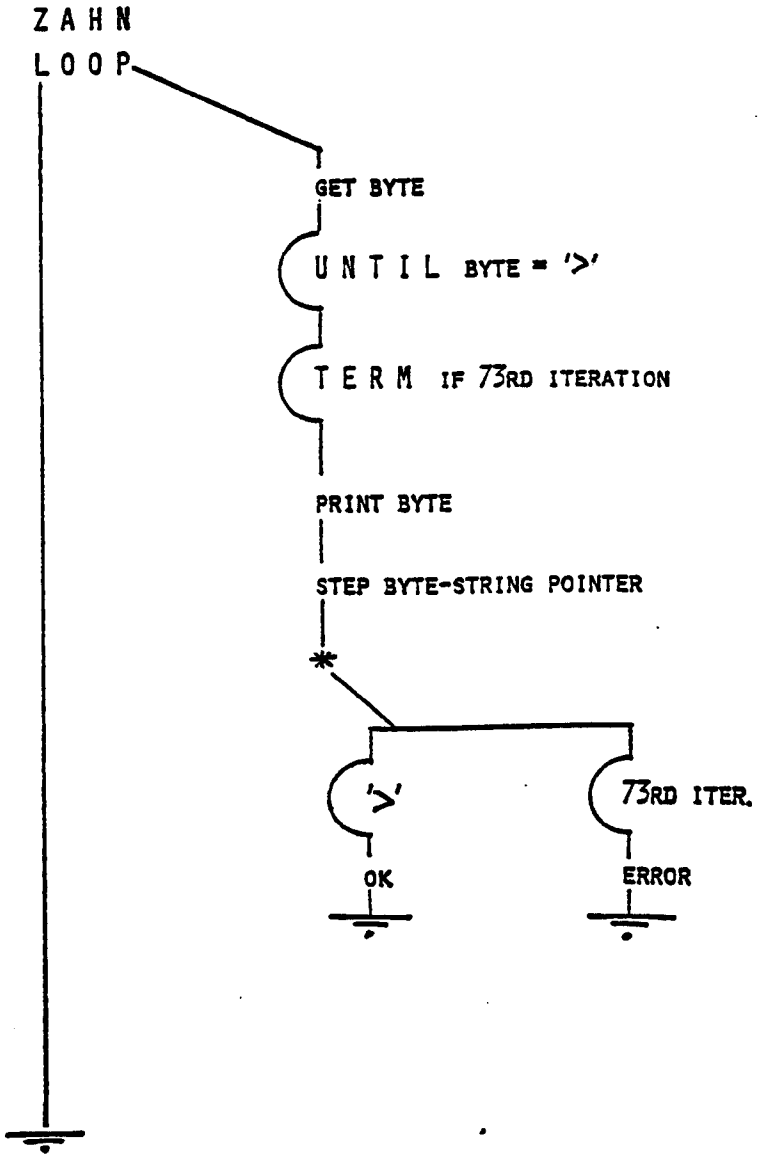


Figure 12(a). Zahn loop representation of a reliable loop

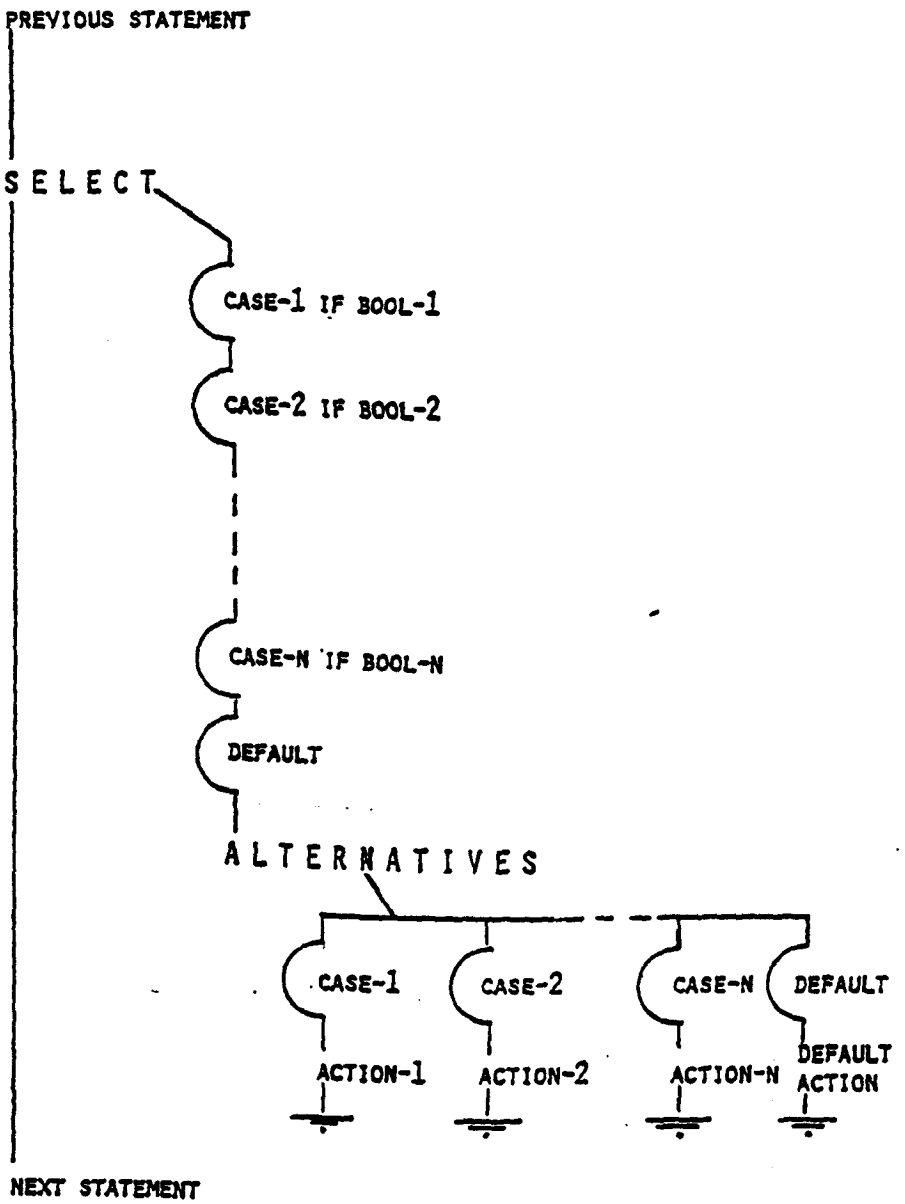
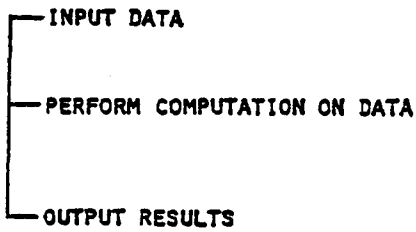
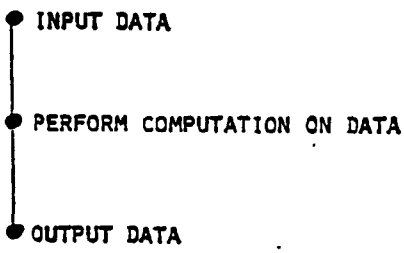


Figure 12(b). Dimensional representation of Zahn Selective CASE Statement

mechanism. If the routine in Figure 10 is only supposed to print out one line of text terminated by a '⟩' on a 72-character teletype then Figure 11 is a more reliable design. It is, though, inefficient and tedious to follow each loop with a CASE statement to determine the exact cause of termination, something already known within the loop body.

The Zahn loop[5–7] is a construction in which a CASE statement is an integral part of the loop, forming an epilogue. The flowcharting symbol '*' may now define both the end of the loop body and the start of the epilogue CASE statement as in Figure 12(a). The Zahn loop greatly simplifies the design of reliable loops and is easily compiled into efficient machine code (see Figures 25–27).





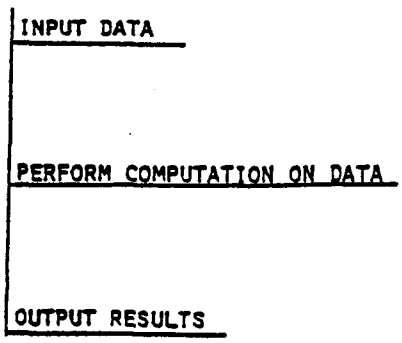
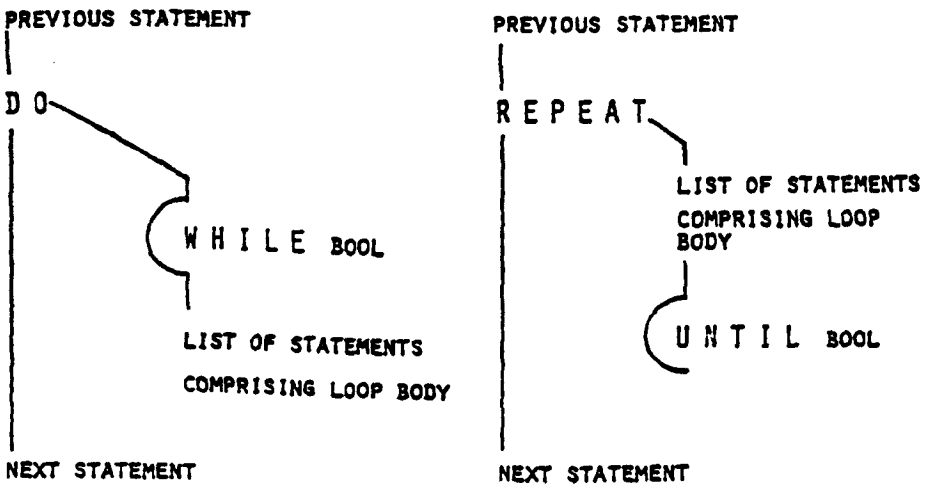


Figure 13(a). Alternative representations of statements

Zahn's construct may be applied to a block of statements which is not repeated and which consists entirely of conditional statements (Figure 12(b)). This gives a good representation of the conventional selective case statement, the serial nature of the selection mechanism contrasting well with the parallel nature of the alternative actions.



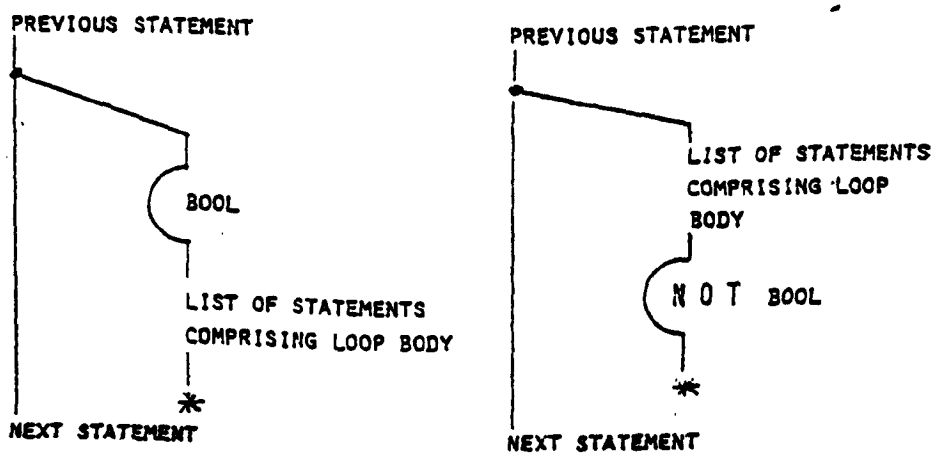


Figure 13(b). Repetition symbol versus keyword contrast

CONVENTIONAL AND 2-D FLOWCHARTS CONTRASTED

The preceding variety of representations (Figures 1–13) shows that Dimensional Flowcharting is flexible and adaptable. The diagrammatic forms of sequences, LOOPs and CASEs (the basic programming constructs) are easy to use and understand because Dimensional Flowcharting models high-level language control constructs and their disciplined control flows, whereas conventional flowcharting models require lower level operations. This point is illustrated by contrasting Figure 8(a) with Figure 8(b). In Figure 8(b) the disciplined sequential control flow is mandatory, and the repetition of a

loop body (S1; S2) a flow charting primitive. In Figure 8(a) the loop has to be synthesized from the 'Boolean' and the 'goto' primitives of conventional flowcharting causing the reader the unnecessary task of establishing firstly that the flowchart is indeed modelling a loop and not, say, an IF-THEN-ELSE, secondly exactly which statements comprise the loop body and thirdly whether or not the sequential discipline is maintained overall. It is this inadequately low level of primitives that causes some programmers to prefer designing by means of a high-level programming language rather than conventional flowcharting.

So far, Dimensional Flowcharting has used two dimensions, SCF and P, to represent alternative choices, parallel code and the sequential control flow discipline which will help the programmer to code and document his program. First though the program must be designed. Step-Wise Refinement can generate well disciplined programs. How can Dimensional Flowcharting help the software designer to use SWR?

STEP-WISE REFINEMENT

A property of SWR is that the most up-to-date design is expressed in terms of the lowest level reached so far. This means that, at worst, the derivations of large parts of the design are 'lost' as they are refined or, at best, can only be deduced from a study of separate flowcharts of the various intermediate stages (Figure 17). Is there a unified method of representation which will preserve what amounts to a record of the designer's thoughts?

Yes, these separate stages of refinement can be connected if the flowchart is given a *third* dimension, called the Refinement (R) (see Figure 14). Using this new dimension, the example given in Figure 17 now becomes Figure 18.

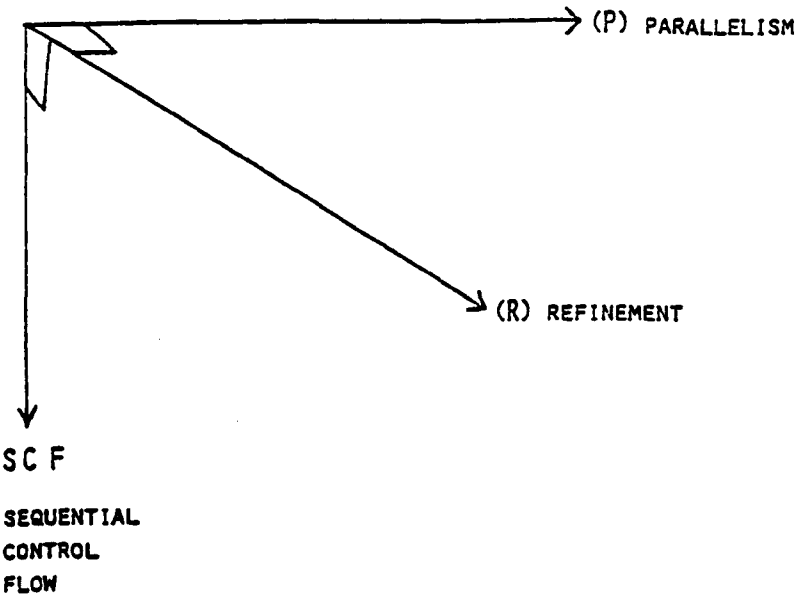


Figure 14. 3-D axes

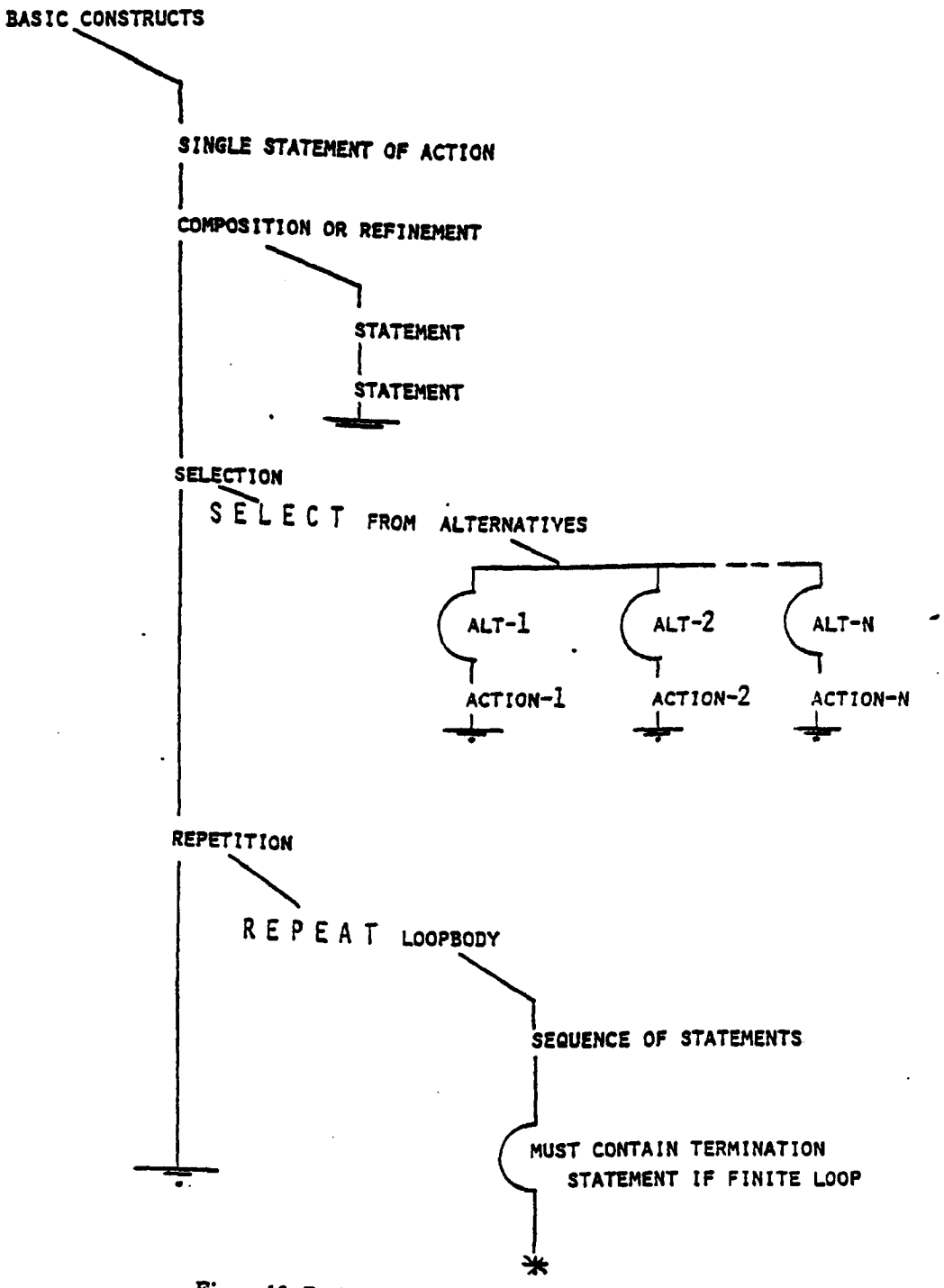


Figure 15. Basic dimensional flowcharting constructs

The SWR proceeds as follows: the initial problem specification 'solve quadratic equation' is broken down into the four steps 'input' to 'stop'. This second level, a flowchart in its own right, is an elaboration of the statement to which it is connected by a line in the Refinement dimension. Each level 2 statement is then separately elaborated to its level 3 version, generating three more flowcharts which are connected to their parents by their Refinement links. This process of refinement continues until a workable, detailed design is achieved (or is not, for design is an iterative process). With such three-dimensional (3-D) flowcharts the SWR design always exists as a whole, even when incompletely elaborated, not as the separate versions of Figure 17.

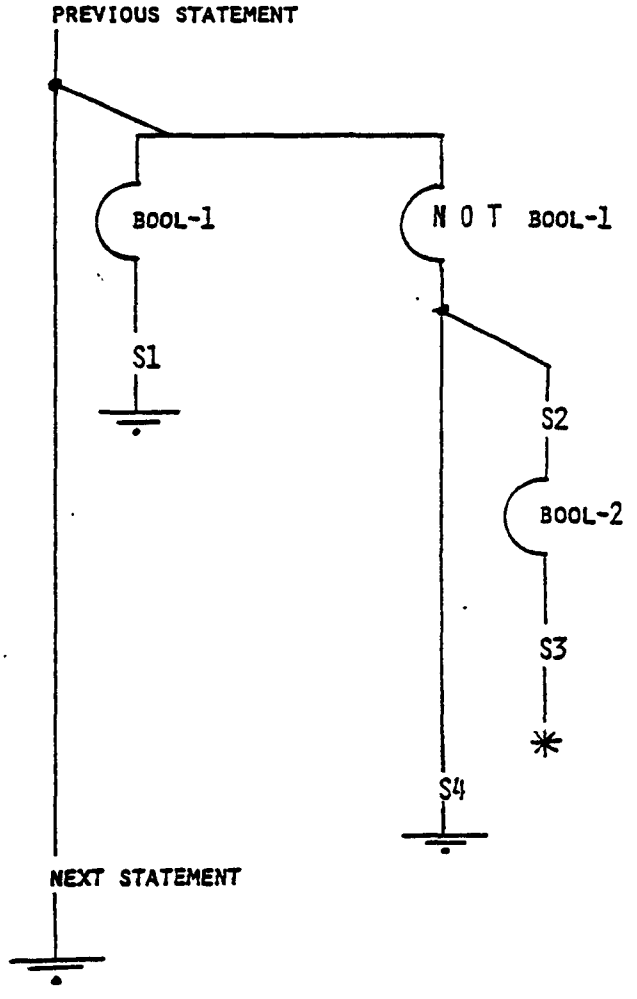


Figure 16. Figure 9 redrawn without keywords

A practical note

If one imagines that Figure 18 is someone else's flowchart then one can now study the program's design at any desired level of abstraction; one may study the design at *varying* stages of refinement as the area of interest changes, digging deeper down into the details of, say, 'input data' to see exactly what happens. Having once understood the action of the input phase one need never again go deeper than the 'input data' level to recall the program's action at this point.

```
SOLVE QUADRATIC EQUATION                                    } LEVEL 1

INPUT DATA
PERFORM COMPUTATION ON DATA                                 } LEVEL 2
OUTPUT RESULTS
STOP

READ(A)
READ(B)
READ(C)
if DISCRIM<0 then PRINTCH ('IMAGINARY ROOTS')
                  POSROOT:=NEGROOT:= 0
             else POSROOT:=(-B+ √(B²-4AC))/2A
                  NEGROOT:=(-B- √(B²-4AC))/2A                . LEVEL 3

PRINT(A)
PRINT(B)
PRINT(C)
PRINT(POSROOT)
PRINT(NEGROOT)
STOP
```

Figure 17. Step-Wise Refinement

If the reader copies Figure 18 and folds it as directed he will have a flowchart of levels 1 and 2 of Figure 13. He may now refine any of the level 2 statements by unfolding the paper beneath that statement, revealing the elaborated version. This process may be nested to any number of levels and is of practical significance. 3-D flowcharts of real systems are several feet long. By folding they are reduced to a convenient, workable size; only the specific area of interest need be unfolded or refined, to whatever depth is necessary, whilst the overall context is always visible at a higher level of abstraction.

A practical corollary to the folding technique occurs during the actual design process when there is insufficient room to refine a statement. The flowchart may be expanded at the point of refinement by the 'cut and paste' insertion of the additional material; this often forms a natural point to fold the flowchart. Experience has shown these techniques to be helpful in both the designing and the learning situations.

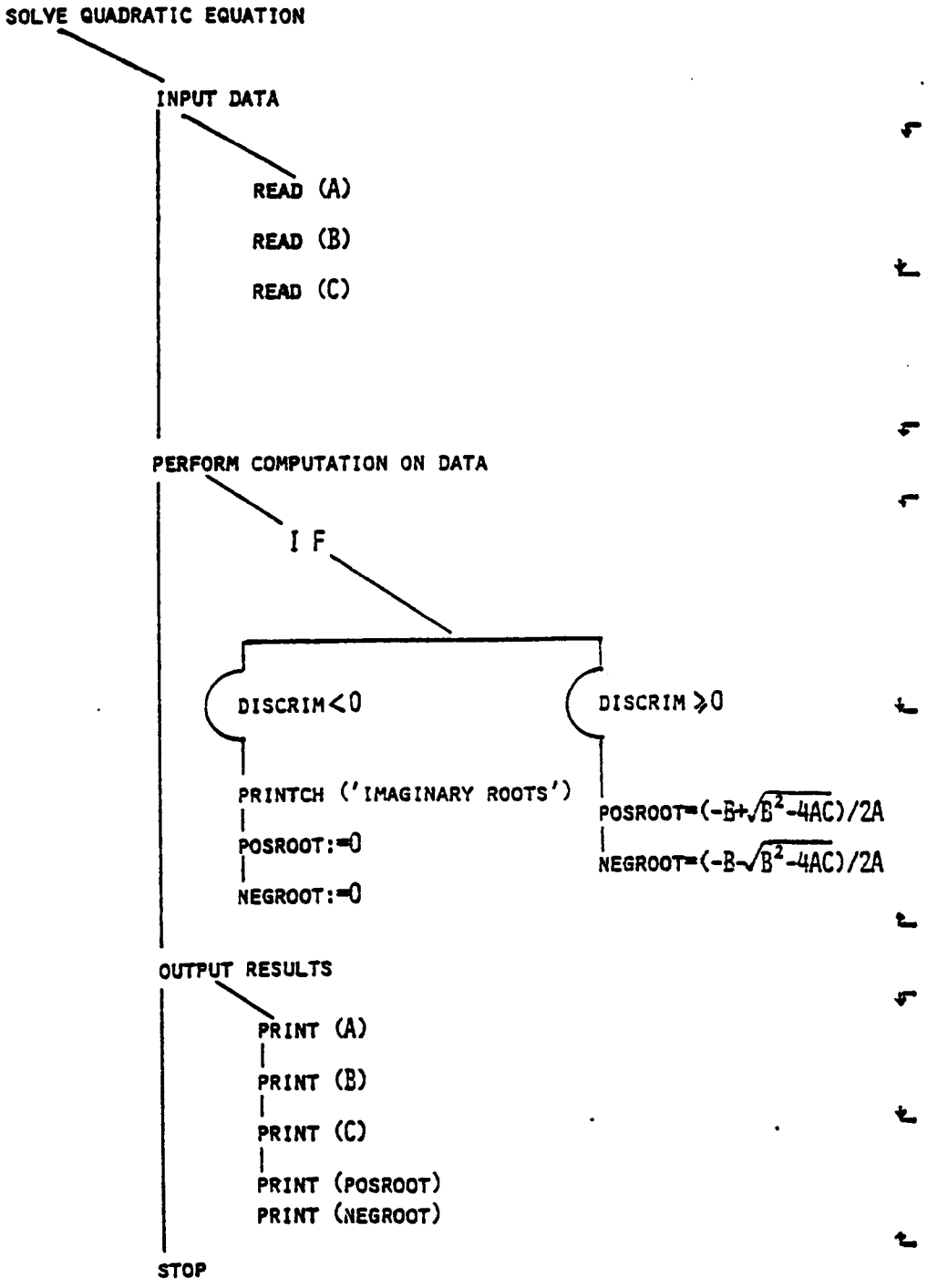


Figure 18. 3-D flowchart of Figure 17. Fold so that all the arrows point to the bottom of the page

BASIC FLOWCHARTING CONSTRUCTS

The basic flowcharting constructs of Composition, Selection and Iteration will now be re-examined in the light of the three dimensions, SCF, P and R, and the principle of Step-Wise Refinement. Figure 15 shows these basic constructs in their Dimensional form. A series of sequential statements is recognized by following the SCF axis. The refinement of a compound statement into such a serial sequence is found by proceeding from the compound statement along the Refinement axis.

The Selection construct selects from one or more 'alternatives' whose exact specifications are found in the refinement of 'alternatives'. This explains why the Refinement line is used in the CASE statement representation in Figure 5. Note that 'alternatives' is implied by the keyword 'CASE'.

Similarly the basic concept of Iteration is that a 'loop body' is repeated. The exact specification of the 'loop body' is a Refinement, so Figures 8 and 15 use the Refinement dimension when representing a loop body. Again 'loop body' is implied by the word 'LOOP'.

'REPEAT', 'CASE', 'LOOP', 'SELECT' and 'IF' are keyword statements which are refined. They aid understanding but are strictly redundant in that the operators $\{$, $\downarrow$, $\downarrow$ are all that is necessary to represent selection, repetition and completion respectively. Figure 16(a) is Figure 9 redrawn without the use of keyword statements, which are replaced by a null keyword or node marker. If used the keywords can allow '*' and '$\overline{=}$' to be included for clarity or omitted, i.e. implied.

FLOWCHART SYNTAX AND AUTOMATED DRAFTING

The process of Refinement is the ALGOL-like discipline that every program is a single (compound) statement which is recursively split up into a sequence of statements. A direct

```
begin
  comment SOLVE QUADRATIC EQUATION;
  comment INPUT DATA;
  READ(A);
  READ(B);
  READ(C);
  comment PERFORM COMPUTATION ON DATA;
  if (B*B-4*A*C)<0
    then begin
         PRINTCH('IMAGINARY ROOTS');
         POSROOT:=NEGROOT:=0;
         end
    else begin
         POSROOT:=(-B+SQRT(B*B=4*A*C))/2*A;
         NEGROOT:=(-B-SQRT(B*B-4*A*C))/2*A;
         end;
    comment OUTPUT RESULTS;
    PRINT(A);
    PRINT(B);
    PRINT(C);
    PRINT (POSROOT);
    PRINT (NEGROOT);
    comment STOP;
end;
```

Figure 19. Linearized comments

benefit of introducing such a discipline into the design methodology is that a context-free grammar can be defined which will generate 3-D flowcharts. This property means a flowchart can be shown to conform to the design methodology and to project standards which can be built into the grammar. This process can be automated by inputting a machine-readable version of the flowchart to a syntax analyser. It can be shown from the grammar that a flowchart is a tree. A simple tree-walk will produce a linear machine-readable version if the various straight lines and flowcharting symbols are encoded as unique identifiers. Such a tree-walk is similar to unparsing.[8]

```
SOLVE QUADRATIC EQUATION:
  begin
  INPUT DATA:
    begin
    READ(A);
    READ(B);
    READ(C);
    end;
  PERFORM COMPUTATION ON DATA:
    begin
    if (B*B-4*A*C)<0
      then begin
        PRINTCH('IMAGINARY ROOTS');
        POSROOT:=NEGROOT:=0;
        end
      else begin
        POSROOT:=(-B+SQRT(B*B-4*A*C))/2*A;
        NEGROOT:=(-B-SQRT(B*B-4*A*C))/2*A;
        end;
    end;
  OUTPUT RESULTS:
    begin
    PRINT(A);
    PRINT(B);
    PRINT(C);
    PRINT (POSROOT);
    PRINT (NEGROOT);
    end;
STOP:
end;
```

Figure 20. Hierarchically commented source program

This human process is quick and easy, and the output matches the original drawing (Figure 21). The linear tree walk can then be input to a syntax analyzer, such as is created by the TREE-META[9] compiler–compiler program in Figure 22, for validation and a straightforward recursive graphical algorithm can be constructed to produce high quality output on a plotting device by exploiting the grammar rules to draw the flowchart left to right, and top to bottom, which makes the formatting very easy (it is 'context-free'). The ease with which drafting may be automated is another advantage of Dimensional Flowcharting.

```
"SOLVE QUADRATIC EQUATION"
  \
   "INPUT DATA"
     \
      "READ(A)"
      "READ(B)"
      "READ(C)"
        #
   "PERFORM COMPUTATION ON DATA"
         \
          "IF"
            \
             ?DESCRIM < 0?
             "PRINTCH('IMAGINARY ROOTS')"
             "POSROOT:=0"
             "NEGROOT:=0"
               #
             ¬
             ?DISCRIM>0?
             "POSROOT:=(-B+SQRT(B*B-4*AC))/2*A"
             "NEGROOT:=(-B-SQRT(B*B-4*A*C))/2*A"
               #
        #
   "OUTPUT RESULTS"
         \
          "PRINT(A)"
          "PRINT(B)"
          "PRINT(C)"
          "PRINT (POSROOT)"
          "PRINT (NEGROOT)"
            #
   "STOP"
    #
```

Figure 21. Machine-readable version of 3-D flowchart. Notes: (1) ¬ represents horizontal line, the P dimension; (2) \ represents angled line, the R dimension; (3) ≠ represents the ≒ symbol; (4) vertical lines, the SCF dimension, are implied by the juxtaposition of statements

```
•META DIMFLOWCHART
DIMFLOWCHART = NAME '\' FLOWC ;
FLOWC        = SERIAL $ (SERIAL) '#' ;
SERIAL       = STATEMENT (REFINE / .EMPTY) $ (PARALLEL) ;
REFINE       = '\'  FLOWC;
PARALLEL     = '¬'  FLOWC;
STATEMENT    = ACTION / CONDITIONAL
ACTION       = '"' TEXT '"' ;
CONDITIONAL= '?' BOOLEAN EXPRESSION OR TEXT '?' ;
•END
```

Figure 22. Tree-Meta syntax analyser definition for recognizing machine-readable 3-D flowcharts. Notes on Tree-Meta: (1) $( ) — zero or more of; (2) 'string' — string must appear in input; (3) / — alternative; (4) ; — Tree Meta rule terminator; (5) '.EMPTY' — the null symbol, always recognized

DESIGN AND CONSTRUCTION

Tackling a problem by creating a 3-D flowchart via SWR produces a logical solution not a working program. This result is analogous to the hardware engineers' Logic Diagram. Actually constructing a program from the logical solution introduces a new set of practical problems which vary with the peculiarities of particular source languages, compilers and machines. Considerations such as addressing mechanisms, core sizes, macros and procedures being defined before being called, and separate compilation for individual subroutines cause a working source program to vary considerably from the neat order of the abstract logical solution.

The second phase problem of implementing a logical solution in terms of an actual programming language should itself be tackled using the 3-D SWR method.[2] This leads to a neater source code program, the 3-D flowchart of which is an 'engineering drawing' or map of the way the source code is *physically* constructed (see Figure 23). This division into Logical Solution (Logic Diagram) and executable source code (Physical Construction) is exactly analogous to the way hardware engineers must produce circuit board, component and wiring layouts to implement their Logic Diagrams, and the two drawings used together are as invaluable when working on a large piece of software as they are for hardware.

Linearized comments

If one regards the higher levels of abstraction in a 3-D flowchart as comments about the lowest level, the code, then this hierarchy of comments may be linearized and incorporated into the source code text, whence the correspondence between the flowchart and the source code will be exact in the refinement sense (see Figures 18 and 19).

Use of the Logic Design flowchart in conjuction with the Physical Construction flowchart and the source text, with its linearized SWR comments, greatly simplifies the problem of relating the physical program code to its logical design and action.

Scoped comments

Comparison of Figures 18 and 19 highlights a weakness of conventional comment statements, namely their undefined scope. In a large program it is always clear at what point a

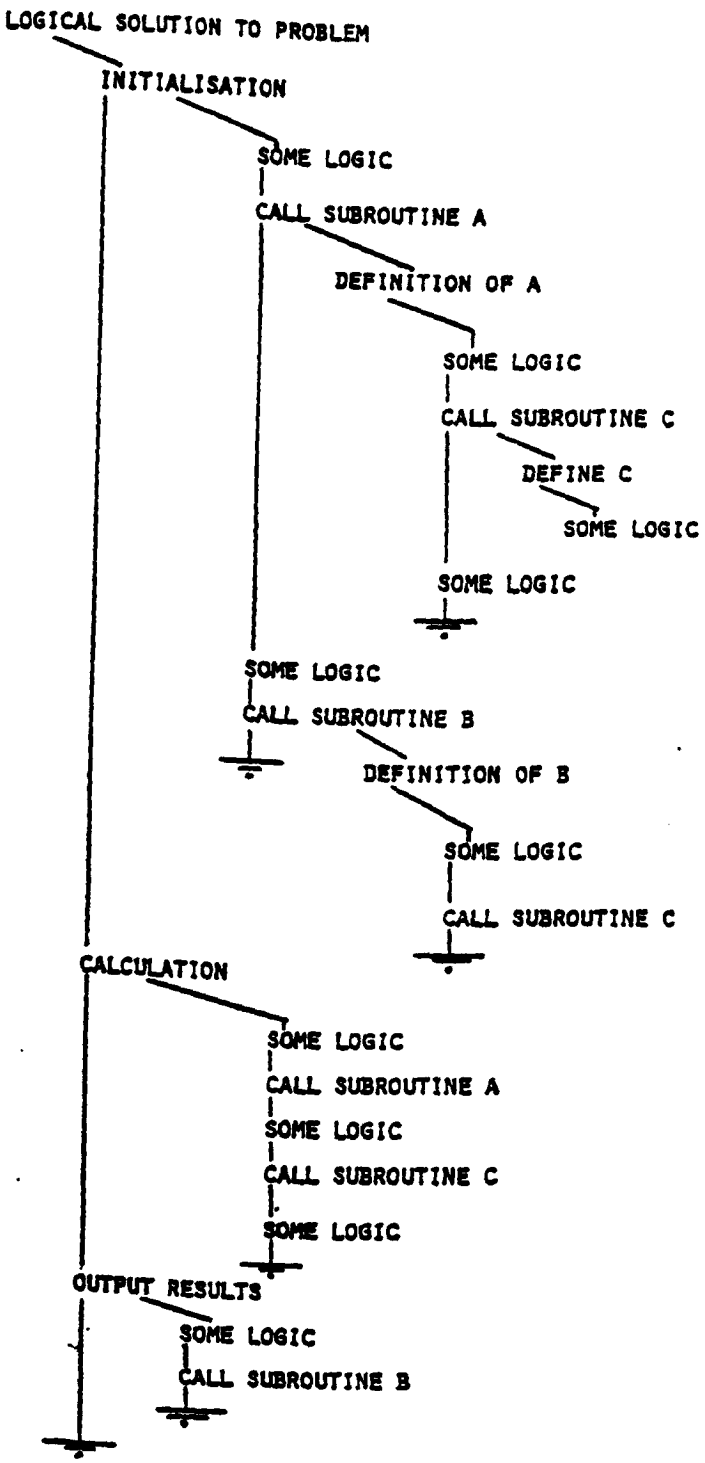


Figure 23. Simplified Logic Diagram

comment's relevance begins, but is not always as clear where its relevance ends. This is because, conceptually, the Refinement dimension is 'lost' when a 3-D flowchart is transformed into a 1-D text string. (Note that the P dimension is similarly lost.) If the source

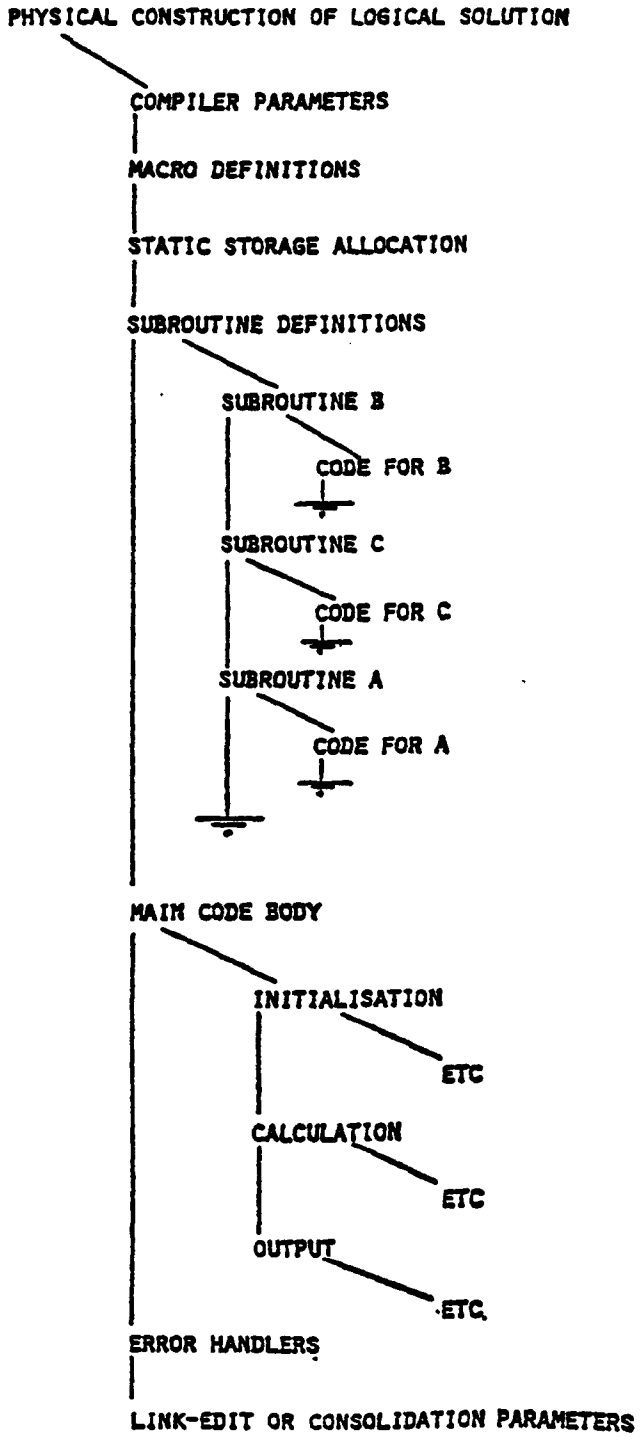


Figure 24. Simplified Physical Construction

38

language is block-structured then it is possible to map a 3-D flowchart so as to preserve the R dimension as in Figure 17. (Note that if the ALGOL symbols **'begin'** and **'end'** are replaced by the symbols '\' and '=' then the pictorial resemblance between flowchart and source code is excellent.)

Figures 21 and 22 outline a general system for automating the drawing of 3-D flowcharts; note the similarity between Figure 20, the ALGOL version and Figure 21, the flowchart tree walk (**'begin'** ≡ '\' and **'end'** ≡ '#'), the SCF and R dimensions are exactly the same, only the P dimensions differ, though it is easy to see that they are equivalent. The ALGOL version contains all the necessary information to regenerate the 3-D flowchart which created it, and it would be a simple task to build an automatic 3-D flowchart generator for Algol 60 programs constructed by the hierarchically commented compound statement technique.

The comment-code hierarchy of Figure 20 is only made possible through a programming trick exploiting ALGOL's block structure. To build comment-code hierarchies in non-block structured languages the syntax of comment statements must be slightly enhanced to include some equivalents of the '\' and '#' in Figure 20 to allow Scoped Comments to be a proper language component, not an optional programming trick (see Figure 24).

If a program is written in a language with scoped comments, in such a way that some of the scoped comments do not yet have their refinements, their actual source code, attached, then such a program, though incomplete and uncompilable, can be checked to see that it conforms to the syntax of the language and a 3-D flowchart of it produced if the unrefined comments are regarded as refinement stubs standing in lieu of actual code. Such a system would allow the evolving design of a program to be held as a simple machine-readable file.

The software designer can now use the 3-D flowcharting technique on paper to evolve his ideas. A simple tree-walk of this paper design will enable him to produce, at intervals, an incomplete program using scoped comments and refinement stubs from which the language flowchart generator will produce for him a neat copy of his original flowchart, the Logic Diagram. The Physical Construction may be similarly designed and produced. When the refinement of the Logic Diagram is complete and the design's Physical Construction actually works then the flow chart generator will be able to produce two neat, accurate flowcharts for documentation purposes. Thereafter, should the program be changed, it is a painless task for the maintenance programmer to produce new, up-to-date flowcharts. If the flowcharts are drawn on say, microfiche, then they can be kept to form a valuable but compact history of the program's development. Such a system is being developed by the author to help construct assembler programs for a PDP-15-like machine. The first programs to be designed using 3-D flowcharts[8] were hand-coded into 10,000 assembler statements using 'templates' to hand-compile the control constructs from their 3-D representations.

The success of this approach has prompted the current development of software tools to automatically generate 3-D flowcharts and assembler code from a single, simple source language, which includes Scoped Comments, IF-THEN-ELSE, Zahn loops, Zahn case statements and assembler statements, to help with the design and construction of a larger software project. Figures 25–27 are an example.

This 3-D programming language enables the Physical Construction map to be automatically produced from the source code program. If the logical solution is also expressed in terms of the 3-D language then it is a simple matter to maintain both versions so that the maintenance programmer has up-to-date copies of both the Logic Diagram (3-D source and flowchart) and the Physical Construction (3-D source and flowchart) to help him. How many maintenance programmers have to build their own personal versions of the

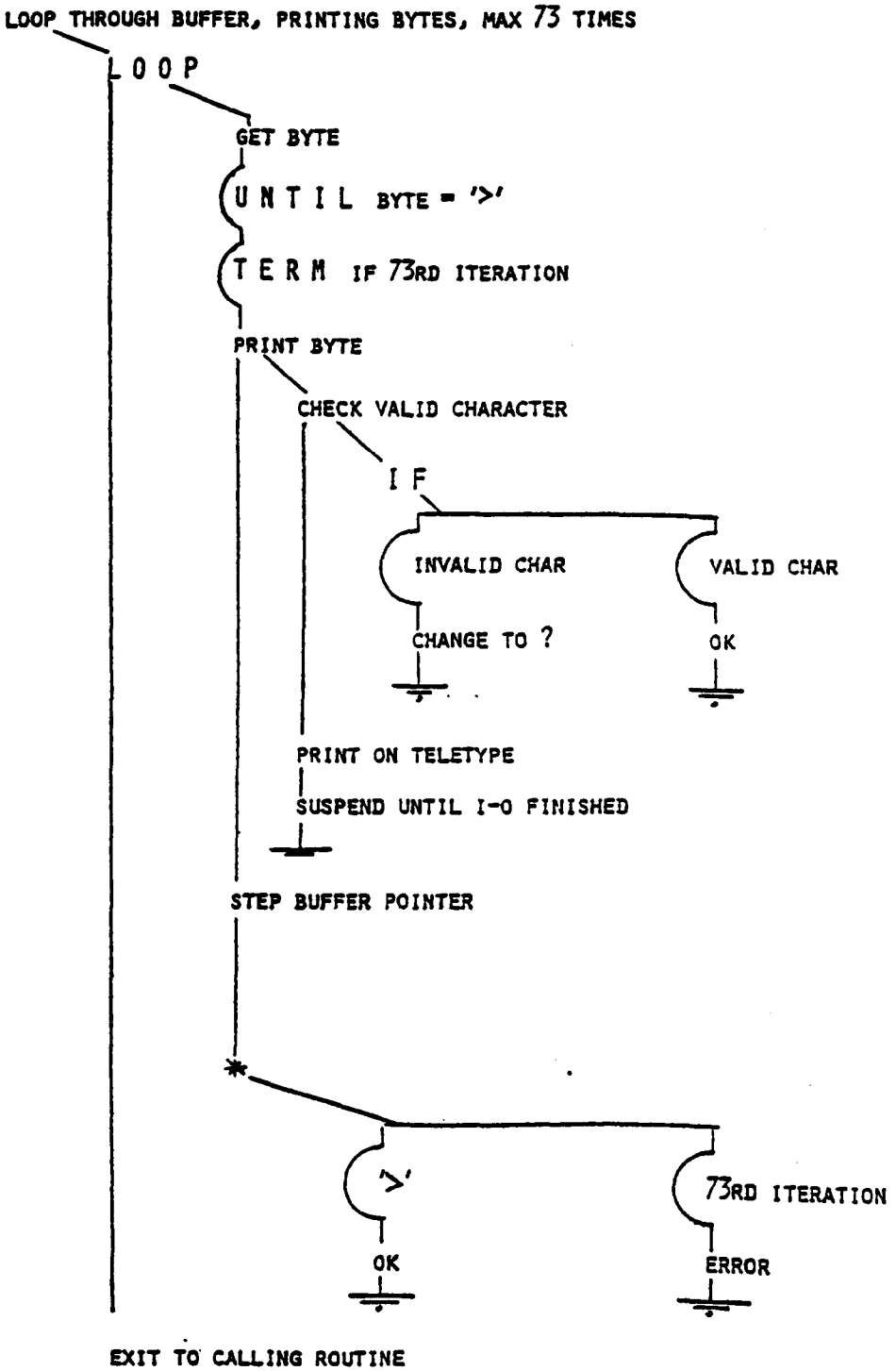


Figure 25. Logic Diagram of teletype printing routine, coded in Figure 26

```
.PROG TTY3D
      \
"ROUTINE TO PRINT STRING ON TTY"
"STRING POINTED TO BY TBUFF"
"STRING TERMINATED BY > WHICH IS NOT PRINTED OUT "
"ONE CHAR PER WORD"
"INVALID CHARS PRINTED AS ? "
      \
'TTYOUT,  XX    / ROUTINE ENTRY POINT'
      \
"LOOP THRU THE BUFFER, MAX 73 SO NOT OFF END OF LINE"
      \
.MAXLOOP MAX73  := 73 .ITERS
.LOOP
    .WITH .TERMS EOSTR, TMNY .ENDW

    "GET BYTE"
        \
'       LAC I TBUFF'
'       DAC BYTE'
         #

    .TERM EOSTR .IFF BYTE .EQ GTSIGN .ENDE

    .TERM TMNY .IFF .DONE MAX73 .TIMES .ENDE

    "PRINT BYTE"
        \
       "CHECK VALID CHAR"
            \
           .IF "INVALID CHAR" \  [BYTE .GT (177)] .OR [BYTE .LT 0] #
              .THEN
                 "CHANGE TO ? "
                     \
            '       LAC QMARK'
            '       DAC BYTE'
                     #
              .ELSE
                 .OK
           .ENDI
            #

       "PRINT ON TTY"
            \
'       LAC BYTE'
'       TLS'
         #

       "SUSPEND UNTIL I/O FINISHED"
            \
'       TSF'
'       JMP .-1'
         #
      #
```

Figure 26. Teletype printing routine, an example of a macro-assembler program incorporating Scoped Comments, comment-code hierarchy and Sequential Control Flow discipline

```
    "STEP POINTER"
        \
'       ISZ TBUFF'
'       NOP'
        #

.REPEAT
    .SIT EOSTR .CAUSES .OK .ENDS
    .SIT TMNY  .CAUSES '      JMS TOOLNG' .ENDS
.ENDL
      #
   #

'       JMP I TTYOUT    / EXIT TO CALLER'


"LOCAL STORAGE"
      \
    'BYTE,      0'
    'GTSIGN,    .ASCII/>/ '
    'QMRK,      .ASCII/?/ '
      #


"GLOBAL REFERENCES"
      \
   "TBUFF   - POINTS TO START OF STRING, CHANGED BY THIS ROUTINE"
   "TOOLNG  - PRINTS ERROR MESSAGE, LINE TOO LONG"
      #


      #
      #
.ENDP TTYOUT

*****
```

Figure 26 (cont.)

```
 0 /OBJ;DRIL TTY3D   OBJECT TTY3D
 1 /
 2 /
 3 /
 4 /
 5 /
 6 /
 7 /:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 8 /
 9 /
10 /   DRIL MACRO DEFINITIONS
11 /
12 .INSERT ROB;DRIL MACROS
13 /
14 /   END OF DRIL MACROS
15 /
16 /
17 /:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
18 /
19 /
20 /
21 /
22 /
23 /
24 /:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
25 /
26 /
27 /
28 / BEGIN USER CODE
29 /.
30 /
31 /ROUTINE TO PRINT STRING ON TTY
32 /STRING POINTED TO BY TBUFF
33 /STRING TERMINATED BY > WHICH IS NOT PRINTED OUT
34 /ONE CHAR PER WORD
35 /INVALID CHARS PRINTED AS ?
36 /
37 TTYOUT,  XX     / ROUTINE ENTRY POINT
38 /
39 /LOOP THRU THE BUFFER, MAX 73 SO NOT OFF END OF LINE
40 /
41 /
42 /
43 / SET LOOP LIMIT COUNTER VARIABLE
44        LAC (73.)
45        TCA
46        DAC #MAX73
47 /
48 /
49 / START LOOP
50 LB1,
51 /       .WITH   .EXITS EOSTR, TMNY
52 /
53 /GET BYTE
54 /
55        LAC I TBUFF
56        DAC BYTE
```

Figure 27. Assembler code produced from Figure 26

```
57 /
58 /
59 /
60 / EXIT?
61        SKPEQ BYTE,GTSIGN
62        SKP       / SKP IF FALSE
63        JMP EOSTR       / EXIT EOSTR
64 /
65 /
66 /
67 /
68 / EXIT?
69 /   STEP + TEST LOOP LIMIT COUNTER, (AND EXIT?)
70        ISZ MAX73
71        SKP       / SKP IF FALSE
72        JMP TMNY       / EXIT TMNY
73 /
74 /
75 /PRINT BYTE
76 /
77 /CHECK VALID CHAR
78 /
79 /
80 /
81 / IF
82 /INVALID CHAR
83        SKPGT BYTE,(177)
84        SKP       / FALSE - TRY NEXT
85        JMP LB2+1       / TRUE
86        SKPLT BYTE,(0,)
87 LB2,       / LB2+1  IF TRUE
88        JMP LB3       / JMP IF FALSE
89 / THEN
90 /CHANGE TO ?
91 /
92        LAC QMARK
93        DAC BYTE
94        JMP LB4
95 / ELSE
96 LB3,
97 /       OK,NULL STATEMENT
98 LB4,
99 / END OF IF
100 /
101 /
102 /PRINT ON TTY
103 /
104       LAC BYTE
105       TLS
106 /
107 /SUSPEND UNTIL I/O FINISHED
108 /
109       TSF
110       JMP .-1
111 /
112 /STEP POINTER
113 /
```

Figure 27 (cont.)

```
114         ISZ TBUFF
115         NOP
116 /
117 /
118 /
119 / REPEAT LOOP
120         JMP LB1
121 /
122 /
123 / SITUATION EOSTR
124 EOSTR,
125 /        OK,NULL STATEMENT
126         JMP LB5         / LEAVE BLOCK,LOOP
127 /
128 /
129 / SITUATION TMNY
130 TMNY,
131         JMS TOOLNG
132 LB5,       / END OF BLOCK,LOOP
133 / END LOOP
134 /
135 /
136         JMP I TTYOUT     / EXIT TO CALLER
137 /
138 /
139 /
140 /LOCAL STORAGE
141 /
142 BYTE,       0
143 GTSIGN,     .ASCII/>/
144 QMRK,       .ASCII/?/
145 /
146 /
147 /GLOBAL REFERENCES
148 /
149 /TBUFF    - POINTS TO START OF STRING, CHANGED BY THIS ROUTINE
150 /TOOLNG   - PRINTS ERROR MESSAGE, LINE TOO LONG
151 /
152 /
153 /
154 /
155 /
156 CONSTANTS VARIABLES
157 /
158 /
159 / END USER CODE
160 /
161 /
162 /
163 /
164 / END OF PROG TTYOUT
165 /
166 /
167 START
```

Figure 27 (cont.)

missing Logic Diagram and Physical Construction by tedious 'bottom-up' readings of the source code? Not an easy job when the program is an operating system written in assembler. Is it possible to design a programming language which will express the top down logical solution, unobscured by physical considerations but including them, so that the Logic Diagram and the Physical Construction may be produced from a single source program?

SUMMARY OF ADVANTAGES OF DIMENSIONAL FLOWCHARTING

1. Quick and easy to draw, by hand or machine, because drawn left to right, top to bottom, i.e. naturally.
2. Grammar ensures flowcharts are well formed.
3. Grammar provides a mechanism for enforcing project standards.
4. Easy to convert to machine-readable form via simple 'tree-walk' method.
5. Grammar makes automatic drafting very easy.
6. Statements may be any length.
7. Statements are not constrained to fit into boxes.
8. No need for templates.
9. Few special symbols.
10. Adaptable to any source code and machine.
11. Easy to introduce new features such as control constructs and synchronization devices. Models high-level language constructs.
12. Shows inherent parallelism which is often not obvious from the source code.
13. 'Automatically' ensures well disciplined program design. Cluttered logic is prevented.
14. Encourages and facilitates SWR design.
15. Makes it easy for a third party to understand how the design arrived at because SWR explicit.
16. One flowchart shows all three dimensions at once, making it possible (and easy) to visualize the whole program at varying levels of abstraction, especially if the flowchart is folded according to its refinement.
17. When both the logical design and the physical construction diagrams are drawn as large size 'engineering drawings' and when all the levels of abstraction are 'linearized' into the source code as comments, or a 3-D programming language is used, then these documents work in unison to greatly improve and speed the understanding of the program by a third party (and the designer himself).

CONCLUSION

Dimensional flowcharts are easy to draw by hand or machine and their syntax can be checked automatically. They help the designer to proceed via the Step-Wise Refinement Method, they help the programmer to produce well disciplined code and they help the maintenance programmer to understand the finished product.

REFERENCES

1. E. W. Dijkstra, O.-J. Dahl and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972.
2. R. W. Witty, *FR80 DRIVER Software Construction*, Atlas Computer Laboratory, Oxon, 1975.
3. J. Rushby and R. W. Witty, *FR80 Logging System*, Atlas Computer Laboratory, Oxon, 1975.
4. G. M. Weinberg *et al.*, 'IF-THEN-ELSE considered harmful', *SIGPLAN Notices*, 34–43 (1975).
5. C. T. Zahn, 'A control statement for natural top-down structured programming', *Lecture Notes Comput. Sci.* **19**, 170–180 (1974).

6. D. E. Knuth, 'Structured programming with *goto* statements', *Comput. Surv.* **6**, No. 4, 261–301 (1974).
7. D. E. Knuth and C. T. Zahn, 'Ill chosen use of "event"', *Communs Ass. Comput. Mach.* **18**, No. 6, 360 (1975).
8. V. Donzeau-Gouge *et al.*, 'A structure-oriented program editor', *Int. Computing Symp.*, North Holland Publishing Co., Amsterdam, 1975.
9. F. R. A. Hopgood, *The 1906A Tree-Meta Manual*, Atlas Computer Laboratory, Oxon, 1974.
10. E. W. Dijkstra, 'Guarded commands', *Communs Ass. Comput. Mach.* **18**, No. 8, 453–457 (1975).

10.2. The 'B2D' Example Program

10.2.1. Hand Drawn Detailed Design

- see sample marked 'Hand Drawn B2D'
  in wallet attached to rear cover.

10.2.2. Machine Drawn Detailed Design

- see plotter output marked 'DESN'
  in wallet attached to rear cover.

10.2.3. Static Analysis - Quality Control Histograms

CHARS. PER STATEMENT

```
    BIGGEST VAL=       39 SCALED BY=        1 WIDTH=      100
  39       <     1!****************************************
  20       1-    2!********************
  23       3-    4!************************
  30       5-    6!******************************
   7       7-    8!*******
  34       9-   10!**********************************
  25      11-   12!*************************
   8      13-   14!********
   3      15-   16!***
   8      17-   18!********
   7      19-   20!*******
   6      21-   22!******
   3      23-   24!***
   3      25-   26!***
   5      27-   28!*****
  11      29-   30!***********
   8      31-   32!********
   6      33-   34!******
   0      35-   36!
   5      37-   38!*****
   3      39-   40!***
   2      41-   42!**
   3      43-   44!***
   5      45-   46!*****
   0      47-   48!
   0      49-   50!
   2      51-   52!**
   0      53-   54!
   0      55-   56!
   0      57-   58!
   0      59-   60!
   2      61-   62!**
   1      63-   64!*
   0      65-   66!
   0      67-   68!
   0      69-   70!
   0      71-   72!
   0      73-   74!
   0      75-   76!
   0      77-   78!
   0      79-   80!
   0      81-   82!
   0      83-   84!
   0      85-   86!
   0      87-   88!
   0      89-   90!
   0      91-   92!
   0      93-   94!
   0      95-   96!
   0      97-   98!
   0      99-  100!
   0        >  100!
TOTAL CHARS. PER STATEMENT =     3606
```

CHARS. PER ACTION STATEMENT

```
     BIGGEST VAL=       39 SCALED BY=        1 WIDTH=      100
   39       <     1!****************************************
   20      1-     2!********************
   17      3-     4!*****************
   28      5-     6!****************************
    5      7-     8!*****
   24      9-    10!************************
   21     11-    12!************************
    4     13-    14!****
    3     15-    16!***
    7     17-    18!*******
    7     19-    20!*******
    6     21-    22!******
    3     23-    24!***
    3     25-    26!***
    5     27-    28!*****
   10     29-    30!**********
    8     31-    32!********
    5     33-    34!*****
    0     35-    36!
    5     37-    38!*****
    3     39-    40!***
    2     41-    42!**
    3     43-    44!***
    5     45-    46!*****
    0     47-    48!
    0     49-    50!
    2     51-    52!**
    0     53-    54!
    0     55-    56!
    0     57-    58!
    0     59-    60!
    2     61-    62!**
    1     63-    64!*
    0     65-    66!
    0     67-    68!
    0     69-    70!
    0     71-    72!
    0     73-    74!
    0     75-    76!
    0     77-    78!
    0     79-    80!
    0     81-    82!
    0     83-    84!
    0     85-    86!
    0     87-    88!
    0     89-    90!
    0     91-    92!
    0     93-    94!
    0     95-    96!
    0     97-    98!
    0     99-   100!
    0       >   100!
TOTAL ACTION CHARS. =     3278
```

CHARS. PER CONDITIONAL STATEMENT

```
      BIGGEST VAL=       10 SCALED BY=        1 WIDTH=      100
    0       <     1!
    0     1-      2!
    6     3-      4!******
    2     5-      6!** .
    2     7-      8!**
   10     9-     10!**********
    4    11-     12!****
    4    13-     14!****
    0    15-     16!
    1    17-     18!*
    0    19-     20!
    0    21-     22!
    0    23-     24!
    0    25-     26!
    0    27-     28!
    1    29-     30!*
    0    31-     32!
    1    33-     34!*
    0    35-     36!
    0    37-     38!
    0    39-     40!
    0    41-     42!
    0    43-     44!
    0    45-     46!
    0    47-     48!
    0    49-     50!
    0    51-     52!
    0    53-     54!
    0    55-     56!
    0    57-     58!
    0    59-     60!
    0    61-     62!
    0    63-     64!
    0    65-     66!
    0    67-     68!
    0    69-     70!
    0    71-     72!
    0    73-     74!
    0    75-     76!
    0    77-     78!
    0    79-     80!
    0    81-     82!
    0    83-     84!
    0    85-     86!
    0    87-     88!
    0    89-     90!
    0    91-     92!
    0    93-     94!
    0    95-     96!
    0    97-     98!
    0    99-    100!
    0      >    100!
TOTAL CONDITIONAL CHARS. =      328
```

STATEMENTS PER SEQUENCE

```
    BIGGEST VAL=       37 SCALED BY=        1 WIDTH=      100
  0       <     1!
 37             1!*************************************
 31             2!********************************
 35             3!************************************
  5             4!*****
  5             5!*****
  2             6!**
  1             7!*
  0             8!
  0             9!
  0            10!
  0            11!
  0            12!
  0            13!
  0            14!
  0            15!
  0            16!
  0            17!
  0            18!
  0            19!
  0            20!
  0            21!
  0            22!
  0            23!
  0            24!
  0            25!
  0            26!
  0            27!
  0            28!
  0            29!
  0            30!
  0            31!
  0            32!
  0            33!
  0            34!
  0            35!
  0            36!
  0            37!
  0            38!
  0            39!
  0            40!
  0       >    40!
TOTAL STATEMENTS =         269
```

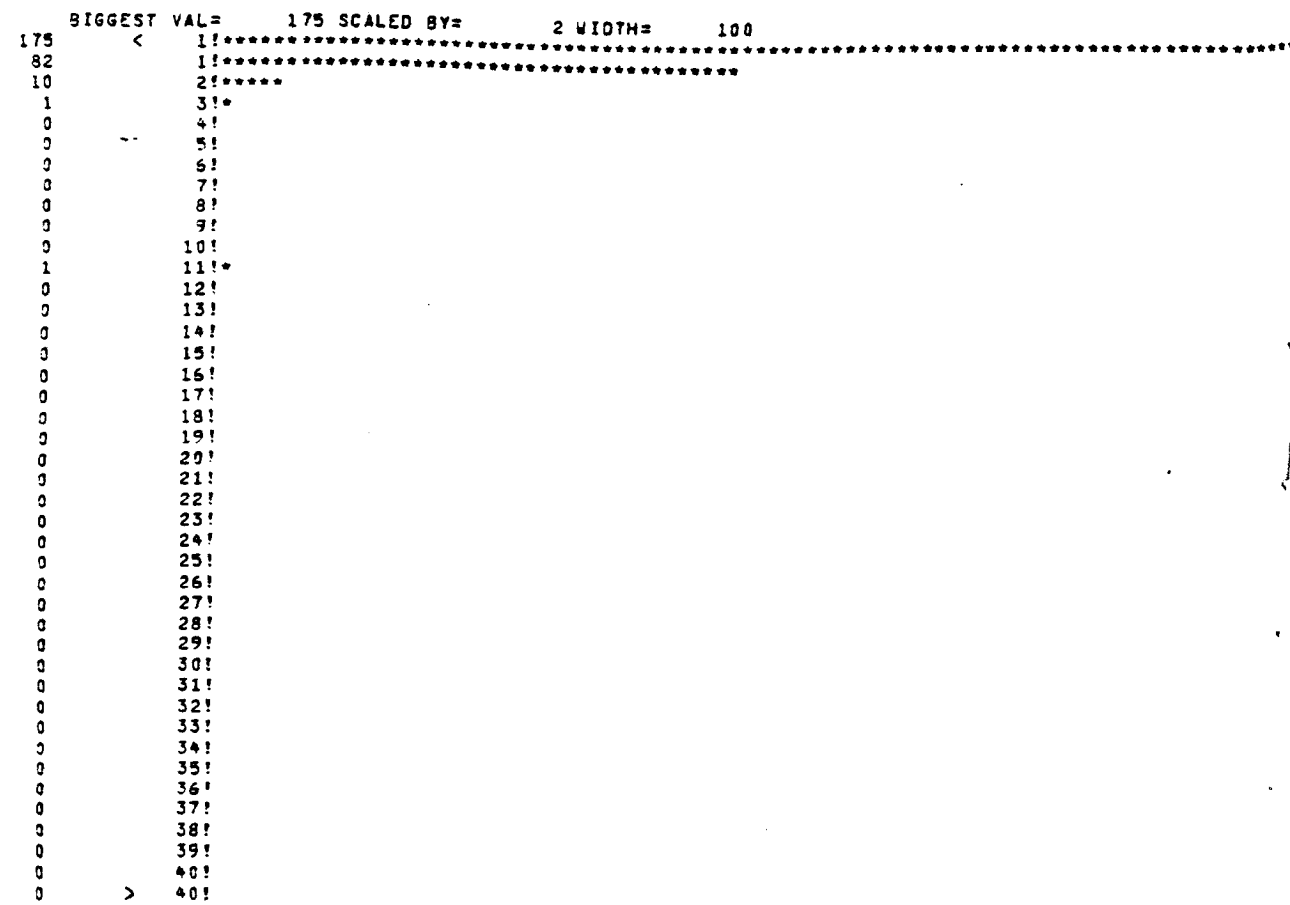
ACTION STATEMENTS PER SEQUENCE

```
      BIGGEST VAL=       56 SCALED BY=        1 WIDTH=      100
    0       <     1!
   37             1!*************************************
   56             2!******************************************************
   14             3!**************
    3             4!***
    3             5!***
    2             6!**
    1             7!*
    0             8!
    0             9!
    0            10!
    0            11!
    0            12!
    0            13!
    0            14!
    0            15!
    0            16!
    0            17!
    0            18!
    0            19!
    0            20!
    0            21!
    0            22!
    0            23!
    0            24!
    0            25!
    0            26!
    0            27!
    0            28!
    0            29!
    0            30!
    0            31!
    0            32!
    0            33!
    0            34!
    0            35!
    0            36!
    0            37!
    0            38!
    0            39!
    0            40!
    0       >    40!
TOTAL ACTION STATEMENTS =      237
```

CONDITIONAL STATEMENTS PER SEQUENCE

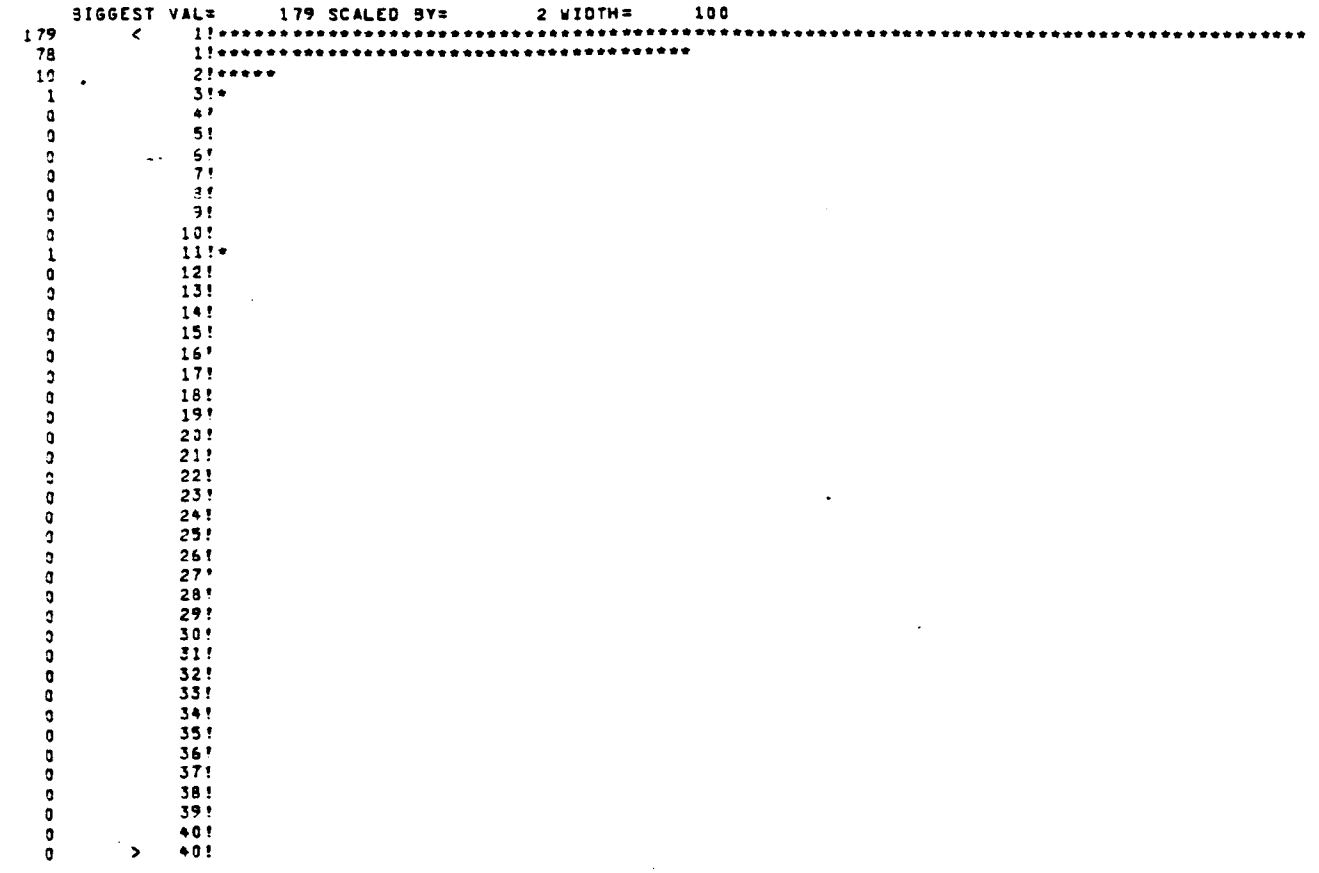
```
    BIGGEST VAL=       87 SCALED BY=        1 WIDTH=     100
 87       <     1!****************************************************************************************************
 27             1!***************************
  2             2!**
  0             3!
  0             4!
  0             5!
  0             6!
  0             7!
  0             8!
  0             9!
  0            10!
  0            11!
  0            12!
  0            13!
  0            14!
  0            15!
  0            16!
  0            17!
  0            18!
  0            19!
  0            20!
  0            21!
  0            22!
  0            23!
  0            24!
  0            25!
  0            26!
  0            27!
  0            28!
  0            29!
  0            30!
  0            31!
  0            32!
  0            33!
  0            34!
  0            35!
  0            36!
  0            37!
  0            38!
  0            39!
  0            40!
  0       >    42!
TOTAL CONDITIONAL STATEMENTS =       31
```

SEQUENCES PER REFINEMENT



```
.......UNSCALED

 BIGGEST VAL=      175 SCALED BY=        1 WIDTH=      100
      <     1!**************************************************************************************>>
            1!**************************************************************************
            2!**********
            3!*
            4!
            5!
            6!
            7!
            8!
            9!
           10!
           11!*
           12!
           13!
           14!
           15!
           16!
           17!
           18!
           19!
      .    20!
           21!
           22!
           23!
           24!
           25!
           26!
           27!
           28!
           29!
           30!
           31!
           32!
           33!
           34!
           35!
           36!
           37!
           38!
           39!
           40!
      >    40!
AL SEQUENCES =       116
```

EARTHED SEQUENCES PER REFINEMENT



,..........UNSCALED

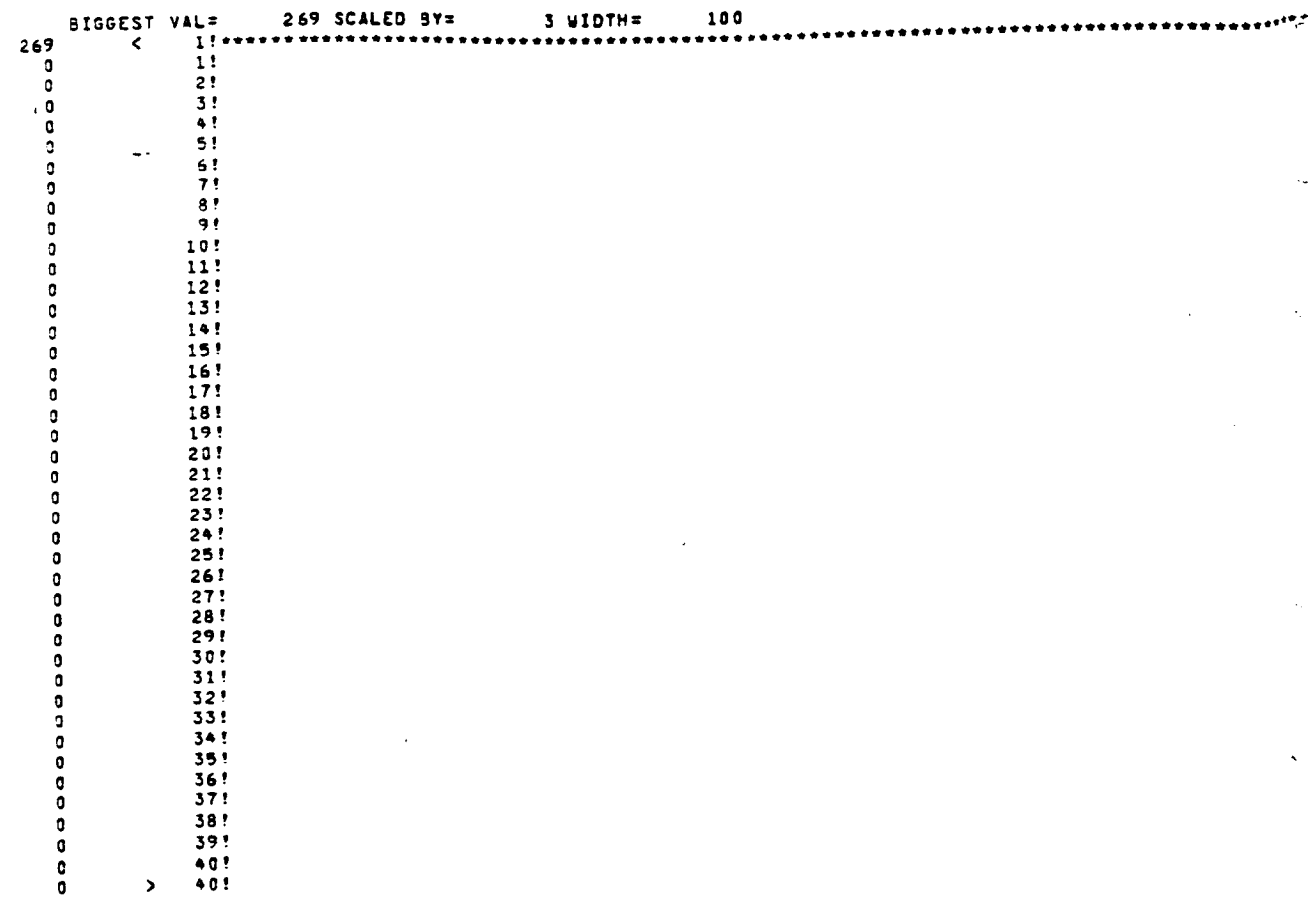
```
    BIGGEST VAL=     179 SCALED BY=       1 WIDTH=     100
179       <    1!**************************************************************************************************>>.
 79            1!***************************************************************************
 10            2!**********
  1            3!*
  0            4!
  0            5!
  0            6!
  0            7!
  0            8!
  0            9!
  0           10!
  1           11!*
  0           12!
  0           13!
  0           14!
  0           15!
  0           16!
  0           17!
  0           18!
  0           19!
  0           20!
  0           21!
  0           22!
  0           23!
  0           24!
  0           25!
  0           26!
  0           27!
  0           28!
  0           29!
  0           30!
  0           31!
  0           32!
  0           33!
  0           34!
  0           35!
  0           36!
  0           37!
  0           38!
  0           39!
  0           40!
  0       >   40!
TOTAL EARTHED SEQUENCES =      112
```

REPEATED SEQUENCES PER REFINEMENT

```
    BIGGEST VAL=      265 SCALED BY=        3 WIDTH=     100
265       <     1!*****************************************************************************************
  4             1!*
  0             2!
  0             3!
  0             4!
  0             5!
  0             6!
  0             7!
  0             8!
  0             9!
  0            10!
  0            11!
  0            12!
  0            13!
  0            14!
  0            15!
  0            16!
  0            17!
  0            18!
  0            19!
  0            20!
  0            21!
  0            22!
  0            23!
  0            24!
  0            25!
  0            26!
  0            27!
  0            28!
  0            29!
  0            30!
  0            31!
  0            32!
  0            33!
  0            34!
  0            35!
  0            36!
  0            37!
  0            38!
  0            39!
  0            40!
  0       >    40!
```

```
          UNSCALED

     BIGGEST VAL=     265 SCALED BY=       1 WIDTH=     100
265        <     1!*****************************************************************************************>>.
  4              1!****
  0              2!
  0              3!
  0              4!
  0              5!
  0              6!
  0              7!
  0              8!
  0              9!
  0             10!
  0             11!
  0             12!
  0             13!
  0             14!
  0             15!
  0             16!
  0             17!
  0             18!
  0             19!
  0             20!
  0             21!
  0             22!
  0             23!
  0             24!
  0             25!
  0             26!
  0             27!
  0             28!
  0             29!
  0             30!
  0             31!
  0             32!
  0             33!
  0             34!
  0             35!
  0             36!
  0             37!
  0             38!
  0             39!
  0             40!
  0          >  40!
TOTAL REPEATED SEQUENCES =
```

MISSED OUT SEQUENCES PER REFINEMENT



```
...UNSCALED

BEST VAL=     269 SCALED BY=       1 WIDTH=     180
    <    1!*************************************************************************************>>>
         1!
         2!
         3!
         4!
         5!
         6!
         7!
         8!
         9!
        10!
        11!
        12!
        13!
        14!
        15!
        16!
        17!
        18!
        19!
        20!
        21!
        22!
        23!
        24!
        25!
        26!
        27!
        28!
        29!
        30!
        31!
        32!
        33!
        34!
        35!
        36!
        37!
        38!
        39!
        40!
    >   40!
MISSED OUT SEQUENCES =        0
```

DEPTH OF REFINEMENT PER STATEMENT

```
     BIGGEST VAL=       80 SCALED BY=        1 WIDTH=     100
   1        <    1!*
   6             1!******
  20             2!********************
  35             3!***********************************
  54             4!******************************************************
  80             5!********************************************************************************
  54             6!******************************************************
  16             7!********************
   3             8!***
   0             9!
   0            10!
   0            11!
   0            12!
   0            13!
   0            14!
   0            15!
   0            16!
   0            17!
   0            18!
   0            19!
   0            20!
   0            21!
   0            22!
   0            23!
   0            24!
   0            25!
   0            26!
   0            27!
   0            28!
   0            29!
   0            30!
   0            31!
   0            32!
   0            33!
   0            34!
   0            35!
   0            36!
   0            37!
   0            38!
   0            39!
   0            40!
   0        >   40!
TOTAL REFINEMENTS =        94
TOTAL STATEMENTS =        269
```

DEPTH OF REFINEMENT PER TERMINAL STATEMENT

```
     BIGGEST VAL=       68 SCALED BY=        1 WIDTH=     100
   0       <     1!
   1             1!*
   8             2!********
  16             3!****************
  23             4!***********************
  68             5!**********************************************************************************
  43             6!******************************************
  13             7!*************
   3             8!***
   0             9!
   0            10!
   0            11!
   0            12!
   0            13!
   0            14!
   0            15!
   0            16!
   0            17!
   0            18!
   0            19!
   0            20!
   0            21!
   0            22!
   0            23!
   0            24!
   0            25!
   0            26!
   0            27!
   0            28!
   0            29!
   0            30!
   0            31!
   0            32!
   0            33!
   0            34!
   0            35!
   0            36!
   0            37!
   0            38!
   0            39!
   0            40!
   0       >    40!
TOTAL TERMINAL STATMTS.=     175
```

DEPTH OF REFINEMENT PER NON-TERMINAL STATEMENT

```
     BIGGEST VAL=       31 SCALED BY=        1 WIDTH=      100
   1       <     1!*
   5             1!*****
  12             2!************
  19             3!*******************
  31             4!********************************
  12             5!************
  11             6!***********
   3             7!***
   0             8!
   0             9!
   0            10!
   0            11!
   0            12!
   0            13!
   0            14!
   0            15!
   0            16!
   0            17!
   0            18!
   0            19!
   0            20!
   0            21!
   0            22!
   0            23!
   0            24!
   0            25!
   0            26!
   0            27!
   0            28!
   0            29!
   0            30!
   0            31!
   0            32!
   0            33!
   0            34!
   0            35!
   0            36!
   0            37!
   0            38!
   0            39!
   0            40!
   0       >    40!
TOTAL NON-TERMINAL STMTS.=      94
```

PATHS PER STATEMENT

```
     BIGGEST VAL=      268 SCALED BY=        3 WIDTH=      100
   0       <     1!
 268       1-   789!**********************************************************************************************
   0     790-  1578!
   0    1579-  2367!
   0    2368-  3156!
   0    3157-  3945!
   0    3946-  4734!
   0    4735-  5523!
   0    5524-  6312!
   0    6313-  7101!
   0    7102-  7890!
   0    7891-  8679!
   0    8680-  9468!
   0    9469- 10257!
   0   10258- 11046!
   0   11047- 11835!
   0   11836- 12624!
   0   12625- 13413!
   0   13414- 14202!
   0   14203- 14991!
   0   14992- 15780!
   0   15781- 16569!
   0   16570- 17358!
   0   17359- 18147!
   0   18148- 18936!
   0   18937- 19725!
   0   19726- 20514!
   0   20515- 21303!
   0   21304- 22092!
   0   22093- 22881!
   0   22882- 23670!
   0   23671- 24459!
   0   24460- 25248!
   0   25249- 26037!
   0   26038- 26826!
   0   26827- 27615!
   0   27616- 28404!
   0   28405- 29193!
   0   29194- 29982!
   0   29983- 30771!
   1       >30771!*
```

```
...........UNSCALED


     BIGGEST VAL=     268 SCALED BY=        1 WIDTH=      100
  0        <     1!
268       :-   789!****************************************************************************************************>
  0    790-  1578!
  0   1579-  2367!
  0   2368-  3156!
  0   3157-  3945!
  0   3946-  4734!
  0   4735-  5523!
  0   5524-  6312!
  0   6313-  7101!
  0   7102-  7890!
  0   7891-  8679!
  0   8680-  9468!
  0   9469-10257!
  0  10258-11046!
  0  11047-11835!
  0  11836-12624!
  0  12625-13413!
  0  13414-14202!
  0  14203-14991!
  0  14992-15780!
  0  15781-16569!
  0  16570-17358!
  0  17359-18147!
  0  18148-18936!
  0  18937-19725!
  0  19726-20514!
  0  20515-21303!
  0  21304-22092!
  0  22093-22881!
  0  22882-23670!
  0  23671-24459!
  0  24460-25248!
  0  25249-26037!
  0  26038-26826!
  0  26827-27615!
  0  27616-28404!
  0  28405-29193!
  0  29194-29982!
  0  29983-30771!
  1        >30771!*
TOTAL PATHS =   184320
```

ATHS PER SEQUENCE

```
    BIGGEST VAL=      115 SCALED BY=        2 WIDTH=      100
  0       <    1!
115     1-  789!**********************************************************
  0   790- 1578!
  0  1579- 2367!
  0  2368- 3156!
  0  3157- 3945!
  0  3946- 4734!
  0  4735- 5523!
  0  5524- 6312!
  0  6313- 7101!
  0  7102- 7890!
  0  7891- 8679!
  0  8680- 9468!
  0  9469-10257!
  0 10258-11046!
  0 11047-11835!
  0 11836-12624!
  0 12625-13413!
  0 13414-14202!
  0 14203-14991!
  0 14992-15780!
  0 15781-16569!
  0 16570-17358!
  0 17359-18147!
  0 18148-18936!
  0 18937-19725!
  0 19726-20514!
  0 20515-21303!
  0 21304-22092!
  0 22093-22881!
  0 22882-23670!
  0 23671-24459!
  0 24460-25248!
  0 25249-26037!
  0 26038-26826!
  0 26827-27615!
  0 27616-28404!
  0 28405-29193!
  0 29194-29982!
  0 29983-30771!
  1      >30771!*
```

```
..........UNSCALED

    BIGGEST VAL=     115 SCALED BY=       1 WIDTH=     100
  0       <    1!
115      1-  789!****************************************************************************************************
  0    790-  1578!
  0   1579-  2367!
  0   2368-  3156!
  0   3157-  3945!
  0   3946-  4734!
  0   4735-  5523!
  0   5524-  6312!
  0   6313-  7101!
  0   7102-  7890!
  0   7891-  8679!
  0   8680-  9468!
  0   9469-10257!
  0  10258-11046!
  0  11047-11835!
  0  11836-12624!
  0  12625-13413!
  0  13414-14202!
  0  14203-14991!
  0  14992-15780!
  0  15781-16569!
  0  16570-17358!
  0  17359-18147!
  0  18148-18936!
  0  18937-19725!
  0  19726-20514!
  0  20515-21303!
  0  21304-22092!
  0  22093-22881!
  0  22882-23670!
  0  23671-24459!
  0  24460-25248!
  0  25249-26037!
  0  26038-26826!
  0  26827-27615!
  0  27616-28404!
  0  28405-29193!
  0  29194-29982!
  0  29983-30771!
  1      >30771!*
```

CUM. CYCLOMATIC PER STATEMENT

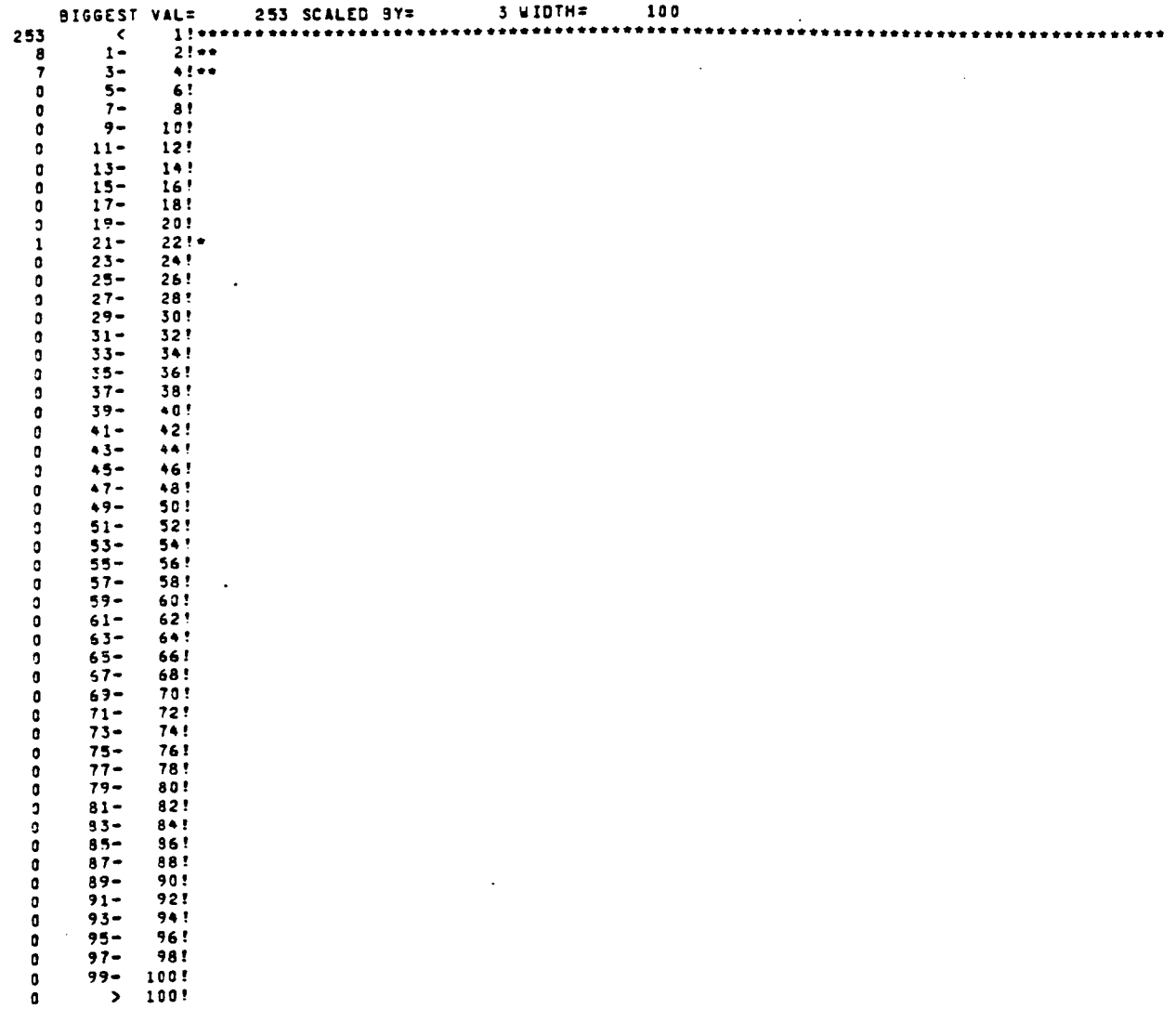
```
    BIGGEST VAL=      227 SCALED BY=        3 WIDTH=      100
227       <     1!**************************************************************************************************
  6      1-     2!**
 22      3-     4!*******
  5      5-     6!*
  3      7-     8!*
  1      9-    10!*
  0     11-    12!
  0     13-    14!
  0     15-    16!
  0     17-    18!
  0     19-    20!
  3     21-    22!*
  0     23-    24!
  0     25-    26!
  0     27-    28!
  1     29-    30!*
  0     31-    32!
  0     33-    34!
  0     35-    36!
  0     37-    38!
  0     39-    40!
  0     41-    42!
  0     43-    44!
  0     45-    46!
  0     47-    48!
  0     49-    50!
  0     51-    52!
  1     53-    54!*
  0     55-    56!
  0     57-    58!
  0     59-    60!
  0     61-    62!
  0     63-    64!
  0     65-    66!
  0     67-    68!
  0     69-    70!
  0     71-    72!
  0     73-    74!
  0     75-    76!
  0     77-    78!
  0     79-    80!
  0     81-    82!
  0     83-    84!
  0     85-    86!
  0     87-    88!
  0     89-    90!
  0     91-    92!
  0     93-    94!
  0     95-    96!
  0     97-    98!
  0     99-   100!
  0       >   100!
```

```
.........UNSCALED

   BIGGEST VAL=      227 SCALED BY=        1 WIDTH=     100
17       <     1!**************************************************************************************************>>>
 6      1-     2!******
22      3-     4!**********************
 8      5-     6!*****
 3      7-     8!***
 1      9-    10!*
 0     11-    12!
 0     13-    14!
 0     15-    16!
 0     17-    18!
 0     19-    20!
 1     21-    22!***
 0     23-    24!
 0     25-    26!
 0     27-    28!
 1     29-    30!*
 0     31-    32!
 0     33-    34!
 0     35-    36!
 0     37-    38!
 0     39-    40!
 0     41-    42!
 0     43-    44!
 0     45-    46!
 0     47-    48!
 0     49-    50!
 0     51-    52!
 1     53-    54!*
 0     55-    56!
 0     57-    58!
 0     59-    60!
 0     61-    62!
 0     63-    64!
 0     65-    66!
 0     67-    68!
 0     69-    70!
 0     71-    72!
 0     73-    74!
 0     75-    76!
 0     77-    78!
 0     79-    80!
 0     81-    82!
 0     83-    84!
 0     85-    86!
 0     87-    88!
 0     89-    90!
 0     91-    92!
 0     93-    94!
 0     95-    96!
 0     97-    98!
 0     99-   100!
 0       >   100!
:L CYCLOMATIC =          54
```

CUM. CYCLOMATIC PER SEQUENCE

```
    BIGGEST VAL=       54 SCALED BY=        1 WIDTH=      100
54        <     1!**************************************************************************************************
30       1-     2!******************************
20       3-     4!********************
 6       5-     6!******
 2       7-     8!**
 0       9-    10!
 0      11-    12!
 0      13-    14!
 0      15-    16!
 0      17-    18!
 0      19-    20!
 3      21-    22!***
 0      23-    24!
 0      25-    26!
 0      27-    28!
 0      29-    30!
 0      31-    32!
 0      33-    34!
 0      35-    36!
 0      37-    38!
 0      39-    40!
 0      41-    42!
 0      43-    44!
 0      45-    46!
 0      47-    48!
 0      49-    50!
 0      51-    52!
 1      53-    54!*
 0      55-    56!
 0      57-    58!
 0      59-    60!
 0      61-    62!
 0      63-    64!
 0      65-    66!
 0      67-    68!
 0      69-    70!
 0      71-    72!
 0      73-    74!
 0      75-    76!
 0      77-    78!
 0      79-    80!
 0      81-    82!
 0      83-    84!
 0      85-    86!
 0      87-    88!
 0      89-    90!
 0      91-    92!
 0      93-    94!
 0      95-    96!
 0      97-    98!
 0      99-   100!
 0        >   100!
```

CYCLOMATIC PER STATEMENT



**********UNSCALED

```
    BIGGEST VAL=      253 SCALED BY=        1 WIDTH=     100
253       <    1!**************************************************************************************************>>
  8      1-    2!********
  7      3-    4!*******
  0      5-    6!
  0      7-    8!
  0      9-   10!
  0     11-   12!
  0     13-   14!
  0     15-   16!
  0     17-   18!
  0     19-   20!
  1     21-   22!*
  0     23-   24!
  0     25-   26!
  0     27-   28!
  0     29-   30!
  0     31-   32!
  0     33-   34!
  0     35-   36!
  0     37-   38!
  0     39-   40!
  0     41-   42!
  0     43-   44!
  0     45-   46!
  0     47-   48!
  0     49-   50!
  0     51-   52!
  0     53-   54!
  0     55-   56!
  0     57-   58!
  0     59-   60!
  0     61-   62!
  0     63-   64!
  0     65-   66!
  0     67-   68!
  0     69-   70!
  0     71-   72!
  0     73-   74!
  0     75-   76!
  0     77-   78!
  0     79-   80!
  0     81-   82!
  0     83-   84!
  0     85-   86!
  0     87-   88!
  0     89-   90!
  0     91-   92!
  0     93-   94!
  0     95-   96!
  0     97-   98!
  0     99-  100!
  0       >  100!
```

CYCLOMATIC PER SEQUENCE

```
    BIGGEST VAL=       87 SCALED BY=        1 WIDTH=      100
87        <    1!****************************************************************************************************
29       1-    2!****************************
 0       3-    4!
 0       5-    6!
 0       7-    8!
 0       9-   10!
 0      11-   12!
 0      13-   14!
 0      15-   16!
 0      17-   18!
 0      19-   20!
 0      21-   22!
 0      23-   24!
 0      25-   26!
 0      27-   28!
 0      29-   30!
 0      31-   32!
 0      33-   34!
 0      35-   36!
 0      37-   38!
 0      39-   40!
 0      41-   42!
 0      43-   44!
 0      45-   46!
 0      47-   48!
 0      49-   50!
 0      51-   52!
 0      53-   54!
 0      55-   56!
 0      57-   58!
 0      59-   60!
 0      61-   62!
 0      63-   64!
 0      65-   66!
 0      67-   68!
 0      69-   70!
 0      71-   72!
 0      73-   74!
 0      75-   76!
 0      77-   78!
 0      79-   80!
 0      81-   82!
 0      83-   84!
 0      85-   86!
 0      87-   88!
 0      89-   90!
 0      91-   92!
 0      93-   94!
 0      95-   96!
 0      97-   98!
 0      99-  100!
 0        >  100!
```

10.2.4. Dynamic Analysis - Performance

Two different sets of performance data are given. The first shows the consumption of resources at the highest level of abstraction (PERF-B). The second is the same as the first but with the additional measurement points inside the PB2DR and PB2DI routines which are the key subroutines (PERF-A).

10.2.4.1. PERF-B

1. Note how easy it is to see which aspects of the program consume CPU time.
2. Note how in this run the recursive conversion is actually *faster* than the iterative one! Why?
3. Note how the frequency counters (FRQ) differentiate those statements executed once or many times.
4. Note from the MONITOR's self-monitoring in PERF-B-MON that the MONITOR itself is a dominant user of resources. This is because the B2D program is a toy example.

- please refer to the Dimensional Design marked 'PERF-B' in the wallet attached to the rear cover.

****
****
****
GSIN21 PERF-B-PF
****
****
SPOOLED AT 18:06 ON 02/10/81 LISTED AT 18:03 ON FRI, 02 OCT 1981

PERFORMANCE MONITOR

STMNO	CPU-TIME	IO-TIME	FREQUENCY	MAX.REC.DEP	CURR.REC.DEP
1	171	1	1	1	0
2	3	0	1	1	0
3	3	0	1	1	0
4	12	0	1	1	0
5	11	0	1	1	0
6	99	0	1	1	0
7	99	0	1	1	0
8	83	0	100	1	0
9	15	0	100	1	0
10	24	0	100	1	0
11	55	1	1	1	0
12	55	1	1	1	0

```
**********************************************************************
**********************************************************************
                                                          **********
          GSIN21   PERF-B-MON                             **********
                                                          **********
**********************************************************************
**********************************************************************
 SPOOLED AT 18:07 ON 02/10/81     LISTED AT 18:03 ON FRI, 02 OCT 1981


OK, OPEN PERF-DATA 2 1
OK, OPEN PERF-B-OUTP 3 2
OK, SEG #PERF-B
BUFFER TYPE(1=HIST,2=CIRC)
1
PLEASE INPUT CHANNEL NO FOR OUTPUT
11
PLEASE INPUT PERFORMANCE FILENAME
PERF-B-PF
PLEASE INPUT CONTROL PATHS FILENAME
PERF-B-CP
PLEASE INPUT SNAP-SHOT FILENAME
PERF-B-SS
PLEASE INPUT TRACE FILENAME
PERF-B-TR
TRACE ACTIVE?(T OR F)
F
5
7
PERFORMANCE MONITOR FOR MONITORING SUBROUTINE
TOTAL CPU-TIME=          92
TOTAL I/O-TIME=           9
FREQUENCY=          620

**** STOP
OK, CLOSE PERF-DATA
OK, CLOSE PERF-B-OUTP
OK, SLIST PERF-B-PF
PERFORMANCE MONITOR

STMNO      CPU-TIME       IO-TIME     FREQUENCY  MAX.REC.DEP CURR.REC.DEP
    1           171             1             1             1            0
    2             3             0             1             1            0
    3             3             0             1             1            0
    4            12             0             1             1            0
    5            11             0             1             1            0



    6            99             0             1             1            0
    7            99             0             1             1            0
    8            83             0           100             1            0
    9            15             0           100             1            0
   10            24             0           100             1            0
   11            55             1             1             1            0
   12            55             1             1             1            0

OK, COMO -E
```

10.2.4.2. PERF-A

1. Note the difference in frequencies between PB2DR and PB2DI. This is because PB2DR is being called recursively.

2. Note the difference in MXR between PB2DR and PB2DI. MXR is the Maximum Depth of Recursion reached during execution. Clearly PB2DI is not called recursively whereas PB2DR reaches a recursive depth of 5 showing that the greatest number of digits in any one input number is 5 (see PERF-DATA listing).

3. The large difference in CPU consumption is due to the invocation of PB2DR being monitored 498 times whereas the iterative PB2DI is only monitored 100 times. PERF-B shows the 'proper' comparison between the two algorithms. This increased MONITOR overhead is shown by its self-monitoring output. Contrast the MONITOR resource consumption and frequency of invocation between PERF-A-MON and PERF-B-MON.

- please refer to the Dimensional Design marked 'PERF-A' in the wallet attached to the rear cover.

GSIN21 PERF-A-PF

SPOOLED AT 18:07 ON 02/10/81 LISTED AT 18:03 ON FRI, 02 OCT 1981

PERFORMANCE MONITOR

STMNO	CPU-TIME	IO-TIME	FREQUENCY	MAX.REC.DEP	CURR.REC.DEP
1	330	2	1	1	0
2	3	0	1	1	0
3	3	0	1	1	0
4	13	0	1	1	0
5	13	0	1	1	0
6	260	0	1	1	0
7	259	0	1	1	0
8	242	0	100	1	0
9	137	0	100	1	0
10	59	0	100	1	0
11	54	2	1	1	0
12	54	2	1	1	0
13	111	0	498	5	0
14	19	0	100	1	0

```
**********************************************************************************
**********************************************************************************
                                                          ********************
                GSIN21  PERF-A-MON                        ********************
                                                          ********************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:07 ON 02/10/81    LISTED AT 18:03 ON FRI, 02 OCT 1981


OK, OPEN PERF-DATA 2 1
OK, OPEN PERF-A-OUTP 3 2
OK, SEG #PERF-A
BUFFER TYPE(1=HIST,2=CIRC)
1
PLEASE INPUT CHANNEL NO FOR OUTPUT
11
PLEASE INPUT PERFORMANCE FILENAME
PERF-A-PF
PLEASE INPUT CONTROL PATHS FILENAME
PERF-A-CP
PLEASE INPUT SNAP-SHOT FILENAME
PERF-A-SS
PLEASE INPUT TRACE FILENAME
PERF-A-TR
TRACE ACTIVE?(T OR F)
F
5
7
PERFORMANCE MONITOR FOR MONITORING SUBROUTINE
TOTAL CPU-TIME=          193
TOTAL I/O-TIME=            8
FREQUENCY=         1816

**** STOP
OK, CLOSE PERF-DATA
OK, CLOSE PERF-A-OUTP
OK, SLIST PERF-A-PF
PERFORMANCE MONITOR

STMNO      CPU-TIME       IO-TIME     FREQUENCY  MAX.REC.DEP CURR.REC.DEP
    1           330             2             1             1            0
    2             3             0             1             1            0
    3             3             0             1             1            0
    4            13             0             1             1            0
    5            13             0             1             1            0




    6           260             0             1             1            0
    7           259             0             1             1            0
    8           242             0           100             1            0
    9           137             0           100             1            0
   10            59             0           100             1            0
   11            54             2             1             1            0
   12            54             2             1             1            0
   13           111             0           498             5            0
   14            19             0           100             1            0

OK, COMO -E
```

```
**********************************************************************************
**********************************************************************************
                                                                    **************
                GSIN21  RUNPLOT-PERF-A-MON                          **************
                                                                    **************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:08 ON 02/10/81    LISTED AT 18:03 ON FRI, 02 OCT 1981


OK, SEG NSIN02>.DFC>.VER8>#DFC-8.2
DIMENSIONAL FLOWCHARTING PROGRAM, VERSION RECURS 8.1
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
12
WHAT DO YOU WANT TO USE INSTEAD ? -I4-
999
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
27
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
F-perf-a
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
28
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
perf-a-PF
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
32
WHAT DO YOU WANT TO USE INSTEAD ? -32A1-
perf-a-PLOT
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
N
```

```
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
Y
WHICH DO YOU WANT TO CHANGE ? (I2)
24
WHAT DO YOU WANT TO USE INSTEAD ? -L1-
T
DO YOU WANT TO SEE THE PARAMETERS? (Y OR N) -A1-
Y
```

PARAM WANTED	DEFAULT	INPUT	!	PARAM WANTED	DEFAULT	INPUT
ACTDEL	*	1	!	FRAME WIDTH	6000	14
CONDEL	?	2	!	FRAME HEIGHT	6000	15
REFBEG	!	3	!	CHAR WIDTH	6	16
ABSBEG	@	4	!	CHAR HEIGHT	4	17
PARBEG	=	5	!	SPACING	2	18
REFEND	#	6	!	DIAG	8	19
RPTEND	*	7	!	DTB	3	20
PAREND	#	8	!	DMLP	40	21
DRWDYN	+	9	!	DEVICE	PLOT	22
NOTDYN	-	10	!	TRACE LEVEL	0	23
FOREST	T	11	!	DYNAMIC INFO	T	24
STMCUT	999	12	!	GRID WANTED	F	25
FRMSIZ	100	13	!	DIVISIONS	10	26
INFILE	F-perf-a					27
DYNFIL	perf-a-PF					28
OUTFIL	FLOWIT					29
DEBUG	DEBUG					30
SNPFIL	SNAPS					31
PLTFIL	perf-a-PLOT					32
SNPSHT	F	33	!	NO OF DYNAMS	2	34

```
DO YOU WISH TO CHANGE A PARAMETER? (Y OR N) -A1-
N
O.K.
|ADVANCE FRAMES        0|-
DO YOU WANT THE HISTOGRAMS?(Y OR N -A1-)

Unit not open. Cominput. (Input from terminal.)
N
*THE TOTAL CPU TIME USED BY THIS PROGRAM =      867
THE TOTAL I/O TIME USED BY THIS PROGRAM =       27
 IN HUNDREDTHS OF A SECOND
THE NUMBER OF RECORDS READ WAS :      353
THE CHAR POSN ON THE RECORD WAS :       68
THE NUMBER OF CHARACTERS WAS    :    42593*-

**** STOP
OK, COMO -E
```

****

```
**********************************************************************************
**********************************************************************************
                                                                    ****************
                GSIN21  PERF-A-CP                                   ****************
                                                                    ****************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:08 ON 02/10/81    LISTED AT 18:03 ON FRI, 02 OCT 1981


CONTROL BUFFER EMPTY


**********************************************************************************
```

```
**********************************************************************
**********************************************************************
                                                          ************
          GSIN21  PERF-A-SS                               ************
                                                          ************
**********************************************************************
**********************************************************************
 SPOOLED AT 18:08 ON 02/10/81     LISTED AT 18:05 ON FRI, 02 OCT 1981


THERE ARE NO SNAP-SHOTS


**********************************************************************
```

```
**********************************************************************************
**********************************************************************************
                                                                      **************
                GSIN21  PERF-DATA                                     **************
                                                                      **************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:08 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981


101
-345
-12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
```

```
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345



12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
12345
9999
*

***********************************************************************
```

```
*************************************************************************
*************************************************************************
                                                             ************
                GSIN21  PERF-A-OUTP                          ************
                                                             ************
*************************************************************************
*************************************************************************
 SPOOLED AT 18:09 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981
```

```
  Original  Recursive              Iterative

      -345  -345                        -345
    -12345  -12345                    -12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
     12345  +12345                    +12345
```

```
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345


12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
12345  +12345                      +12345
 9999

*********************************************************************************
```

10.2.5. Dynamic Analysis - Debugging

1. Three snap-shots are shown.

2. The first is in the main loop in the MASTER routine. It shows the input data. Note the indication of each snap-shot's position in the circular buffer. Entry 0 is the most recent.

3. The second and third snap-shots are in LEVEL 2 routines PB2DR and PB2DI and show which digit has been extracted from the binary number. These snap-shots should be related to the input data listed in SNAP-DATA.

4. Note how the extraction of each digit in PB2DR is the mirror of PB2DI's sequence - a nice illustration of recursion versus iteration.

5. Note the difference in iteration numbers. The iteration number in each snap-shot is the 'innermost' one therefore PB2DI's iteration number relates to its internal loop. It shows that the input data had (3,4,1,2) digits in each number. The iteration number in PB2DR's snap-shot can be seen to be the currently active iteration outside of PB2DR ie the major loop in the MASTER segment. Note how the length of the input numbers can still be computed from ((1,1,1)=3, (2,2,2,2)=4, (3)=1, (4,4)=2).

6. Only iteration numbers are shown in snap-shots. In a production version of the MONITOR the recursion numbers would also be shown making this inductive concept fully general.

- please refer to the Dimensional Design marked 'SNAP' in the wallet attached to the rear cover.

```
**********************************************************************************
**********************************************************************************
                                                                        **********
                GSIN21  SNAP-SS                                         **********
                                                                        **********
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:09 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981


STATEMENT NUMBER            1
ENTRY     ITERATION AND SNAP-SHOT
   -3     (            1)    -123 is   1th of   5
   -2     (            2)    4321 is   2th of   5
   -1     (            3)       0 is   3th of   5
    0     (            4)     -32 is   4th of   5


STATEMENT NUMBER            2
ENTRY     ITERATION AND SNAP-SHOT
   -9     (            1) char 1 line    1 column    2
   -8     (            1) char 2 line    1 column    3
   -7     (            1) char 3 line    1 column    4
   -6     (            2) char 4 line    2 column    2
   -5     (            2) char 3 line    2 column    3
   -4     (            2) char 2 line    2 column    4
   -3     (            2) char 1 line    2 column    5
   -2     (            3) char 0 line    3 column    2
   -1     (            4) char 3 line    4 column    2
    0     (            4) char 2 line    4 column    3


STATEMENT NUMBER            3
ENTRY     ITERATION AND SNAP-SHOT
   -9     (            1) char 3 line    1 column   30
   -8     (            2) char 2 line    1 column   29
   -7     (            3) char 1 line    1 column   28
   -6     (            1) char 1 line    2 column   30
   -5     (            2) char 2 line    2 column   29
   -4     (            3) char 3 line    2 column   28
   -3     (            4) char 4 line    2 column   27
   -2     (            1) char 0 line    3 column   30
   -1     (            1) char 2 line    4 column   30
    0     (            2) char 3 line    4 column   29


**********************************************************************************
```

```
**********************************************************************************
**********************************************************************************
                                                                  ****************
                GSIN21  SNAP-DATA                                 ****************
                                                                  ****************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:09 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981


5
-123
+4321
0
-32
9999


**********************************************************************************
```

```
**********************************************************************
**********************************************************************
                                                          **********
                GSIN21  SNAP-OUTP                         **********
                                                          **********
**********************************************************************
**********************************************************************
 SPOOLED AT 18:09 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981


  Original  Recursive              Iterative

      -123  -123                        -123
      4321  +4321                      +4321
         0  +0                           +0
       -32  -32                         -32
      9999

**********************************************************************
```

```
**********************************************************************************
**********************************************************************************
                                                                    **************
                GSIN21   SNAP-MON                                   **************
                                                                    **************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:10 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981


OK, OPEN snap-DATA 2 1
OK, OPEN snap-OUTP 3 2
OK, SEG #snap
BUFFER TYPE(1=HIST,2=CIRC)
1
PLEASE INPUT CHANNEL NO FOR OUTPUT
11
PLEASE INPUT PERFORMANCE FILENAME
snap-PF
PLEASE INPUT CONTROL PATHS FILENAME
snap-CP
PLEASE INPUT SNAP-SHOT FILENAME
snap-SS
PLEASE INPUT TRACE FILENAME
snap-TR
TRACE ACTIVE?(T OR F)
T
PLEASE INPUT LEVEL OF DETAIL OF TRACE
30
PLEASE INPUT MAXIMUM REFINEMENT TRACED
20
PLEASE INPUT LENGTH OF STATEMENT
IN CHARACTERS REQUIRED IN TRACE
80
6
7
PERFORMANCE MONITOR FOR MONITORING SUBROUTINE
TOTAL CPU-TIME=          718
TOTAL I/O-TIME=           14
FREQUENCY=         1142

**** STOP
OK, CLOSE snap-DATA
OK, CLOSE snap-OUTP
OK, SLIST snap-SS
```

```
STATEMENT NUMBER            1
ENTRY      ITERATION AND SNAP-SHOT
   -3      (            1)    -123 is    1th of    5
   -2      (            2)    4321 is    2th of    5
   -1      (            3)       0 is    3th of    5
    0      (            4)     -32 is    4th of    5


STATEMENT NUMBER            2
ENTRY      ITERATION AND SNAP-SHOT
   -9      (            1) char 1 line     1 column     2
   -8      (            1) char 2 line     1 column     3
   -7      (            1) char 3 line     1 column     4
   -6      (            2) char 4 line     2 column     2
   -5      (            2) char 3 line     2 column     3
   -4      (            2) char 2 line     2 column     4
   -3      (            2) char 1 line     2 column     5
   -2      (            3) char 0 line     3 column     2
   -1      (            4) char 3 line     4 column     2
    0      (            4) char 2 line     4 column     3


STATEMENT NUMBER            3
ENTRY      ITERATION AND SNAP-SHOT
   -9      (            1) char 3 line     1 column    30
   -8      (            2) char 2 line     1 column    29
   -7      (            3) char 1 line     1 column    28
   -6      (            1) char 1 line     2 column    30
   -5      (            2) char 2 line     2 column    29
   -4      (            3) char 3 line     2 column    28
   -3      (            4) char 4 line     2 column    27
   -2      (            1) char 0 line     3 column    30
   -1      (            1) char 2 line     4 column    30
    0      (            2) char 3 line     4 column    29

OK, COMO -E
```

****

10.2.6. Dynamic Analysis - Animation

Two approaches to animation are given.

1. A comprehensive animation of the program's execution given itself as a Dimensional Design.
2. A simple, cheap way of relating the path of control flow to the static Dimensional Design.

10.2.6.1. Trace

1. Note how the Trace shows the (psuedo) parallel execution of the recursive and iterative algorithms.
2. Note how the different shapes show the different nature of the algorithms.
3. The B2DR trace shows clearly how recursive algorithms *descend* and then *ascend*.
4. The B2DI trace shows clearly the iterative nature of the algorithm. Contrast the depths of refinement reached by B2DR and B2DI and the iteration numbers of the loops.
5. Note how the resolution of IF statements is clearly animated.
6. Note how depth pruning has been used to cut down the size of the output.
7. Due to the experimental nature of the MONITOR many superfluous REF DEPTH statements are generated. A production version would not suffer from such wasteful inelegance.

- please refer to the Dimensional Design marked 'TRAC' in the wallet attached to the rear cover.

```
**********************************************************************************
**********************************************************************************
                                                                      **************
                GSIN21   TRAC-MON                                     **************
                                                                      **************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:10 ON 02/10/81    LISTED AT 18:06 ON FRI, 02 OCT 1981


OK, OPEN trac-DATA 2 1
OK, OPEN trac-OUTP 3 2
OK, SEG #trac
BUFFER TYPE(1=HIST,2=CIRC)
1
PLEASE INPUT CHANNEL NO FOR OUTPUT
11
PLEASE INPUT PERFORMANCE FILENAME
trac-PF
PLEASE INPUT CONTROL PATHS FILENAME
trac-CP
PLEASE INPUT SNAP-SHOT FILENAME
trac-SS
PLEASE INPUT TRACE FILENAME
trac-TR
TRACE ACTIVE?(T OR F)
T
PLEASE INPUT LEVEL OF DETAIL OF TRACE
30
PLEASE INPUT MAXIMUM REFINEMENT TRACED
40
PLEASE INPUT LENGTH OF STATEMENT
IN CHARACTERS REQUIRED IN TRACE
80
6
7
PERFORMANCE MONITOR FOR MONITORING SUBROUTINE
TOTAL CPU-TIME=          367
TOTAL I/O-TIME=           14
FREQUENCY=          592

**** STOP
OK, CLOSE trac-DATA
OK, CLOSE trac-OUTP
OK, SLIST trac-SS
```

```
-ATEMENT NUMBER            1
;TRY      ITERATION AND SNAP-SHOT
   0      (            1)    -123 is    1th of    2


-ATEMENT NUMBER            2
JTRY      ITERATION AND SNAP-SHOT
  -2      (            1) char 1 line     1 column     2
  -1      (            1) char 2 line     1 column     3
   0      (            1) char 3 line     1 column     4


ATEMENT NUMBER            3
JTRY      ITERATION AND SNAP-SHOT
  -2      (            1) char 3 line     1 column    30
  -1      (            2) char 2 line     1 column    29
   0      (            3) char 1 line     1 column    28

<, COMO -E


**********************************************************************************
```

```
**********************************************************************
**********************************************************************
                                                              *********
                GSIN21  TRAC-DATA                             *********
                                                              *********
**********************************************************************
**********************************************************************
 SPOOLED AT 18:10 ON 02/10/81    LISTED AT 18:05 ON FRI, 02 OCT 1981


2
-123
9999


**********************************************************************
```

```
*****************************************************************************
*****************************************************************************
                                                      ************************
              GSIN21  TRAC-SORC                       ************************
                                                      ************************
*****************************************************************************
*****************************************************************************
SPOOLED AT 18:20 ON 02/10/81     LISTED AT 18:16 ON FRI, 02 OCT 1981


      .PROG
      .MONITOR SNAPS,PERFORMANCE,CONTROL,HISTORY
      .TRACE
         .T1: DEP(40,40), DET( 3),'RF(1) .ET
         .T2: DEP( 0, 0), DET(10)        .ET
         .T3: DEP( 2, 2), DET( 3)        .ET
      .ENDTRACE
      .SNAP-SHOT
         .SS1: DET(20), FORMAT(200), SIZE(80) .ESS
         .SS2: DET(25), FORMAT(210), SIZE(80) .ESS
         .SS3: DET(25), FORMAT(220), SIZE(80) .ESS
      .ENDSNAP
      .FILTERS
         .BF1: ( (IT(N).LE.5).OR.(MOD(IT(N),20).EQ.0) ) .EBF
      .ENDFILTERS
      .ENDMONITOR
      .MASTER
      .C Macros
          define(maxdata,101)
          define(widmax,30)
          define(lenmax,101)
          define(terminator,9999)
          define(stleftcol,1)
          define(strightcol,widmax/2+1)
          define(err,1)
          define(monitordecs,
                 INTEGER*4 IT
                 INTEGER*4 ISTRNG(80)
                )
          .EC
      .C Declarations
          INTEGER IN,EX,NUM,J,K
          INTEGER BIN(maxdata)
          INTEGER PAGE(widmax,lenmax)
          monitordecs
          .EC



      .C I/O Formats
100       FORMAT(I10)
110       FORMAT(I10,2X,widmax A1)
120       FORMAT(2X,'Original',2X,'Recursive',12X,'Iterative'//)
200       FORMAT(I8,' is ',I3,'th of ',I3)
          .EC
      .T1: .BEGIN
      .T3: .C Initialise
          .C I/O channels
               IN = 1
               READ(IN,100) IN,EX
              .EC
          .C Clear output PAGE buffer to spaces
              .FOR J=1,widmax .DO
                  .FOR K=1,lenmax .DO
                      PAGE(J,K) = ' '
                  .ENDFR
              .ENDFR
              .EC
       .EC
      .T1: .C Read input data into array BIN
          READ(IN,100)NUM,(BIN(J),J=1,NUM)
          .EC
      .T1: .C Convert input set of bin nums to dec in PAGE buffer
          J = 1
          .T1: .CYCLE K=1,lenmax .TILL(1) .DO
              .UNTIL(end of input reached).IE
                  .EXITIF(BIN(J).EQ.terminator).TOSITU(1)
              .SS1: BIN(J),J,NUM
              .T1: .C Convert Jth bin to dec 2 ways,one per col
                       .T1: .CALL(1) B2DR(BIN(J),PAGE,J,stleftcol)
                  .PARSEP
                       .T1: .CALL(1) B2DI(BIN(J),PAGE,J,widmax)
                  .EC
              J = J+1
          .REPEAT
              .SITU(1)
                  .ASSUMPTION Input terminator found
                       .T1: .OK
              .LIMIT
                  .T1: .FAIL(EX,'Terminator not found & PAGE full')
          .ENDCY
          .EC
      .T1: .C Output contents of PAGE buffer
          WRITE(EX,120)
          WRITE(EX,110)(BIN(K),(PAGE(J,K),J=1,widmax),K=1,NUM)
          .EC
```

```
      .T1: .STOP
      .ENDM
      .LEVEL 1
      .SUBROUTINE B2DR(BINARY,PAGE,LINE,START)
      .C Declarations
          .C Parameters
              INTEGER BINARY,PAGE(widmax,lenmax),LINE,START
              .EC
          .C Locals
              INTEGER NCP,BIN
              .EC
          .EC
      .T1: .BEGIN
      .C Initialise output line position
         NCP = START
         .EC
      .T1: .C Output sign of BINARY
          .IF(BINARY.LT.0).THEN
              .CALL(3) PRINT('-',PAGE,LINE,NCP)
          .ELSE
              .CALL(3) PRINT('+',PAGE,LINE,NCP)
          .ENDIF
          NCP = NCP+1
          .EC
      .T1: .C Output magnitude in decimal chars from most sig digit to least
          BIN = IABS(BINARY)
          .T1: .CALL(2) PB2DR(BIN,PAGE,LINE,NCP)
          .EC
      .RETURN
      .END
      .SETSEP
      .SUBROUTINE B2DI(BINARY,PAGE,LINE,START)
      .C Declarations
          .C Parameters
              INTEGER BINARY,PAGE(widmax,lenmax),LINE,START
              .EC
          .C Locals
              INTEGER NCP,BIN,I,REM
              .EC
          .EC
      .T1: .BEGIN
      .C Initialise output line position
          NCP = START
         .EC
      .T1: .C Output magnitude in decimal chars from least sig digit to most
          BIN = IABS(BINARY)
          .T1: .CALL(2) PB2DI(BIN,PAGE,LINE,NCP)



          .EC
      .T1: .C Output sign of BINARY
          .IF(BINARY.LT.0).THEN
              .CALL(3) PRINT('-',PAGE,LINE,NCP)
          .ELSE
              .CALL(3) PRINT('+',PAGE,LINE,NCP)
          .ENDIF
          NCP = NCP-1
          .EC
      .RETURN
      .END
      .ENDLEVEL
      .LEVEL 2
      .SUBROUTINE PB2DR(BINARY,PAGE,LINE,NCP)
      .C Declarations
          .C Parameters
              INTEGER BINARY,PAGE(widmax,lenmax),LINE,NCP
              .EC
          .C Locals
              INTEGER LSTDIG,BIN
              monitordecs
              INTEGER CHR
              .EC
          .C Functions
              INTEGER CHAR
              .EC
          .EC
      .T1: .BEGIN
      .T1: .C Convert all but least sig digit
          .T1: .IF(BINARY.GE.10).THEN
              .T1: BIN = BINARY/10
              .T1: .CALL(*) PB2DR(BIN,PAGE,LINE,NCP)
          .ELSE
              .T1: .NULL
          .ENDIF
          .EC
      .T1: .C Detatch least sig digit
          LSTDIG = MOD(BINARY,10)
          .EC
      .T1: .C Print least sig digit
          CHR = CHAR(LSTDIG)
          .SS2: CHR,LINE,NCP
210       FORMAT(' char ',A1,' line ',I4,' column ',I4)
          .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
          NCP = NCP+1
          .EC
      .RETURN
```

```
        .END
        .SETSEP
        .SUBROUTINE PB2DI(BINARY,PAGE,LINE,NCP)
        .C Declarations
            .C Parameters
                INTEGER BINARY,PAGE(widmax,lenmax),LINE,NCP
                .EC
            .C Locals
                INTEGER LSTDIG,J,BIN
                monitordecs
                INTEGER CHR
                .EC
            .C Functions
                INTEGER CHAR
                .EC
            .EC
        .T1: .BEGIN
        BIN = BINARY
        .T1: .CYCLE J=1,widmax .TILL(1) .DO
            .T1: .C Detatch Jth least sig digit
                LSTDIG = MOD(BIN,10)
                .T1: BIN = BIN/10
                .EC
            .T1: .C Print Jth least sig digit
                CHR = CHAR(LSTDIG)
                .SS2: CHR,LINE,NCP
210             FORMAT(' char ',A1,' line ',I4,' column ',I4)
                .CALL(3) PRINT(CHR,PAGE,LINE,NCP)
                NCP = NCP-1
                .EC
            .T1: .UNTIL(only leading zeros left).IE
                .EXITIF(BIN.EQ.0).TOSITU(1)
        .REPEAT
            .SITU(1)
                .OK
            .LIMIT
                .FAIL(err,'PB2DI LOOP GUARD ERROR')
        .ENDCY
        .RETURN
        .END
        .ENDLEVEL
        .LEVEL 3
        .SUBROUTINE PRINT(CHCODE,PAGE,LINE,WIDPOS)
        .C Declarations
            .C Parameters
                INTEGER CHCODE,PAGE(widmax,lenmax),LINE,WIDPOS
                .EC



            .C Locals
                LOGICAL WIDOK,LENOK
                .EC
            .EC
        .BEGIN
        .C Check valid position within PAGE
            WIDOK = (1.LE.WIDPOS).AND.(WIDPOS.LE. widmax )
            LENOK = (1.LE.LINE  ).AND.(LINE  .LE. lenmax )
            .IF (WIDOK.AND.LENOK) .THEN
               .OK
            .ELSE
                .FAIL(err,'Page area violation')
            .ENDIF
            .EC
        PAGE(WIDPOS,LINE) = CHCODE
        .RETURN
        .END
        .SETSEP
        .INTEGER FUNCTION CHAR(INT)
        .C Declarations
            .C Parameters
                INTEGER INT
                .EC
            .EC
        .BEGIN
        .IF(INT.EQ.0).THEN
            CHAR = '0'
        .ELIF(INT.EQ.1).THEN
            CHAR = '1'
        .ELIF(INT.EQ.2).THEN
            CHAR = '2'
        .ELIF(INT.EQ.3).THEN
            CHAR = '3'
        .ELIF(INT.EQ.4).THEN
            CHAR = '4'
        .ELIF(INT.EQ.5).THEN
            CHAR = '5'
        .ELIF(INT.EQ.6).THEN
            CHAR = '6'
        .ELIF(INT.EQ.7).THEN
            CHAR = '7'
        .ELIF(INT.EQ.8).THEN
            CHAR = '8'
        .ELIF(INT.EQ.9).THEN
            CHAR = '9'
        .ELSE
            .FAIL(err,'FUNC CHAR PARAM NOT DIGIT')
```

```
.ENDIF
.RETURN
.END
.SETSEP
.INTEGER FUNCTION DIGITS(POSINT)
.C Declarations
    .C Parameters
        INTEGER POSINT
        .EC
    .C Functions
        INTEGER DIGTSR
        .EC
    .EC
.BEGIN
.IF(POSINT.GE.10).THEN
    DIGITS = 1+DIGTSR(POSINT/10)
.ELSE
    DIGITS = 1
.ENDIF
.RETURN
.END
.ENDLEVEL
.LEVEL 4
.INTEGER FUNCTION DIGTSR(INT)
INTEGER INT.DIGITS
.BEGIN
DIGTSR = DIGITS(INT)
.RETURN
.END
.ENDLEVEL
.ENDP
```

****

10.2.6.2. Control Path

1. The static Dimensional Design is augmented with 'CL n' statements in which 'n' corresponds to a STATEMENT NUMBER in TRAC-CP.

2. TRAC-CP gives a conventional circular buffer of statement numbers executed. Note how much more difficult it is to visualise the execution of the program from this control path scheme than it is from the trace animation Dimensional Design.

3. TRAC-CP includes the iteration number information unlike most conventional implementations. Note how useful this is in visualising the execution of the program. This idea would be generalised to include a depth of recursion indicator in a production quality ROOTS system.

4. The 'PF n' part of the Control statement label indicates that performance data was collected for the points marked PF as well as the Control Path data. This is given in TRAC-PF.

- please refer to the Dimensional Design marked 'CP' in the wallet attached to the rear cover.

```
**********************************************************************************
**********************************************************************************
                                                                        **********
                GSIN21  TRAC-CP                                         **********
                                                                        **********
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:20 ON 02/10/81    LISTED AT 18:16 ON FRI. 02 OCT 1981
```

CIRCULAR CONTROL BUFFER

EXEC. ORDER	STMT. NO.	ITERATION
-80	1	1
-79	2	1
-78	3	1
-77	4	1
-76	5	1
-75	6	1
-74	7	1
-73	8	1
-72	9	1
-71	10	1
-70	11	1
-69	12	1
-68	20	1
-67	21	1
-66	22	1
-65	23	1
-64	24	1
-63	25	1
-62	32	1
-61	33	1
-60	34	1
-59	35	1
-58	36	1
-57	37	1
-56	38	1
-55	32	1
-54	33	1
-53	34	1
-52	35	1
-51	36	1
-50	37	1
-49	38	1
-48	32	1

-47	33	1
-46	34	1
-45	35	1
-44	39	1
-43	40	1
-42	41	1
-41	42	1
-40	43	1
-39	40	1
-38	41	1
-37	42	1
-36	43	1
-35	40	1
-34	41	1
-33	42	1
-32	43	1
-31	13	1
-30	14	1
-29	26	1
-28	27	1
-27	28	1
-26	29	1
-25	44	1
-24	45	1
-23	46	1
-22	47	1
-21	48	1
-20	49	1
-19	50	1
-18	51	1
-17	47	2
-16	48	2
-15	49	2
-14	50	2
-13	51	2
-12	47	3
-11	48	3
-10	49	3
-9	50	3
-8	51	3
-7	52	1
-6	53	1
-5	30	1
-4	31	1
-3	15	1
-2	16	1
-1	18	1
0	19	1

GSIN21 TRAC-PF

SPOOLED AT 18:20 ON 02/10/81 LISTED AT 18:17 ON FRI. 02 OCT 1981

PERFORMANCE MONITOR

STMNO	CPU-TIME	IO-TIME	FREQUENCY	MAX.REC.DEP	CURR.REC.DEP
1	274	5	1	1	0
2	2	0	1	1	0
3	1	0	1	1	0
4	1	0	1	1	0
5	1	0	1	1	0
6	264	5	1	1	0
7	262	5	1	1	0
8	262	5	1	1	0
9	259	5	1	1	0
10	255	2	1	1	0
11	105	0	1	1	0
12	105	0	1	1	0
13	149	2	1	1	0
14	148	2	1	1	0
15	2	0	1	1	0
16	1	0	1	1	0
17	0	0	0	0	0
18	3	0	1	1	0
19	2	0	1	1	0
20	104	0	1	1	0
21	4	0	1	1	0
22	3	0	1	1	0
23	97	0	1	1	0
24	96	0	1	1	0
25	95	0	1	1	0
26	95	1	1	1	0
27	89	1	1	1	0
28	88	1	1	1	0
29	88	1	1	1	0
30	4	0	1	1	0
31	2	0	1	1	0
32	93	0	3	3	0
33	83	0	3	3	0
34	82	0	3	3	0
35	82	0	3	3	0
36	81	0	2	2	0
37	1	0	2	1	0
38	79	0	2	2	0
39	1	0	1	1	0
40	3	0	3	1	0
41	0	0	3	1	0
42	18	0	3	1	0
43	14	0	3	1	0
44	87	1	1	1	0
45	85	1	1	1	0
46	82	1	1	1	0
47	6	0	3	1	0
48	3	0	3	1	0
49	1	0	3	1	0
50	68	1	3	1	0
51	65	1	3	1	0
52	2	0	1	1	0
53	0	0	1	1	0

10.2.7. ROOTS Source Code

```
**********************************************************************************
**********************************************************************************
                                                          **********************
                GSIN21  DESN                              **********************
                                                          **********************
**********************************************************************************
**********************************************************************************
 SPOOLED AT 18:21 ON 02/10/81    LISTED AT 18:17 ON FRI, 02 OCT 1981


      .PROG
      .MASTER
      .C Macros
          define(maxdata,101)
          define(widmax,30)
          define(lenmax,101)
          define(terminator,9999)
          define(stleftcol,1)
          define(strightcol,widmax/2+1)
          define(err,1)
          .EC
      .C Declarations
          INTEGER IN,EX,NUM,J,K
          INTEGER BIN(maxdata)
          INTEGER PAGE(widmax,lenmax)
          .EC
      .C I/O Formats
100       FORMAT(I10)
110       FORMAT(I10,2X,widmax A1)
120       FORMAT(2X,'Original',2X,'Recursive',12X,'Iterative'//)
          .EC
      .BEGIN
      .C Initialise
          .C I/O channels
              IN = 1
              READ(IN,100) IN,EX
              .EC
          .C Clear output PAGE buffer to spaces
              .FOR J=1,widmax .DO
                  .FOR K=1,lenmax .DO
                      PAGE(J,K) = ' '
                  .ENDFR
              .ENDFR
              .EC
          .EC
      .C Read input data into array BIN



          READ(IN,100)NUM,(BIN(J),J=1,NUM)
          .EC
      .C Convert input set of bin nums to dec in PAGE buffer
          J = 1
          .CYCLE K=1,lenmax .TILL(1) .DO
              .UNTIL(end of input reached).IE
                  .EXITIF(BIN(J).EQ.terminator).TOSITU(1)
              .C Convert Jth bin to dec 2 ways,one per col
                      .CALL(1) B2DR(BIN(J),PAGE,J,stleftcol)
                  .PARSEP
                      .CALL(1) B2DI(BIN(J),PAGE,J,widmax)
                  .EC
              J = J+1
          .REPEAT
              .SITU(1)
                  .ASSUMPTION Input terminator found
                      .OK
              .LIMIT
                  .FAIL(EX,'Terminator not found & PAGE full')
          .ENDCY
          .EC
      .C Output contents of PAGE buffer
          WRITE(EX,120)
          WRITE(EX,110)(BIN(K),(PAGE(J,K),J=1,widmax),K=1,NUM)
          .EC
      .STOP
      .ENDM
      .LEVEL 1
      .SUBROUTINE B2DR(BINARY,PAGE,LINE,START)
      .C Declarations
          .C Parameters
              INTEGER BINARY,PAGE(widmax,lenmax),LINE,START
              .EC
          .C Locals
              INTEGER NCP,BIN
              .EC
          .EC
      .BEGIN
      .C Initialise output line position
          NCP = START
          .EC
      .C Output sign of BINARY
          .IF(BINARY.LT.0).THEN
              .CALL(3) PRINT('-',PAGE,LINE,NCP)
          .ELSE
              .CALL(3) PRINT('+',PAGE,LINE,NCP)
          .ENDIF
```

```
    NCP = NCP+1
    .EC
.C Output magnitude in decimal chars from most sig digit to least
    BIN = IABS(BINARY)
    .CALL(2) PB2DR(BIN,PAGE,LINE,NCP)
    .EC
.RETURN
.END
.SETSEP
.SUBROUTINE B2DI(BINARY,PAGE,LINE,START)
.C Declarations
    .C Parameters
        INTEGER BINARY,PAGE(widmax,lenmax),LINE,START
        .EC
    .C Locals
        INTEGER NCP,BIN,I,REM
        .EC
    .EC
.BEGIN
.C Initialise output line position
    NCP = START
    .EC
.C Output magnitude in decimal chars from least sig digit to most
    BIN = IABS(BINARY)
    .CALL(2) PB2DI(BIN,PAGE,LINE,NCP)
    .EC
.C Output sign of BINARY
    .IF(BINARY.LT.0).THEN
        .CALL(3) PRINT('-',PAGE,LINE,NCP)
    .ELSE
        .CALL(3) PRINT('+',PAGE,LINE,NCP)
    .ENDIF
    NCP = NCP-1
    .EC
.RETURN
.END
.ENDLEVEL
.LEVEL 2
.SUBROUTINE PB2DR(BINARY,PAGE,LINE,NCP)
.C Declarations
    .C Parameters
        INTEGER BINARY,PAGE(widmax,lenmax),LINE,NCP
        .EC
    .C Locals
        INTEGER LSTDIG,BIN
        .EC
    .C Functions



        INTEGER CHAR
        .EC
    .EC
.BEGIN
.C Convert all but least sig digit
    .IF(BINARY.GE.10).THEN
        BIN = BINARY/10
        .CALL(*) PB2DR(BIN,PAGE,LINE,NCP)
    .ELSE
        .NULL
    .ENDIF
    .EC
.C Detatch least sig digit
    LSTDIG = MOD(BINARY,10)
    .EC
.C Print least sig digit
    .CALL(3) PRINT(CHAR(LSTDIG),PAGE,LINE,NCP)
    NCP = NCP+1
    .EC
.RETURN
.END
.SETSEP
.SUBROUTINE PB2DI(BINARY,PAGE,LINE,NCP)
.C Declarations
    .C Parameters
        INTEGER BINARY,PAGE(widmax,lenmax),LINE,NCP
        .EC
    .C Locals
        INTEGER LSTDIG,J,BIN
        .EC
    .C Functions
        INTEGER CHAR
        .EC
    .EC
.BEGIN
BIN = BINARY
.CYCLE J=1,widmax .TILL(1) .DO
    .C Detatch Jth least sig digit
        LSTDIG = MOD(BIN,10)
        BIN = BIN/10
        .EC
    .C Print Jth least sig digit
        .CALL(3) PRINT(CHAR(LSTDIG),PAGE,LINE,NCP)
        NCP = NCP-1
        .EC
    .UNTIL(only leading zeros left).IE
        .EXITIF(BIN.EQ.0).TOSITU(1)
```

```
.REPEAT
    .SITU(1)
        .OK
    .LIMIT
        .FAIL(err,'PB2DI LOOP GUARD ERROR')
.ENDCY
.RETURN
.END
.ENDLEVEL
.LEVEL 3
.SUBROUTINE PRINT(CHCODE,PAGE,LINE,WIDPOS)
.C Declarations
    .C Parameters
        INTEGER CHCODE,PAGE(widmax,lenmax),LINE,WIDPOS
        .EC
    .C Locals
        LOGICAL WIDOK,LENOK
        .EC
    .EC
.BEGIN
.C Check valid position within PAGE
    WIDOK = (1.LE.WIDPOS).AND.(WIDPOS.LE. widmax )
    LENOK = (1.LE.LINE  ).AND.(LINE  .LE. lenmax )
    .IF (WIDOK.AND.LENOK) .THEN
       .OK
    .ELSE
        .FAIL(err,'Page area violation')
    .ENDIF
    .EC
PAGE(WIDPOS,LINE) = CHCODE
.RETURN
.END
.SETSEP
.INTEGER FUNCTION CHAR(INT)
.C Declarations
    .C Parameters
        INTEGER INT
        .EC
    .EC
.BEGIN
.IF(INT.EQ.0).THEN
    CHAR = '0'
.ELIF(INT.EQ.1).THEN
    CHAR = '1'
.ELIF(INT.EQ.2).THEN
    CHAR = '2'
.ELIF(INT.EQ.3).THEN

    CHAR = '3'
.ELIF(INT.EQ.4).THEN
    CHAR = '4'
.ELIF(INT.EQ.5).THEN
    CHAR = '5'
.ELIF(INT.EQ.6).THEN
    CHAR = '6'
.ELIF(INT.EQ.7).THEN
    CHAR = '7'
.ELIF(INT.EQ.8).THEN
    CHAR = '8'
.ELIF(INT.EQ.9).THEN
    CHAR = '9'
.ELSE
    .FAIL(err,'FUNC CHAR PARAM NOT DIGIT')
.ENDIF
.RETURN
.END
.SETSEP
.INTEGER FUNCTION DIGITS(POSINT)
.C Declarations
    .C Parameters
        INTEGER POSINT
        .EC
    .C Functions
        INTEGER DIGTSR
        .EC
    .EC
.BEGIN
.IF(POSINT.GE.10).THEN
    DIGITS = 1+DIGTSR(POSINT/10)
.ELSE
    DIGITS = 1
.ENDIF
.RETURN
.END
.ENDLEVEL
.LEVEL 4
.INTEGER FUNCTION DIGTSR(INT)
INTEGER INT,DIGITS
.BEGIN
DIGTSR = DIGITS(INT)
.RETURN
.END
.ENDLEVEL
.ENDP
```

10.2.8. Execution

```
**********************************************************************
**********************************************************************
                                                          **********
                GSIN21  DESN-DATA                         **********
                                                          **********
**********************************************************************
**********************************************************************
 SPOOLED AT 18:21 ON 02/10/81    LISTED AT 18:17 ON FRI, 02 OCT 1981


5
-123
+4321
0
-32
9999
```

```
**********************************************************************
**********************************************************************
                                                          **********
                GSIN21  DESN-OUTP                         **********
                                                          **********
**********************************************************************
**********************************************************************
 SPOOLED AT 18:21 ON 02/10/81    LISTED AT 18:17 ON FRI, 02 OCT 1981


  Original  Recursive              Iterative

      -123  -123                      -123
      4321  +4321                    +4321
         0  +0                         +0
       -32  -32                       -32
      9999
```

```
**********************************************************************
**********************************************************************
                                                          **********
                GSIN21  DESN-MON                          **********
                                                          **********
**********************************************************************
**********************************************************************
 SPOOLED AT 18:21 ON 02/10/81    LISTED AT 18:17 ON FRI, 02 OCT 1981


OK, OPEN DESN-DATA 2 1
OK, OPEN DESN-OUTP 3 2
OK, SEG #DESN
6
7

**** STOP
OK, CLOSE DESN-DATA
OK, CLOSE DESN-OUTP
OK, COMO -E
```

```
**********************************************************************
```

CHAPTER 11. REFERENCES

11.1 References, Sorted by Author

OUTLINE

- none

11. REFERENCES

11.1. References, Sorted by Author

1. A.M. Addyman and R. Lane, "Software Production Using FLOCODER," *Conf. on Soft. Eng. for Telecomms. Switching Systems*, pp.253-261 (2-5 April 1973).
2. R.H. Anderson, "Programming on a Tablet: a Proposal for a New Notation," *SIGPLAN Notices* 7(10) (Oct. 1972).
3. T. Anderson and R.W. Witty, "Safe Programming," *BIT* 18, pp.1-8 (1978).
4. J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Comm.ACM* 21(8), pp.613-641 (Aug. 1978).
5. L. Belady and M. Lehman, "Large Systems," in *Research Directions in Software Technology*, ed. P. Wegner, MIT Press (1979).
6. M. Bertran and J. Xampeny, "A Computerised Power Network Telecontrol Center: Environment and Solution Framework," *Power Engineering Society, Summer Meeting* (July 1979).
7. B.W. Boehm, "Software and its impact: a quantitative assessment," *Datamation* 19(9) (May 1973).
8. C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Comm.ACM* 9(5), pp.366-371 (May 1966).
9. F.P. Brooks, *Mythical Man Month*, Addison-Wesley (1975).
10. R.E. Brown, "Towards a Better Language for Structured Programming," *SIGPLAN Notices*, p.41 (July 1976).
11. L. Carroll, *Alice's Adventures in Wonderland*, 1865.
12. N. Chapin and S. Denniston, "Characteristics of a Structured Program," *SIGPLAN Notices* 13(5) (May 1978).
13. M.H. Clifton, "A Technique for Making Structured Programs More Readable," *SIGPLAN Notices* 13(4), pp.58-63 (April 1978).
14. D. Coleman, "The Systematic Design of File Processing Programs," *Software - Practice & Experience* 7, pp.371-381 (1977).
15. O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, London (1972).
16. J.B. Dennis, "First Version of a Data Flow Programming Language," in *Lecture Notes in Computer Science vol 19*, Springer Verlag (1974).
17. E.W. Dijkstra, "The Structure of the 'THE' Multiprogramming System," *Comm.ACM* 11(5), pp.341-346 (May 1968).
18. E.W. Dijkstra, "Structured Programming," in *Software Engineering Techniques*, ed. Buxton & Randell, NATO Scientific Affairs Div., Brussels, Belgium (1970).
19. E.W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey (1976).

20. D.A. Duce and A.H. Francis, "FINGS Primer & Manual," Prime User Note 17, Rutherford Lab., Oxon, England (July 1977).

21. D.A. Duce, "FOREST - A Structured Fortran Preprocessor," Prime User Note 5, Rutherford Lab., Oxon, England (Feb. 1977).

22. D.A. Duce and R.W. Witty, "ROOTS," Prime User Note 72, Rutherford Lab., Oxon, England (June 1979).

23. Ecclesiastes, "The Quest for Wisdom 1:10," in *The Bible*, ed. J. Stirling, The British & Foreign Bible Soc., London (1960).

24. O. Ferstl, "Flowcharting by Stepwise Refinement," *SIGPLAN Notices* 13(1) (Jan 1978).

25. M. Fitter and T.R.G. Green, "When Do Diagrams Make Good Computer Languages?," *Int. J. Man-Machine Studies* 11, pp.235-261 (1979).

26. R.W. Floyd, "Assigning Meanings to Programs," *Proc. Amer. Math. Soc. Symposia in Applied Maths.* 19, pp.19-31 (1967).

27. Frei, Weller, and Williams, "A Graphics-based Programming Support System," *SIGGRAPH 78, Computer Graphics* 12(3) (Aug 1978).

28. J. Guttag and J.J. Horning, "Formal Specification as a Design Tool," CSL-80-1, Xerox PARC (Jan. 1980).

29. M.H. Halstead, "Natural Laws Controlling Algorithm Structure?," *SIGPLAN Notices* 7(2), pp.19-27 (Feb. 1972).

30. Hamilton and Zeldin, "High Order Software - A Metholology for Defining Software," *IEEE Trans. on Soft. Eng.* SE-2(1) (Mar. 1976).

31. P. Henderson and R.A. Snowden, "A Tool For Structured Program Development," *IFIP 74* (1974).

32. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm.ACM* 12(10), pp.576-583 (Oct. 1969).

33. C.A.R. Hoare, "Computer Programming as an Engineering Discipline," *Electronics and Power*, pp.316-320 (Aug. 1973).

34. C.A.R. Hoare, "Data Reliability," *1975 Int. Conf. on Reliable Software* (April 1975).

35. C.A.R. Hoare, "Communicating Sequential Processes," *Comm.ACM* 21(8), pp.666-677 (Aug. 1978).

36. G. Jackson, "A Graphical Technique for Describing Recursion," *SIGPLAN Notices*.

37. M.A. Jackson, *Principles of Program Design*, Academic Press (1975).

38. J.B. Johnston, "The Contour Model of Block Structured Processes," *Proc. Symp. on Data Structs in Prog. Lang in SIGPLAN Notices* 6(2) (Feb. 1971).

39. A. Jonsson, "DIMED Dimensional Flowcharting Editor Manual," LiTH-ISY-I-0362, Linkoping Un., Sweden (Nov 1980).

40. B.W Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass. (1976).

41. D.E. Knuth, "Computer Drawn Flowcharts," *Comm.ACM* 6(9), pp.555-563 (1963).

42. D.E. Knuth, "Structured Programming with GOTO Statements," *Computing Surveys* 6(4) (Dec. 1974).

43. P.J. Landin, "The Next 700 Programming Languages," *Comm.ACM* 9(3), pp.157-166 (Mar. 1966).

44. H.W. Lawson, *Understanding Computer Systems*, Lawson, Sweden (1979).

45. H.F. Ledgard, "The Case for Structured Programming," *BIT* 13, pp.45-57 (1973).

46. C. Lindsey, "Structure Charts. A Structured Alternative to Flowcharts," *SIGPLAN Notices* 12(11) (1977).

47. B.J. MacLennan, "Observations of the Differences Between Formulas and Sentences and their Application to Programming Language Design," *SIGPLAN Notices* 14(7), pp.51-61 (July 1979).

48. J. Malone and R.W. Witty, "Automated DFC on the P400," Soft. Eng. Technical Paper 2, Rutherford Laboratory (Nov 1978).

49. T.J. McCabe, "A Complexity Measure," *IEEE Trans. on Soft. Eng.* SE-2(4) (Dec. 1976).

50. H.D. Mills, "Syntax-Directed Documentation for PL360," *Comm.ACM* 12(4) (April 1970).

51. H.D. Mills, "Mathematical Foundations for Structured Programming," FSC 72-6012, IBM Corp, Gaithersburgh, Maryland (Feb. 1972).

52. H.D. Mills, "Software Development," *IEEE Trans. on Soft. Eng.* SE-2(4) (Dec. 1976).

53. D.T. Moore and T.A. Galloway, "Structured Diagrams - a Software Design Tool," *Pragmatic Programming Proceeding*, Online (Feb. 1978).

54. D.R. Musser, "Abstract Data Type Specification in the AFFIRM System," *IEEE Trans. on Soft. Eng.* SE-6(1), p.24 (Jan. 1980).

55. G.J. Myers, *Software Reliability: Principles & Practices*, Wiley & Sons (1976).

56. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices* 8(8), pp.12-26 (Aug. 1973).

57. P. Naur, "Proof of Algorithms by General Snapshots," *BIT* 6, pp.310-316 (1966).

58. P. Naur and B. Randell, *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, Brussels (1968).

59. J.von Neumann, "Planning & Coding of Problems for an Electronic Computing Instrument, Princeton,1947," pp. 80-151 in *Von Neumann's Collected Works, vol 5*, ed. A.H. Taub, Pergamon, London (1963).

60. J.von Neumann, *Theory of Self Reproducing Automata*, U.of Illinois Press (1966).

61. E.I. Organick and J.W. Thomas, "Computer-Generated Semantics Displays," *IFIP 1974*, pp.898-902 (1974).

62. D. Parnas, "On a Buzzword: Hierarchical Structure," *IFIP 74*, pp.336-339 (1974).

63. L.J. Peters and L.L. Tripp, "Comparing Software Design Methodologies," *Datamation* (Nov. 1977).

64. G.D. Pratten and R.A. Snowden, "CADES - Support for the Development of Complex Software," *Eurocomp*, Online Ltd (Sept. 1976).

65. B. Randell and L.J. Russell, *Algol 60 Implementation*, Academic Press (1964).

66. C.W. Rose, "LOGOS and the Software Engineer," *Proc. Fall Joint Comp. Conf.*, p.311 (1972).

67. D.T. Ross and K.E. Shoman, "Structured Analysis For Requirements Definition," *IEEE Trans. on Soft. Eng.* SE-2(4) (Jan. 1976).

68. J.M. Rushby and R.W. Witty, "FR80 Logging System," FR80 Technical Paper 4, Atlas Computer Lab. (April 1975).

69. B.G. Ryder, "The PFORT Verifier," *Software - Practice & Experience* 4, pp.359-377 (1974).

70. G.G. Scarrott, "Wind of Change," *ICL Technical Journal* (1978).

71. R.S. Scowen and A.R. Lawrence, "Some Experiments in Improving Program Documentation," *Int. Computing Symp. 1973*, North Holland (1974).

72. M. Shelley, "Computer Software Reliability, Fact or Myth?," TR-MMER/RM-73-125, Hill Air Force Base Utah (1973).

73. H.A. Simon, "The Architecture of Complexity," *Proc. American Philosophical Soc.* 106(6), pp.467-482 (Dec. 1962).

74. J.E. Stockenberg and A. van Dam, "STRUCT Programming Analysis System," *IEEE Trans. Soft. Eng.* SE-1(4) (Dec. 1975).

75. D. Teichroew and E.A. Hershey, "PSL/PSA : A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. on Soft. Eng.* SE-2(4) (Dec. 1976).

76. R.H. Thayer, "Rome Air Development Center R&D Program in Computer Language Controls and Software Engineering Techniques," RADC-TR-74-80, Griffiss Air Force Base, Rome, N.Y. (1974).

77. J.D. Warnier, *Logical Construction of Programs*, Stenfert Kroese B.V., Leiden (1974).

78. G. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold (1971).

79. R.J. Wilson, *Introduction to Graph Theory*, Oliver & Boyd, Edinburgh (1972).

80. N. Wirth, "Program Development by Stepwise Refinement," *Comm.ACM* 14(4) (April 1971).

81. J. Witt, "The Columbus Approach," *IEEE Trans. Soft. Eng.* SE-1(4) (Dec. 1974).

82. R.W. Witty, "FR80 Driver Software Construction," FR80 Technical Paper 21, Rutherford Lab. (Dec. 1975).

83. R.W. Witty, "Dimensional Flowcharting," *Software - Practice & Experience* 7, pp.553-584 (1977).

84. R.W. Witty, "FR80 Driver," FR80 Technical Papers, Rutherford Lab. (April 1977).

85. R.W. Witty, "Design & Construction of Hierarchically Structured Software," *Proc. Pragmatic Programming & Sensible Software*, pp.361-388, Online (1978).

86. R.W. Witty, *On the Calculation of Factorial or Recursion & Iteration expressed as Communicating Sequential Processes*, Mar. 1978.

87. E. Yourdon and L.L. Constantine, *Structured Design*, Yourdon Inc, New York (1975).