INTERACTIVE USE OF A COMPUTER IN THE PREPARATION OF STRUCTURED PROGRAMS

R.A. SNOWDON

Ph.D Thesis April 1974

University of Newcastle Upon Tyne

Acknowledgements

I gratefully acknowledge the encouragement given to me by many colleagues during the course of this work. In particular, I should like to thank Professor B. Randell for his supervision and for his critical reading of my manuscripts. Professor J.J. Horning must also be acknowledged for suggesting the acronym "Pearl".

I would also like to thank Miss Moira Dearden for being a most efficient typist and for her patience in waiting for the thesis in its final form.

Finally, thanks must also go to my wife for her understanding, particularly during the period of preparation of this thesis.

The research described here was supported by the Science Research Council.

ABSTRACT

An experimental system providing assistance in the task of program construction, validation and description is presented. This system (Pearl) encourages a particular top-down approach to programming such that programs so developed exhibit a multi-level, hierarchical structure.

Amongst several tools provided by the system is one which enables programs to be exercised even though they may be expressed in terms of abstract operations and data types.

The whole system is designed to be used in an interactive environment. Programs are developed by the programmer with appropriate assistance and guidance from the computer.

Contemporary programming tools and methods are surveyed and their relevance to the development of high quality software is discussed. In particular attention is given to programming methodologies, design representations and issues of program correctness.

The practicality of the system is demonstrated in a number of examples.

Contents

Chapter 1:

Introduction

The tools and techniques used in the construction of computer programs have evolved rapidly during the short history of computers. This rapid evolution has resulted in the current position whereby there is a great variety of such tools and techniques in use, each more or less suited to particular programming activities. It has become increasingly apparent that this variety is not itself sufficient to enable the construction of programs which will allow computers to perform the ever more complex tasks demanded of them (e.g. Naur and Randell 1969, Buxton and Randell 1970). In society, the reliance that is placed upon the correct functioning of computer systems is increasing at a great rate (e.g. air-traffic control systems, banking systems etc.). It is true, therefore, that society, and particularly the individual within society, will become more vulnerable unless a higher degree of confidence can be placed in the correct functioning of such systems. Thus it is crucial that ways are discovered by which computer systems may be constructed in order that such confidence may be justifiably expressed.

This thesis is concerned with an investigation into a number of aspects of programming which have a direct bearing upon the quality of the software component of a computer system. There are undoubtedly problems concerning the reliable function of computer hardware. Such problems, however, are left to other workers.

The research reported in subsequent chapters follows closely many of the ideas of "structured programming" as illustrated by Dijkstra in a number of papers, but primarily in Dijkstra (1972a). In order to demonstrate why we believe that the programming techniques which are subsumed by the general term "structured programming" are so important, it

is necessary to appreciate what is involved in the task of writing a computer program. Indeed, as Dijkstra (1972b) points out so clearly, it is essential that we realize that programming is an extremely difficult task. A very succinct analysis is given by Ershov (1972).

Computer programming may be regarded as a complex problem solving activity. Simon (1969), Hormann (1970) and Koestler (1964) are amongst several writers who have attempted to describe the problems of complexity and how human beings can overcome them.

Much of the recent work on how complex programs are developed has stressed the importance of hierarchies and levels. (Zurcher and Randell 1968, Wirth 1971b, Woodger 1971, Dijkstra 1968b and 1972a). These ideas accord well with those of those authors mentioned above concerning more general complexity.

Confidence in the trustworthiness of a program comes ultimately from its observed behaviour when executed by a computer. This fact has long been recognized and has spawned many of the tools used by programmers at present (e.g. debugging tools, testing procedures, etc.). The usefulness of such tools should not be overlooked in the development of a program, despite the fact that their use cannot guarantee the absolute worth of complex software.

Program proof methods represent further attempts at generating confidence in a program. Floyd (1967a) and Naur (1966) describe methods by which the properties of a program can be checked against assertions representing the intention of that program. Tools have been described (King 1969, Good 1970, Deutsch 1973) which assist the programmer in the generation of such proofs.

Program proof methods may also be used during the development of a program to ensure that it is correctly constructed (Naur 1969,

Hoare 1971a, Allen and Jones 1973). Floyd (1971) describes how a tool might be constructed to assist in this process. Less formal methods may also be applied during the development of a program to make it more likely that the program will exhibit the appropriate properties. Zurcher and Randell (1968), Mills (1971) and Baker (1972) all describe how program development may be aided by the use of tools based upon such methods.

A major goal of the present research has been to combine and analyse these and other somewhat separate ideas and to use them as a basis for a coherent design methodology which is explicit enough to be embodied in a tool to assist the complete programming task. This tool takes the form of a (prototype) computer system which acts as a data base for the design of programs. Programs may be developed by the programmer by entering textual information which represents additions to the incomplete design. The form of this information is based upon a notation which encourages the representation of programs in a highly structured, hierarchical manner. In addition the programmer is encouraged to follow a particular development method so as to gain full benefit from the system during the early stages of his design. The system provides the programmer with a number of explicit facilities, each aimed at improving his understanding of the program as it is developed. These include aids in checking the logical consistency of input information, execution of partially developed programs, certain debugging facilities and a number of interrogation mechanisms. The system has been designed and implemented as an online, conversational system.

The following chapters form two distinct parts. The reader who is only interested in details of the implemented system is recommended to omit Chapters 2,3 and 4. These describe and discuss certain aspects of

program construction, and techniques and tools presently available to assist the programmer.

In Chapter 2 a simple view of programming is taken. This is described in terms of three elements.

(i) problem

(ii) man

(iii) machine

Each of these elements is considered in turn, although most attention is given to "man" and to the interfaces between man and problem and man and machine.

Attention is paid, in Chapter 3, to the ideas of structure and method in the representation and development of a program. An informal notion of a level of description is given whereby a program may be represented in terms of concepts which capture some essential property of the problem or the programming language, but not necessarily all such properties. A program may be represented at a number of different levels of description related according to their various mutual interpretations. Different methods of developing a program are described and discussed using the notion of representation at many levels of description.

Chapter 4 presents a discussion of various tools and techniques to do with establishing the correctness of computer programs. These range from proof techniques (applied both to a given program and to the development of a program), to program testing tools and other mechanical aids which may help the programmer to increase the level of confidence he may have in his program.

Chapters 5 and 6 give details of the experimental system referred to above. Chapter 5 serves to introduce the system and to describe how it enables a programmer to build up a program according to a particular design method. Chapter 6 describes the more extensive features of the system enabling design evaluation, interrogation and re-appraisal. A number of examples of the system in use are given in these chapters. Further, and more complete examples are to be found in appendices D, E and F.

The experimental nature of the system has generated a number of interesting points of discussion. These are grouped together in Chapter 7. Here, also, are presented some conclusions on the relevance of such systems as an aid to the programming activity. These are, of course, to a certain extent limited by the prototypical nature of the implemented system. However, we feel that most of them are valid in a wider sense.

Chapter 2:

Basic Elements of Programming

There are at least three elements which are basic in programming. One is the machine for which the program is being written, the second is the problem (or task) which is the reason for the program and the third is the programmer (or programmers) whose job it is to construct the program from an understanding of the problem and the properties of the machine. The job of the programmer is, (Dijkstra 1972b, Ershov 1972) very difficult and represents a significant intellectual challenge. Amongst reasons for this are the inherent complexity of the tasks for which computers are used and of the computers themselves, and also the requirements of the program as being amongst other things, precise, adaptable, extendable, well-documented and correct.

In this chapter we study the effect on programming of these three basic elements in order to give some insight into the actual sources of complexity and of ways by which the difficulties can be reduced. In particular we discuss programming as being a problem solving activity in order to relate wider observations of creative human activity (e.g. Polya 1945, Koestler 1964, Simon 1969, Hormann 1970) to the construction of programs. Such a discussion allows a number of observations to be made as to the appropriateness of certain tools which are often used in program construction (e.g. flowcharts, decision tables, particular programming languages). The observations we make in this chapter are mainly of a critical nature. A more constructive approach is taken in later chapters.

2.1 A view of programming: the basic elements

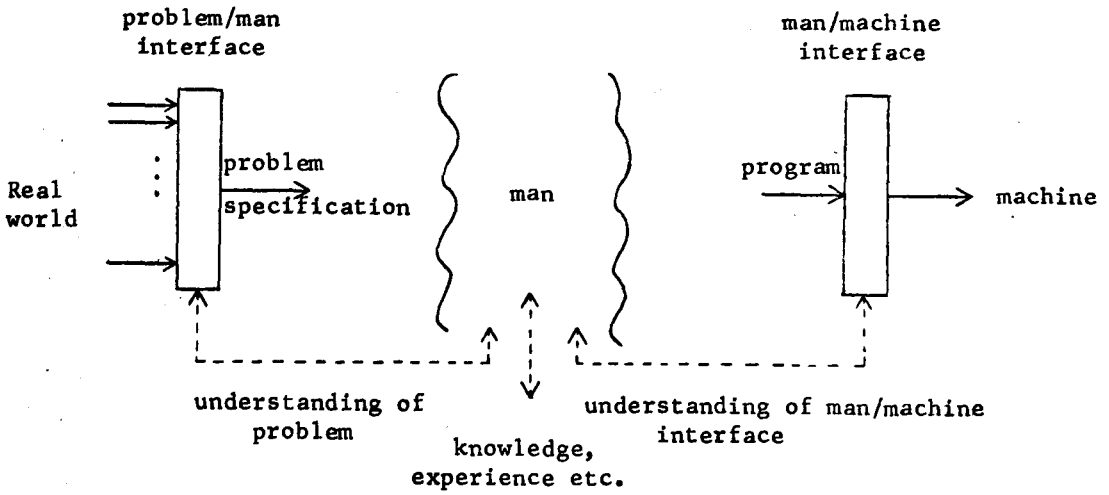


Figure 2.1

Figure 2.1 represents a simplified view of the programming activity. The central element is the programmer. He has two interfaces. One, the problem/man interface is with the outside world; the other, the man/machine interface, is with the computer (the machine).

The programmer accepts (understands) the specification of a problem in the outside world. His task is to develop a solution in the sense of describing, in a program, the process which the machine must carry out to generate the answer to the problem. This process we shall call the solution process. (We have not made figure 2.1 complete but only included those concepts which are appropriate for the discussions of this chapter. We have not, for example, shown how the results of a program execution can influence the programmer. We describe some extensions in this direction in Chapter 4).

The interface which the programmer has with the outside world is hard to characterize. We intend that this interface should include all methods by which the programmer obtains information about the problem. Problem specification is a difficult task itself and thus it is

hard to state more exactly what form this interface takes. A short discussion of how information about problems is supplied and understood is given in section 2.3.1. Of course it is often the case that details of problems are only uncovered as part of the development of the program. Thus it is not in practice the case that the problem/man interface is divorced from the actual design activity. A part of this interface, therefore, represents interactions arising during the task of program construction.

The program which describes the solution process is generally written in some notation representing concepts which have no direct physical existence in the hardware of the machine. This notation, the programming language, therefore acts as the interface between the programmer and the machine. The influence that this interface has upon the construction of programs and other discussion is given in sections 2.3.2, 2.3.3. and 2.4.

2.2 Program construction as a problem solving activity

Figure 2.1 may be interpreted to cover the development of solution processes to problems which do not require the construction of a computer program. The "machine" need not be a computer but could be any processing device, even a human being. The man/machine interface will then not be characterized by a programming language in the accepted sense, but more generally as some medium for communication.

examples:

(a) A theorem to be proved in mathematics is a problem. The mathematician who solves this problem responds to the stimulus of the statement of the theorem by developing a proof written in some mathematical system. This proof describes the "solution process" to be followed whereby the truth of the original theorem may be accepted. A machine which carries out this "solution process" might be a colleague or perhaps the reader of a book.

(b) An architect may be asked to design a building according to some specification as to its purpose, its location and its estimated cost. The architect accepts this specification and draws up an appropriate design. This design is a solution process for his problem. The "machine" which reacts to the design may be the builder or perhaps the client who wishes to appraise the architect's work before finally committing himself.

There are, thus, parallels which can be drawn between computer programming and other design activities. More generally, as Ross (1967) points out:

> "design is a special term for some ill-defined type of problem solving".

Problem solving is generally thought of as being some process by which possible solutions to a problem are tested for their adequacy. Consider the following problem:

> "Find those numbers, whose absolute value is a natural number smaller than 100, whose square is 36".

A way of finding the solution to this problem would be to consider every number and test it to see if it had the stated properties. The solution would then be the set of numbers found to satisfy this test.

In this context, the process of solving a problem is taken to mean that the actual numbers should be determined and displayed. It is, however, also necessary to determine whether or not the "solution process" by which the results are determined is itself adequate.

The solution process described above is of the form:

> "Pick a number from the set of all numbers. If this number has an absolute value smaller than 100 and a square of 36, then accept it, otherwise reject it. Repeat this process for all numbers".

Clearly this solution process is itself inadequate and should not be accepted. It is necessary, therefore, that, from amongst the set of solution processes for this problem, a better one be chosen. Problem solving may be thought of as being a process of examining the various solution processes themselves for being acceptable. Although one of the criteria of acceptance should, of course, be that the solution process will, indeed, produce the required solution to the stated problem, this is by no means the only one which should be applied. As we discuss more fully in Chapter 4, it is, in fact, a criterion which is very difficult to apply with confidence in computer programming.

It is possible to identify two (at least) separate problem solving activities in programming. Both parallel the views of problem solving described above.

The first is the task of choosing a particular representation within a programming language to fulfill a function whose properties are understood by the programmer. Such a situation can easily arise when a programmer recognizes a problem for which he knows an acceptable solution process,

but which is represented interms different from those of the language in which he must write his program.

examples:

(a) Algorithms published in the literature are often written in Algol whereas, for one reason or another, the programmer must write his program in FORTRAN.

(b) Algorithms expressed in a descriptive manner using natural language (e.g. Knuth 1968).

This is a problem solving activity by which the programmer makes a choice from amongst the features of the programming language. In particular the programmer must generally apply some judgement as to whether one representation is more suitable than another.

The second problem solving activity which we identify in programming is that of the derivation of the solution process itself from the statement of the problem. If it is required to construct, for example, an airline seat reservation system, then it is necessary to decide which computations must be carried out before encoding them in a programming language. Of course, a programmer in this situation will use his knowledge of the properties of any hardware or programming language he may use, as a guide in the overall design. However, the activity which is being followed is separate from that of encoding a solution process which has already been derived. It requires, as in the example above, that different possible solution processes must be examined until one which is adequate is accepted. We believe that this latter viewpoint of problem solving in programming is the most important as, in general, it includes the representation problem.

2.2.1 Method

A primitive notion of problem solving is that it is a process of casting through a set of possible solutions until one is discovered which is acceptable for the problem under investigation. This notion requires further elaboration in the context of programming (and probably in any other non-trivial problem domain).

In the example given in the previous section where the problem is to find certain numbers, the space of possible solutions has an accepted representation which allows each possible solution (a number) to be identified. Further, the properties of the members of the solution space allow of the possibility of some scheme whereby individual "solutions" can be chosen methodically (i.e. the ordering properties of numbers). As is described below a knowledge of such properties is almost essential in the derivation of an acceptable solution process to this problem. In programming, the space of possible solutions has a less well-understood representation and has properties which are often too complex for programmers to appreciate. Even if the problem is merely that of choosing a representation for a solution process otherwise described, few programmers would claim that the representation they have chosen was the best. It is apparent, as we shall describe in a little more detail in section 2.4.2, that the very power of programming languages in some cases adds to the complexity of programming, rather than reducing it.

In the derivation of a program as a solution process, there is a difficulty in the identification of individual elements from the space of possible solution processes (i.e. the space of all programs). What a programmer does, of course, is to use properties he requires in order to derive possible programs which he may then examine. However, the properties he may require of a program are often poorly understood owing to

a lack of a clear and complete specification of the problem (see section 2.3.1) and also because of his lack of knowledge of the properties of the programming language. Thus it is difficult for a programmer to know what he is deriving, and also when he has a program which satisfies his requirements.

Nevertheless the notion of searching allows a basis for a discussion of how complex problems may be tackled. A complex problem may have many possible solutions, all of which ought to be examined. However, it is impossible to do this in a reasonable time. Ways must be sought by which the space to be searched can be drastically reduced in order to focus attention upon an area where an acceptable solution is most likely to be found.

Consider the following steps in the derivation of an acceptable solution process for the numbers problem of the previous section.

1. The set of "possible solutions" may be divided into the real numbers and the complex numbers. From the properties of complex numbers it can be seen that an acceptable solution process need only consider members of the set of real numbers.

2. Only the set of integers $\{-99, \ldots, +99\}$ need be considered because of the definition of absolute value.

3. The set $\{-99, \ldots, +99\}$ may be partitioned into $\{-99, \ldots, -1\}$ and $\{0, \ldots, +99\}$. The solution process need only consider the set $\{0, \ldots, +99\}$ and, for any solution found in this set (different from 0), select also the corresponding negative value of this solution from $\{-99, \ldots, -1\}$ as a solution.

4. An acceptable solution process may be described over the set $\{0, \ldots, +99\}$ which searches from 0 in increasing magnitude of number and which terminates as soon as a number is found whose square is greater than or equal to 36 (as $(x + 1)^2 > x^2$ for $x \in \{0, \ldots, +99\}$).

At each step, the set of possible solutions is further limited until, at stage 4, a solution process can be described which, we suggest, is acceptable. There are other solution processes which could equally have been suggested at stage 4 (e.g. one which commenced with +99 and then continued with 98 etc.). Thus, even in this derivation, there is a choice of solution processes. (Of course further analysis of the set of possible solutions can, in this example, reduce the set of possible solutions from the positive integers to a single element).

The process of the analysis of information contained in the problem statement and of known properties of the space of possible solutions is a means for reducing the set of possible solutions that need be considered. In this example the steps of methodical reduction can be clearly expressed because of the well-formed nature of the problem and because the properties of the solution space are well-understood. However, even in solving problems which are ill-formed or whose solution spaces are incompletely understood by the problem solver, the value of a methodical step-wise investigation has been stressed by several writers (e.g. Polya 1945, Alexander 1966, Mannheim 1966, Simon 1969, Hormann 1970). In programming, also, a similar appreciation has found expression in such ideas as "structured programming" (Dijkstra 1972a) and "step-wise refinement" (Wirth 1971b). We discuss these ideas further in Chapter 3.

In general, the methods suggested by these writers may be characterized as the decomposition of a problem into smaller problems for which, individually, there is a greater likelihood of an acceptable solution being discovered.

example:

> In the derivation given for the numbers problem above, it may be seen that each step represented a decomposition of the stated problem into problems of conceptually less complexity.

The decision to decompose a problem in a particular way is based upon some expectancy of where solutions are to be found or, alternatively, of where solutions are not expected to be found.

example:

> At step 1 above, the decomposition is based upon the "ease of solution" of the problem of finding complex numbers whose square is 36.

In taking these decisions, the problem solver must carry out some form of analysis. In many complex problem solving situations, the validity of such analysis is often not decidable at the time the decision must be taken. It may be only at a much later stage in the problem solving activity that decisions taken earlier are found to be valid or invalid. Whether or not such information can then be used to turn the search for an acceptable solution in other directions depends upon the ease with which a change of direction can be made. In programming a particular decomposition of a problem is often reflected in the modular structure of the program. Each decision concerning the decomposition is therefore embedded in program code. According to the

way in which the structure of the program is represented in the code, it may be a difficult task to alter the program even though the decomposition is demonstrably unsuitable. This phenomenon is, of course, closely related to the forms available for representing the design of a program. This is discussed further in sections 2.3 and 2.4 and also in Chapter 3.

The comments we have made concerning the difficulties to be faced in tackling complex tasks are related more closely to the development of programs in Chapter 3. Our intention at this time has been to draw attention to the fact that programming is a complex and difficult task, but that man has been faced with such tasks before and has developed mechanisms for overcoming them. An insight into what these mechanisms involve can only be of help in deciding how programming should be carried out.

2.2.2 Some "human aspects"

The natural abilities of an individual human being as a problem solver will have a great influence upon the success of that individual when faced with problems of great complexity. Although the means by which complex problems can be tackled may be well appreciated, it is still necessary that the appropriate feats of intellect are accomplished.

It is surely necessary for a programmer to be creative. The sheer immensity of the task of constructing a program requires an individual flair for assimilating apparently unrelated information or for taking the "right decision" even when there is little substantiating evidence. Koestler (1964) describes a possible mechanism to account for the "flash of inspiration" and the "moment of insight" which are so necessary in the task of tackling complex problems. Hormann (1970) characterizes the application of knowledge and experience in problem solving and relates these to an individual's creative abilities in a particular task. The

characterization which he gives is expressed in terms of "prepared" and "unprepared conditions". "Prepared conditions" represent situations recognizable by an individual from his experience. Hormann uses these characterizations to explain a number of observations concerning the ways in which problems may be overcome by a human being. In particular, he discusses the possibility that an individual can solve a problem which he has not previously encountered by means of a mismatch between some prepared condition (representing an earlier experience) and the given problem. Such mismatches can occur if the given problem is, in some sense, similar to the previous experience. A danger here is that a gross mismatch between a problem and some prepared condition may be undetected and lead to the acceptance of incorrect solutions to problems. Unfortunately, a programmer who is pressed to attain production schedules is more likely to commit such errors than a programmer who has time to consider his task with care.

Both Koestler and Hormann attempt to give explanations for an individual's problem solving ability. It is interesting to remark that Polya (1945), in giving rules to follow in solving problems, suggests that a person should consciously try and match his past experience to any problem with which he is faced. Polya states that one should always ask oneself whether the problem has been solved before, and failing an affirmative answer, ask whether any similar problem has been solved before. There is an obvious similarity between these suggestions and the mechanism described by Hormann.

Creativity is, therefore, one characteristic which we believe is essential in a programmer. Weinberg (1971) discusses a number of others. Amongst these is humility. A good illustration of the need for humility is given by the phenomenon of "ownership" described by both Weinberg and Ershov (1972). A programmer is likely to develop protective instincts towards his program because it represents a large intellectual effort on his part. As a result, a programmer may even jealously guard his work, whether or not it is of any worth. The consequences of such an attitude, particularly within a programming team, may be imagined and prompted Weinberg to promote the concept of "egoless programming". Under this approach, a program is written, not by an individual but by a group of people such that no one person feels responsible for it. The success of such a policy depends upon the readiness of all programmers to accept the suggestions of others for the overall good of the program. Such a requirement may, in fact, make it a difficult policy to adopt, but the arguments upon which it is based cannot be questioned. When an individual is working alone on a program, it would still seem to be a wise policy for him to remember that he is fallible and therefore likely to produce a program which may need correction or improvement.

A programmer also needs to be both suspicious and trusting. He should always be wary of possible difficulties and inconsistencies in the task he is required to do and yet must have confidence in his own ability to produce a satisfactory program.

There are, of course, many other aspects of human nature which are relevant in a consideration of programming. A programmer must be able to arrange his work in a methodical manner, be able to organize the information with which he is faced and even overcome boredom induced by the tedium of encoding familiar constructions. An extensive discussion

is given by Weinberg (1971).

2.3 Understanding problems and design

How a programmer understands the problem he is to tackle, the form the programming language takes and the tools which he may use in program construction will play a large part in shaping the eventual program. In the next few sections we discuss some of the issues involved and consider some of the tools which are available to the programmer to use as he designs a program. At this time we are considering only tools which may be thought of as design aids. Others tools, which, though affecting the programming activity are more concerned with program testing or validation are discussed in Chapter 4.

2.3.1 Problem specification

The specification of a problem can and does take various forms. Rarely is the specification of a complex problem sufficient in itself. The programmer will, therefore, find that he needs to discover answers to questions about the problem which arise as part of the development of his program. The difficulty is natural and may occur for a number of reasons. We suggest three, although there are probably many more.

(i) The form of problem specification is incomplete or open to a number of different interpretations.

(ii) The problem itself may be changing with time.

(iii) The problem is so complex that it cannot be expressed succinctly in a sufficiently rigorous manner.

The specification of problems may take many forms. Natural language and jargon are often used, with the danger of misunderstanding or incompleteness. A number of workers (e.g. Rose 1966, Kolsky 1969, Falkoff 1970) advocate the use of a subsidiary programming language (APL) to specify or describe programs. It may be possible to apply these techniques

more generally to the specification of problems. Some discussion of the use of particular languages for problem specification is given in Naur and Randell (1969). Parnas (1972) gives a technique for specifying modules in a program design in terms of functions which describe the purpose of a module. This technique appears promising in those cases where it has been tried.

It is, however, probably true to say that no one technique or language can be sufficient. It is likely that there will always be a need for explanatory material in addition to any formal description of a problem (e.g. an exposition of terminology, a language manual etc.).

One comment which we venture to make is that the form of the problem specification can be suggestive as to the form the solution might take. Notation and other devices used in the design of a program play their part in the form of that program, so it is likely that this observation extends also to the manner of the problem specification.

example:

> A programming problem might be described by a "procedural specification" intended to illustrate a flow of information. Such a specification can colour a programmer's thinking to a greater extent than if the problem was described in a "non-procedural" manner.

2.3.2 Design and documentation

Apart from the programming language, the influences most likely to bear upon the design of a program are the tools and techniques used.

By the use of various notations or other design aids, the programmer may learn more about a problem, and some of its peculiarities as well as experiment with possible solutions.

Many of these notations can be used in documents describing either the purpose of the program or its design. Documentation plays an important role in program construction. Most programs which are intended for more than "one-off" jobs need some description in terms more amenable to a human reader than that afforded by the code of the program itself. Potential users of the program will require knowledge of the purpose of the program, the format of the input data and control records, and the output they may expect. Other programmers may require more detailed descriptions of the program code so that they may maintain the program or modify it to local requirements. Such documentation can conceivably be written after the program itself has been written, though there may be some good arguments why this could be bad practice. For example, in many cases such documentation is generally provided by the programmer himself. Apart from the fact that programmers are not necessarily good at writing documentation (as pointed out by Weinberg 1971), interest in a program can naturally lessen when the creative phase has been completed. The programmer may even move on to other projects and leave the documentation to be completed by his successor, if it is ever properly completed.

It may, therefore,be a good idea to produce documentation directly from the program text using such techniques as automatic flow-charting or by other methods (e.g. Mills 1970).

Documentation of a design itself, made as the design is carried out, is particularly necessary in computer programming (see Naur and Randell 1969 p90, for example). In a project involving numbers of people it is essential. Several massive systems have been constructed (Brown 1970,

Falla and Burns 1973, Pearson 1973) to provide support for information on, for example, design specifications, program methods and progress. Baker (1972) describes how a programming secretary with machine assistance can play a central role in the maintenance of information. For small groups of programmers, a filing cabinet or even a notebook may be sufficient, if its value is fully appreciated.

The form of documentation used or required can influence the work of a programmer.

example:

> It is very much easier to document a program in terms of separately describable modules with few cross-references than one which makes use of intricate relationships amongst a large number of variables and functions.

This influence is likely whether there are many people involved or only one. Being forced to describe a program leads one to appreciate its shortcomings.

As a program is developed it should be documented so that the decisions taken during development and the reasoning behind these decisions will be available later. The development process may well be based upon such information.

example:

> If, in a particular development, the designers maintain a diary of progress made, then they are well equipped to use such information to influence their work. In the absence of such documentation it is likely that future decisions will be ill-considered or invalid with respect to earlier, undocumented and hence forgotten, decisions.

It is likely that well-considered programs are the result of well-documented designs. The converse, that badly documented designs

result in badly considered programs is likely to be an understatement.

Selig:

> "With the rapid proliferation of computer languages, subroutines and programs, and the tremendous effort they represent, meticulous documentation is becoming essential, not just to save money but to prevent chaos".
>
> (Naur and Randell 1969 p116)

Before we consider a few tools and notations used in the design of a program, it must be stressed that documentation is something which is for the benefit of a human reader. Its purpose is to enable a human being to come to an understanding of the program or design being documented. When the documentation is purely descriptive then this need should normally be achievable. However, documentation which is precise is also a requisite in programming and it also should be comprehensible. The method of Parnas (1972) for describing the function of program modules or the use of subsidiary programming languages to describe a program (Rose 1966, Kolsky 1969, Falkoff 1970) are of relevance in this direction.

2.3.3 Some tools used in program design

There are a number of tools available to a programmer for use during program design. Many of these are notational or graphical and facilitate the representation of ideas on paper. We also include a short discussion on machine-assisted tools, but only in the sense of special purpose computer-aided design systems. Machine assistance in the form of compilers, debugging systems or interactive programming systems is dealt with in Chapter 4. The discussion of programming languages at this time is also restricted to their use in design, rather than as being a definition of the interface between man and the computer. Programming

languages are discussed from this latter viewpoint in section 2.4.

Decision tables represent a method of describing the logical connections inherent in a problem (or in any process). In particular they provide

> " . . . a means by which the work required to understand and define a problem, develop and program a solution and provide documentation, is substantially reduced".
>
> (Schmidt and Kavanagh 1970).

However, decision tables alone do not provide a basis for the solution of complex problems. The derivation of a solution in terms of a decision table implies a good understanding of the problem so that the logical connections are correctly established between the various components of the problem.

Once the necessary logical connections are established, decision tables may prove of value in determining such properties as logical completeness. They can also be used to describe the solution process for a problem in a way which may be automatically translated into a representation in a programming language (see, for example, several papers in McDaniel 1970).

It is possible to use decision tables to give many-levelled descriptions of a problem or a solution process. (A discussion of "levels" is given in Chapter 3). The derivation of such descriptions is determined solely by the programmer himself, with the properties of decision tables only acting in a passive role.

Flow charts may be used in similar ways to decision tables. They refer, however, to the flow of action or information, rather than to fixed logical relationships. As the actions may be determined as a result of previous actions described in the flow chart, the generality of flow chart descriptions may be difficult to understand.

It is possible to code directly from a flow chart into a programming language possessing similar primitives to the primitive flow chart symbols (e.g. labels, goto's, functions, tests).

Flow charts may be used, like decision tables, to represent a many-levelled description of a problem or solution process. In this case, however, each level represents a description of a flow of control, rather than of levels of logical connection. If flow charts are used in this way to describe processes, the programmer must himself have a conception of the different levels of control flow and ensure that these are faithfully represented by the description he gives.

Various textual notations are often useful during the design of programs.

Natural language is a common method of description. It offers a means of communication with other people (in either written or spoken form) which is essential if the various facets of a complex problem are to be appreciated. The use of natural language in an unrestricted way is always open to the danger of misinterpretation, but "jargonized" forms can be very helpful whilst avoiding the implications of specialized notations such as programming languages. It is quite possible to describe algorithms in this way (as Knuth 1968 demonstrates so well), provided the terms used are unlikely to be misinterpreted.

Programmers often make use of a "bastardized" form of a programming language in the development of programs. Such a notation retains much of the flavour of the programming language but, as there are no stringent grammatical rules to follow (the programmer is, in effect, devising the language as required) the programmer can express himself as he pleases. The use of such language forms is likely to be beneficial in bridging the gap between the language of the problem statement and the programming language to be used to express the solution process (see also section 3.2.2.).

On a similar theme, any simplifications to the precision of a programming language are likely to be helpful in a notation whose primary use is for the expression of ideas. An example is an expression of non-determinism. Programming languages are, by nature, deterministic. Yet many programs are describing non-deterministic concepts. These programs are often characterized by a "choice" of a particular indeterminate value with appropriate backtracking provisions if the choice was, in fact, the wrong one. It may be helpful to the programmer if he could write his program using non-deterministic constructions where applicable, but without the need to give full details of how the backtracking mechanism should be incorporated. Floyd (1967b) and Johansen (1967) describe how programs which use non-deterministic constructions may be expanded in an automatic way so that the necessary backtracking mechanisms are incorporated. (Unfortunately, the generality of such schemes necessities the inclusion of much inefficient and often unnecessary computation. This can, of course, be removed by "hand tuning" the program, although this may be a non-trivial and error-prone task).

There are doubtless many other concepts whose expression in a programming language is complex, but whose basic notion is well-understood and is easily expressed in a textual manner. Their use by the programmer in documenting his program design is likely to be beneficial. If they are easily mapped into "real" programming language constructs then the task of program development is again simplified.

The ultimate notation available to the programmer is, of course, the programming language itself. This we will discuss in detail in section 2.4 and also in Chapter 3. We believe that its usefulness in the design of the program is more by its influence then by its use as a primary design notation. Indeed, we believe that the use of the programming language itself early in the design process can be bad practice, as it represents a committment to a particular solution process at a time when much of the information which the programmer may be able to find out about his problem is likely to be undiscovered.

In some cases it may be possible to call upon machine (computer) assistance in the design process. The amount of assistance a computer may give varies through special purpose "computer-aided design systems" such as the LOGOS scheme (Glaser 1971) whereby the problem itself is represented in the computer system and the design of its solution aided and maintained also by the system, the AED approach to computer-aided design (Ross 1967) whereby various design packages, a programming language and a "culture" all act to assist the programmer, to systems giving purely clerical assistance. The work at Stanford (Engelbart and English 1968) on a computer system for the augmentation of human intellect,

and that represented by MATHLAB (Engelman 1968) are good examples of this latter form of computer aid. We could also include systems which are more oriented to the production of computer programs (e.g. APL). These systems are also discussed in Chapter 4. We see computer-aided design tools primarily as a means of reducing the intellectual effort required of a human being for tasks which are mainly mechanical but still absolutely essential (e.g. representation, organization and presentation of information). The unique ability of the human being in a creative role is crucial to any design or problem solving activity. Design aids which allow the human being to concentrate his abilities on this role are bound to be of use in extending the human capability for undertaking difficult tasks, such as program construction, with greater confidence.

2.4 Programming Languages

We have suggested that a programming language characterizes the man/machine interface. It is the aim of the programmer to describe a solution process in terms of a programming language, rather than in terms of the physical concepts of the computer. The programming language, therefore, has a very great effect upon the programming activity. Programming languages should be designed with some care in order that it be as straightforward as possible for the programmer to develop a representation for even complex solution processes.

The development of programming languages has tended to recognize this obligation, although we believe there is still a long way to go. Early computers were programmed in machine code and subsequently in a symbolic form of machine code. The man/machine interface was, at that time, only slightly removed from the machine and the programmer required a large intellectual effort to achieve a suitable encoding of his program. Later efforts

(e.g. FORTRAN, Algol, COBOL, etc.) were further removed from particular machines and paid a greater concern to the expression of problem solutions in a form more closely related to problems themselves. Nowadays, high-level languages have been devised for many of the more common computer applications (Sammet 1969).

Most recent language developments have recognized that the programmer will benefit greatly if he has to adapt the problem less to the peculiarities of a machine and is therefore able to concentrate more on the development of the solution process. A human being solving a complex problem has ample opportunity for error. The lessening of the problems of communication with the computer should allow more freedom to concentrate on the real difficulties.

The development of languages represents a steady process of movement away from the concept of a specific form of computer, and more to the general representation of problem concepts and algorithms. A logical conclusion to this development process would appear to be the use of natural language to communicate with the computer. There are many difficulties with this idea, and even were it practical from the point of view of implementation, it is likely to be a source of much misunderstanding. The "heaviness" of legal English should act as a warning that it is very difficult to write unambigious statements in natural language (Hill 1972). What would appear more appropriate is a language that takes due account of both man and machine, with little explicit emphasis on the latter and more attention given to the former. One way in which this may be possible can be seen in the concepts of extensible languages which allow the programmer to add to the basic language of the machine

interface as he thinks fit.

However, it should be stressed that our present concern is to study the role of programming languages in program development. We do not wish to be concerned with arguments about the form new programming languages should take.

2.4.1 Programming language influences

The choice of a particular programming language by a programmer theoretically acts as a constraint upon the number of actual solutions from amongst which he may choose for his particular task. However, any reduction is unlikely to be noticed unless the choice rules out particularly appropriate representations for the problem in hand.

The decision to use a certain language may not always be made on the basis of the merits of the language itself. Other criteria, often based on pragmatic arguments, can play a large part. Programmers may have to make do with ill-conceived language constructions and the likelihood of difficulties later simply because there is a "good" implementation of the language which generates "efficient" machine code and which is well supported by a large library of useful functions. Mass usage of such languages encourages their continued existence to the likely detriment of other concepts in programming which may, in the long term, offer great benefits. The blame does not lie with individual programmers as they are often given little choice in what programming language to use. Their organizations will make this choice for them, having considered (or tried to consider) factors other than that of the language itself. Compatibility and transferability of both programs and programmers are just two examples.

For whatever reasons a particular language may be chosen, it will have a considerable influence upon the way in which a program is developed and

possibly contribute to the difficulties.

Even with contemporary high-level languages which are described as being general purpose, the concepts directly describable are limited. In order to make use of a programming language to represent a solution process, the programmer has to create mappings from the concepts of the problem to those of the programming language. It is natural for a program to be developed along the lines suggested by the programming language as these mappings are then more easily appreciated.

example:

> If APL is chosen as the programming language, then a programmer is encouraged to think in terms of matrices and to consider his problem in such terms. Again, if a string processing language is chosen, a programmer is immediately encouraged to think in the particular terms that the language suggests.

In some circumstances the particular concepts of a programming language are well-suited for a given programming task (e.g. RPG for the construction and printing of tables of data). In general this is not true and thus a part of the programming effort is the choice of suitable representations for problem-oriented concepts in terms of the limited concepts provided in any one programming language. One way of reducing the effort required in this task would be the use of more powerful programming languages. However there is some danger in this approach, namely that the more powerful a programming language is, the more difficult it is for a programmer to appreciate its properties. If a language spans a large set of concepts then the difficulty of choosing the most appropriate representation increases, because there is a potentially larger set of candidates. Conversely, a language which is very restricted and so does not have this problem has, of course, difficulties of its own. A programmer may conceivably have a complete understanding of the properties of such a

language, but, for any given problem, it is unlikely that there exists any obvious, direct representation. The programmer has, therefore, to create one, which may be a non-trivial task. Thus, a programming language which is over restrictive is likely to lead to programming problems, whilst one which provides a vast set of concepts and functions is likely also to cause problems through difficulties in understanding. Extensible languages may prove to be a solution to this particular difficulty, provided that the mechanisms of extension are themselves non-complex whilst being sufficiently general.

If a programmer is free to choose from amongst a set of alternative languages then there is likely to be some advantage if the final decision is delayed. The process of developing a program allows a programmer time to learn about the problem and its difficulties. If he makes no committment to a particular language during the early stages of development then he is likely to be better placed to make a wise choice. The set of possible languages will probably be small (for reasons separate from the task in hand) and so the programmer should be well able to judge which language is best suited to his particular situation.

2.4.2 Programming language design

The problems for which computers are used are generally complex. The various properties and concepts of computers are complex. The interface between these two sources of complexity is the programming language. One function of a programming language should, therefore, be to offer means of simplifying both. This function is carried out by the various languages available with differing degrees of success as illustrated by some of the examples given below.

The number of languages with procedure or subroutine mechanisms which

may be used easily are good examples. The worth of such a concept (and more particularly the use of libraries of subroutines) is obvious when we recall the discussion of man's requirements for solving problems (i.e. the breaking up of the design, the recognition of situations etc.). Indeed, there may be a considerable effect upon the program design itself:

(i) The programmer is spared intellectual effort.

(ii) A program may be designed in a particular way to incorporate an existing subroutine.

example:

> In the solution of a boundary value problem of ordinary differential equations, the existence of a subroutine which solves initial value problems might encourage the programmer to use the "shooting method" (Keller 1968) rather than develop his own solution directly.

The presentation of the language itself can be a powerful simplifying agent. Flow charts may be described as programming languages, and they certainly allow for the concept of a subroutine call mechanism. Yet we do not normally consider flow charts as being suitable for the detailed representation of programs to be input to some machine. One reason of course, is that computers do not possess input devices capable of accepting such graphical information. Textual representation, however, offers a much more concise form for transmitting information and, because of education, is naturally acceptable to the human programmer. On the assumption that programs are to be understood by human readers, the actual symbols of the language, the relationships that may exist between these symbols, and the meaning to be attached to the symbols should be chosen so as to assume as little intellectual effort as possible from the reader. A programming language is likely to be more acceptable if it satisfies this property of

"readability", at least to the extent that comprehension of a program is not obscured by the constructs of the language itself.

example:

> A language such as PL360 (Wirth 1968) has some appeal when compared to the assembly language of the 360 computers. (IBM 1969).

To a certain extent, the clarity of individual programs depends upon the problem and on the ability of the programmer, not simply as a coder, but also as a problem solver and designer. However, as a brief survey will show, there are certain constructs present in current high-level programming languages which are extremely complex and liable, themselves, to lead to much misunderstanding. Even in well designed programs their use will obscure the basic design, whilst in badly designed programs, their use can make it almost impossible for the human reader to discover how the program works. Unfortunately the use of some of these constructs is often necessary. The programmer must then exercise discipline over himself to see that any complexity is reduced to a minimum. We discuss some of the points in the illustrations which follow.

(a) Input/output handling.

Undoubtedly, input/output handling can be a complex problem, but it rarely appears to receive the attention that it warrants in a language. Indeed in some languages, the handling of input and output is regarded as an "add on" feature to be determined by individual implementations. We do not advocate any particular approach for the specification of input/output, but certain methods seem to be more appealing than others.

example:

> The idea of a "picture" of the required output being given by the user (as in COBOL for example).

One feature commonly encountered is that of referring to devices by a number, instead of using a more meaningful name; such a technique is surely indicative of the half-hearted approach that seems to be taken in so many cases.

(b) "goto" statements

There has been much discussion in the literature regarding the efficacy of using "goto" statements in programs. (Dijkstra 1968c, Rice 1968, Wulf 1972, Leavonworth 1972, Hopkins 1972). The arguments for and against are well-known and we will not discuss them further here, though we will return to the "goto" statement briefly in Chapter 3.

(c) the ALTER verb in COBOL

COBOL, as many other high level languages, possesses a "goto" statement. However, it also allows what we may call a "variable destination goto" statement. The destination of a jump may be altered during the program execution. Thus the text of the program may be changed dynamically. It can no longer be read with ease by a human reader. The prospect of a program with many uncontrolled jumps whose destination is unknown, except during the actual execution of the program, makes one marvel at the debugging ability of those programmers who write such COBOL programs.

(d) The CASE statement

(see for example Algol W, PL360, XPL, Algol 68).

The case statement may be considered as a generalization of the alternative statement (if). We can describe its syntax by the following.

```
case  <integer expression>  of
     {<statement-1> ;
      <statement-2> ;
          .
          .
          .
      <statement-n>}
```

The value of the <integer expression> determines which, if any, of the n statements will be executed. The ordering of the individual statements is vital to the correct functioning of the whole statement. If one statement is omitted (a card is lost), or some get out of order (the cards are dropped), then the whole statement is liable to be erroneous. Yet it may still be meaningful to the reader and acceptable to the language processor. The solution to this difficulty is shown by Wirth (1971a) in the language PASCAL. Each of the n statements is given a label and the <integer expression> is replaced by an <expression> which will evaluate to one of the n labels.

example:

Suppose "pointer" is a variable of a certain type yielding the values described as "east", "west", "north" and "south". We may write:-

```
case pointer of
  east:   . . .
  west:   . . .
  north:  . . .
  south:  . . .
```

The ordering of the four possible statements is immaterial and a number of other checks are possible to prevent errors.

(e) Implicit declarations

Weinberg (1971) and Palme (1972) are among many who have written about the dangers of languages where declarations are made implicitly. "New" variables are liable to be introduced through misspelling of variable names without any indication of fault by the language processor. Explicit declarations are useful to a reader in that he is given a full description of what attributes he may assume for the individual variable names (see also Chapter 4 section 4.1).

There are many other instances of error-prone constructions being provided in programming languages (see for example Weinberg 1971). The general point which they illustrate is that it is extremely simple to

introduce complexity into a language design whereas the aim should be simplicity. We suggest, therefore, that language designers should pay a greater heed than is generally apparent to the fact that a programmer is fallible and finds complexity difficult to overcome. In the design of a program, the programmer is learning about his problem. If he can express himself clearly and easily, then his appreciation of his task is likely to grow. However, if he has to struggle with complex language constructions, then much of his effort will be diverted and he may miss opportunities in the discovery of acceptable solutions.

Chapter 3:

Structure in Representation and Method

In the previous chapter, a number of the requirements of man for tackling complex tasks were noted. In particular it was suggested that a methodical approach was essential and that there must be a means of representing and organizing the information concerned with the job being tackled. These needs are closely related. Method relies upon the availability of information, whilst any representation or organization of information will not be helpful if it obstructs the method. One of the most powerful ways of organizing information for describing complex systems is the hierarchy. Simon (1969) says:

> " . . . if there are any important systems in the world that are complex without being hierarchic, they may, to a considerable extent, escape our observation and understanding".

Further support is given by Whyte (1969):

> " . . . hierarchical classification is the most powerful method used by the human brain in ordering experience, observation, entities and information".

A recent paper (Belady and Lehman 1971) analyses the structure of programs from the point of view of its effect upon the economic lifetime of a program. (The economic lifetime of a program describes that period of time during which useful work can, with confidence, be achieved with that program. It ends when errors or malfunctions of the program occurring as a result of modifications or misconceptions incorporated earlier cannot be

removed without adding further errors which will themselves prevent useful work and which also cannot be removed). Amongst the conclusions reached in this paper is that the structure of a program should allow hierarchical representation.

Programming methods which accord with the philosophy of "divide and rule" can lead to programs which exhibit a hierarchic structure. An example of such methods has been given in Chapter 2 in terms of problem decomposition.

Simon (1969) gives a telling illustration of the power of hierarchic design and development methods. This illustration compares two approaches to the construction of a complex mechanism. The first approach, which we will describe as the "single-unit" approach represents a method which is not based upon hierarchic notions. The various primitive elements which form components of the total mechanism are assembled in no particular order and are not recognizable as being correctly in position until the last primitive element is assembled. The second approach is based upon the method of building recognizable sub-components which may themselves be used to form further recognizable sub-components until the total mechanism is constructed. In this second approach, the existence of completed sub-components represents the state of the construction activity at any given time. This information can be used with advantage during the construction activity and allows, for example, for the activity to be interrupted or for the course of the activity to be influenced. The "single-unit" approach offers none of these possibilities. Any interruption of the construction activity will, almost certainly, necessitate the complete recommencement of the task as no information is available to describe the current state of the activity.

This illustration can be translated into programming terms without losing any effect. If a program is constructed from components and sub-structures

which can be recognized as such because of the representation of the program, then the programmer is well placed to decide what he must do next and what relationship this has to his previous work. On the other hand, if a program is constructed without any definite method such that the properties of that program cannot (except in the most trivial cases) be appreciated, then the programmers task is hopeless.

Programming languages are, as discussed in Chapter 2, much too restrictive to allow the representation of components in a form sufficiently related to the problem to be useful in a general way.

example:

> Programming languages generally have a limited domain of data types or structures which they can express. Thus, in any representation of a program in a programming language, all objects manipulated by that program must be expressed in terms of these types or structures.

As we described in Chapter 2, programmers tend to use other notations to represent their program at various stages of its development. Thus natural language may be used to express an overview of a program which is presented to the computer in a programming language.

The various representations of a program can be structured hierarchically according to the forms of notation used. In this way the aggregation of properties given in one representation can be appreciated in terms of some other "higher level" representation. Other structurings might be applied but, following our comments above on hierachies, we wish to base our further discussion of programming upon methods and representations founded upon the ideas of problem decomposition and hierarchically ordered description.

In this chapter we will illustrate relationships between different

representations of a program by the device of a "level of description". A program may be represented at a "level of description" according to a set of concepts whose meaning is understood at that level. The same program may also be represented at another level of description by its expression in terms of other concepts understood at this second level. Various hierarchical relationships can be described which relate representations given at different levels.

Programming methods can be described following the notion of hierarchically organized representations. Various methods have been described (e.g. "top-down", "bottom-up"), all of which are based upon the philosophy of "divide and rule". The different methods are best characterized according to the ordering they suggest for the development of the program. We will describe several methods in terms of the representation scheme afforded by "levels of description" and discuss some particular issues concerning the practical application of programming methods using contemporary programming tools.

3.1 Levels of description

Consider the following pieces of text. Both describe a solution process for the same task.

Text A) "Read 10 input cards and, for each card, make a test to determine whether each of the first 9 values of that card is within acceptable limits and further, whether the 10th value is a valid check sum of the other 9 and is also within acceptable limits".

Text B)

```
integer array values (1:9);
integer check; integer i, j;
   for i: = 1 until 10 do
      begin for j: = 1 until 9 do
         begin read (values (j));
            if ¬ acceptable (values (j)) then writerror (1)
         end;
         read (check);
         if ¬ checked (values, check) then writerror (2);
         if ¬ acceptable (check) then writerror (3)
      end
```

Although both text A and text B represent (essentially) the same solution process for the same problem, the terms in which they are expressed are different. The difference is that each may be understood according to an interpretation attached to the particular set of concepts used. It is clear that the interpretation of the concepts used in text A is not dependent upon the interpretation of the concepts used in text B and vice versa. The reader may have been able to better understand B having read A because of the expressed relationship between text A and text B. However, B is understandable separately from A.

We will say that information may be represented at differing levels of description according to the set of concepts, and their associated interpretation, used in that representation. Information represented at a number of different levels of description may be related by an explanation of how concepts at one level of description can be expressed in terms of concepts of another level of description.

Woodger (1971) makes similar observations about "levels of language".

In particular he stresses that a language (a set of concepts together with an interpretation) should be capable of interpretation independently of any other level of language.

3.1.1 Characterization of a level of description

We characterize a level of description in terms of the primitive concepts which that level provides. These we describe in terms of four sets.

The first is the set of objects. It is sufficient to name only the type of objects which may be described rather than enumerating them individually. In examples and further discussion, this set will be denoted by D.

The second characterizing set is the set of operations which may be performed upon objects described by the set D. By an operation we mean to include not only operations in the normal sense, but also predicates and functions which take objects as operands. We do not regard identification as an operation. This set of operations will be denoted by F.

Operations may be combined by elements of the third characterizing set which we denote by C. The set C contains, therefore, those elements of a level of description describing permissible orderings of operations.

example:

> A particular level of description might be capable of expressing ordering in such terms as:
>
> "and", "then", "after".
>
> A level of description such as provided by a programming language contains terms like
>
> ";" , "if . . . then . . . else . . ."

Finally, objects may be grouped together in certain ways expressed in terms of data structuring primitives.

examples:

a "deck" or a "sequence"

We include in this final set (denoted by S), means of identifying elements of "data structures".

We will adopt the convention of subscripting the set identifiers D, F, C and S in order to distinguish levels of description. We now give two examples of different levels of description.

Example 1

The level of description provided by a simple, conventional programming language (which we call SPL) may be characterized as follows:-

$$D_{SPL} = \{\text{integers, booleans}\}$$

$$F_{SPL} = \{+, -, =, <, \&, |, :=\}$$

$$C_{SPL} = \{;\ ,\ \underline{\text{if}}\ldots\underline{\text{then}}\ldots\underline{\text{else}}\ldots,\ \underline{\text{while}}\ldots\underline{\text{do}}\ldots\}$$

$$S_{SPL} = \{\underline{\text{array}},\ \text{subscription}\}$$

Example 2

Consider the following problem described in natural language.

"A bunch of bananas is hanging just out of reach above a monkey. The monkey wants the bananas. Nearby there is a large box which the monkey can move and onto which the monkey may climb. How can the monkey reach the bananas?"

The solution to this problem is, of course, obvious assuming a reasonably intelligent monkey. The characterization of the level of description at which this solution could be given is:-

$$D_{MB} = \{\text{monkey, box, bananas}\}$$

$$F_{MB} = \{\text{(monkey) move (box),}$$

$$\text{(monkey) climb on (box),}$$

$$\text{(monkey) take (bananas)}\}$$

$$C_{MB} = \{\text{first, then}\}$$

$$S_{MB} = \{\text{bunch}\}$$

The characterization of a level of description as given above is not intended to be the basis for any rigorous treatment of language relationships. It is merely for the purpose of separating certain concepts which are frequently used in the expression of programs and which conveniently allow different descriptions of the same thing.

3.1.2 Related levels of description

If the interpretation of concepts of one level of description may be expressed in terms of the interpretation of concepts of a second level of description, then there exists a relationship between these levels of description.

Such a relationship may take the form of an explicit statement that the meaning of a particular concept (or set of concepts) at one level is equivalent to the meaning of an expression understood in terms of concepts of the second level.

example:

> Suppose that there is an operation " f " understood at level 1. Suppose some expression is given at level 2 whose meaning will be understood according to the concepts of that level. If this meaning is understood to be equivalent to the meaning of the operation " f", at level 1, then there is a relationship between levels 1 and 2.

Alternatively, such a relationship between two levels may exist because a description of a piece of information is given at both levels. The fact that it is the same information which is described implies that the interpretation of the concepts of one level can be expressed according to the interpretation of the concepts of the second level.

example:

> The two representations of the one program given in section 3.1 imply a relationship between the two levels of description used.

We will describe this relationship between two levels of description in terms of the notion of height. A level of description is said to be higher than another if the concepts of the first level are understood by expressions described using the concepts of the second level. It is not useful to define this notion more closely. In particular we do not wish to indicate whether or not the height relationship may be defined cyclically.

If there is some level of description which is considered never to be higher than any other level of description, then this level is known as the base level. It will normally be the level of description of the programming language.

It is intended that the measure of the height of one level with respect to another be connoted with the relative "closeness" of concepts of each level.

example:

> If one level of description contains the notion of "matrix" whilst another provides the concept of "array", then these levels can be described as being closer together than if the second level provided only the concept of a linear address space.

However, it must be noted that we do not attempt to give any quantification of height and further, that for any two levels which have an explicit height relationship, it is always possible to interpose a third level between them provided that we have a sufficiently inventive idea of what is meant by "concept".

Information (including the particular case of programs) may be represented at a number of different levels of description. These various representations can exhibit a hierarchic structure reflecting the actual relationships that exist amongst the set of levels of description. Simon (1969) describes hierarchic structures by the property of "near-decomposability". A set of variables representing certain information can be compounded into groups, each of which may be studied more or less independently of the interactions between the groups. Relations between the groups may themselves be studied more or less independently of their individual element-wise composition. Related levels of description can exhibit such a property according to the expression by which they are related.

An <u>abstraction</u> is a particular relationship between the representations of some information at two separate levels of description, such that the terms

of the lower level are used to express one concept of one of the characterizing sets of the higher level. Between any two levels there may, of course, be more than one abstraction.

We may identify four separate abstractions according to the particular set to which the concept in the higher level belongs.

(i) Representational abstraction (the set D).

(ii) Operational abstraction (the set F).

(iii) Sequential abstraction (the set C).

(iv) Structural abstraction (the set S).

Any of the elements of the lower level may, of course, be used to express any particular abstraction.

example:

> An object at one level may be "represented" by a particular set of operations. An abstraction from this set of operations may be considered as a member of the set D at a higher level and hence be a representational abstraction.

Abstraction represents the aggregation of properties and interactions of concepts from the lower level to be interpreted as a single concept at the higher level. The inverse of this process we call elaboration. Elaboration details an interpretation of an aggregate property in terms of properties and interactions of a set of concepts.

example:

> A program might be described at one level of description in terms of a "stack" using operations "pop an element" and "push an element". At a lower level, the notion of a stack might be elaborated in terms of an "array" and a "pointer" into the array to represent the top of the "stack". Elaborations would also be given for the "stack" operations as being operations upon arrays and pointers. The program described using a "stack" could equally be described in terms of these elaborations.

3.1.3 Density of a set of related levels of description

One of the major reasons for giving representations of a program at a number of different levels of description is that there should, as a result, be an increase in the comprehensibility of that program in terms of the relationships that exist between the concepts of the problem area and the primitives of the programming language. Whether or not this goal can be achieved depends considerably upon the ease with which the actual relationships existing between the various levels of description can be understood. Even if these relationships can be described according to abstraction, comprehension is not necessarily assured. This can be true if the process of understanding individual relationships between levels is very difficult. In this case any measure of the height of two related levels will be large and the number of different levels used will be small. Alternatively, it may be a relatively easy matter to understand the individual relationships between levels, but, because of the large number of such relationships, understanding the whole is difficult.

There is, in general, some point where the number of related levels is large enough such that it is possible to comprehend the form of the relationships existing between individual levels, but not so large that the number of relationships itself is a barrier to comprehension. This number will not be constant, even for a particular problem or a particular programmer. We will describe a set of levels of description which satisfy this necessarily vague criterion as being sufficiently dense. In any discussion which follows we will further assume that a sufficiently dense set of levels of description will be related by abstractions.

3.1.4 Levels of description and programming languages

The primitive concepts of a programming language form a level of description. In addition most programming languages provide well-defined mechanisms by which a programmer can give a representation of a program at levels above the base level of the language itself (e.g. procedures, data structures, macros).

A procedure is a method of aggregating the properties of operations combined in a certain way in order to provide a "higher level" operation. Procedures, therefore, provide a means of describing operational abstraction. The use of a procedure allows the programmer to abstract from the details of the expression describing how a certain operation is implemented to an understanding of effect denoted by the name of the procedure.

Data structuring facilities in a programming language can be used to abstract from a set of relationships amongst data to the notion of a structured object possessing certain properties. Hoare (1972a) stresses the importance of this role in describing and understanding programs and lists a comprehensive set of structures. Many of these are found in the language PASCAL (Wirth 1971a).

In most programming languages, however, there is only a limited provision for deriving a new level of description by representational abstraction. Algol 60, for example, allows arrays to appear as parameters to procedures, but does not allow an array to be used as a primitive in a further array. (Of course, multi-dimensional arrays may be used, but these do not express the appropriate conceptual properties of arrays of arrays).

Extensible languages provide more general facilities for the representation of programs at several levels of description. Algol 68 (van Wijngaarden 1969) allows the expression of both operational and

representational abstractions to provide concepts which may be used to represent a program.

example:

A level of description containing rational numbers may be described in Algol 68 by,

```
mode rational = struct (int numerator, denominator);
op n = (rational r) int : numerator of r;
op d = (rational r) int : denominator of r;
```

together with operations (for example)

```
op sign  = (rational r) int  : sign n r;
op whole = (rational r) bool : d r = 1;
```

(example taken from Lindsey and van der Meulen 1971)

SIMULA 67 (Dahl, Myhrhaug and Nygaard 1968) also provides similar facilities by the class concept.

example:

Rational numbers, as above, can be provided by:

```
class rational;
begin integer numerator, denominator;
  integer procedure sign; sign := if numerator<0 then -1 else 1;
    boolean procedure whole; whole:= denominator = 1;
                    .          .          .
end;
```

The extensible language ECL (Wegbreit 1971), in addition to providing means for both operational and representational abstraction, has a facility for sequential abstraction.

In order that a program expressed at a number of levels may be easily understood, it should be possible for these levels to be described as being sufficiently dense. Certain structuring primitives of programming languages can make this difficult if they are not used in restricted ways.

A pointer is often used to represent relationships amongst elements of data. In most programming languages where the pointer is available, there is little restriction upon the complexity of the relationships that can be so expressed. If the use of a pointer in a program describes relationships which are difficult to understand according to any abstraction, then it will not be possible to represent that program at levels of description which are sufficiently dense.

The goto statement has properties which are similar to those of the pointer except that it represents relationships which describe the flow of control in a program. It is possible to use the goto to describe relationships which are so complex as to preclude the representation of a program at a sufficiently dense set of levels of description.

Wulf and Shaw (1973) have described the global variable in a similar light.

Each of these constructions can, of course, be used and still allow a program to be represented at a set of levels of description which may be described as being sufficiently dense. However, it is necessary that some discipline of use be adopted. This introduces a dilemma for language design as to whether or not it would be better to omit such constructs. It is the author's opinion that it should not be left to the individual programmer to impart his own discipline, for who is he to judge what should form a sufficiently dense set of levels of description and what should not? It is part of human nature to be fascinated by ingenuity

to the detriment, in many cases, of clarity, simplicity and understanding. If programmers are given the freedom to hang themselves, then many of them will probably try.

The development of certain programming languages lends support to the idea of providing a reasonably powerful set of structuring primitives whilst imposing restrictions upon the programmer.

The language BLISS (Wulf, Russell and Habermann 1971), for example, does not have an explicit goto statement. Instead, specialist usages of the goto are retained in the form of exits from loops, blocks and procedures. The language developed as part of the SUE project (Clark and Horning 1971) includes mechanisms (e.g. CONTEXT, DATA and PROGRAM blocks) specifically designed to encourage the programmer to represent his program according to a hierarchical structure.

It is, however, probably true to say that there are many obstacles to be overcome and technical advances to be made before languages possessing such properties as mentioned earlier are widely accepted.

3.2 Methods for constructing programs

As Simon (1969) suggests, and as was described in the previous chapter, one of the most powerful ways of tackling a complex problem is to reduce it to a set of "smaller" (i.e. less complex) problems. Each of these problems may in turn be reduced to sets of smaller problems thereby developing a hierarchy of "problems". Those which are least complex will be found at the extreme points of this hierarchy (i.e. if the structure is thought of as a tree, then the leaves of this tree stand for those problems which are least complex). Eventually the division process ceases when a problem is so "simple" that its solution can be expressed with ease and confidence. The solution to the whole (original) problem may then be found by a composition process, the solutions to a set of

of sub-problems being composed to express a solution to the problem from which they were derived. Thus, the total solution may be expressed. However, a simple recognition of the power of problem decomposition is no more than a guide to how problems may be solved or how computer programs may be written. What is missing is a method, or way of proceeding.

Various programming methods have been described which are based upon this principle of decomposition. Terms such as "structured programming", "step-wise refinement", "top-down", and "bottom-up" have become increasingly familiar in the literature. According to each of these methods, programs are constructed in a piecemeal manner. Individual parts of a program are identified and constructed as separate activities, in a manner similar to the problem decomposition process described above. The various methods differ in the emphasis each places upon the separate tasks which together form the total programming activity. In particular, varying emphasis is placed upon the ordering of the development itself (see section 3.2.2. below).

The structuring of the program development process in these ways can be described, with advantage, in terms of levels of description, abstraction and elaboration. Indeed, many of the methods which are discussed in more detail in section 3.2.2., are based upon notions which are equivalent to the development of a program by its expression at a number of related levels of description.

3.2.1 Relationship with levels of description

The development of a program by methods based upon problem decomposition generates a certain structure amongst the information which describes such a development. This information and this structure may be represented using the notions of levels of description, abstraction

and elaboration. This is best illustrated by an example. Any ordering of the development process which is apparent in this example should, at this time, be taken as merely incidental.

Consider the following problem. (See also section 3.1).

> "Write a program which reads 10 input cards and tests these same 10 input cards for the following conditions. Each of the first 9 values on each card should be within certain limits. The 10th value should also be within these limits and, further, should be a check upon the preceding 9 values on that card".

The first stage in writing such a program is to analyse the problem statement to decide what major concepts require to be represented. Such an analysis might well suggest that this program could be written as a loop, with each pass of the loop first reading a single card and then testing this card to see whether it possesses the required properties. A program to do this can be represented as:

```
"Do the following 10 times:
    Read an input card and then test it".
```

This analysis decomposes the original "problem" into five problems. Pieces of program must be written to represent (a) looping ("do the following 10 times"), (b) carrying out a sequence of operations ("and then"), (c) reading an input card, (d) testing an input card and (e) storing information about an input card in order that, once read, it can be tested. These five concepts are just those concepts which characterize the level of description at which the program is represented above. If we denote

this by level "1", then:-

$$D_1 = \{\text{input card}\}$$

$$F_1 = \{\text{read (input card), test (input card)}\}$$

$$C_1 = \{\text{do . . . 10 times, and then}\}$$

$$S_1 = \{\}$$

As the next stage in developing the program one of these five concepts is chosen and analysed in order to decide how it may be decomposed.

Suppose that it is decided to develop further the operation "test (input card)" by separating the operation of actually checking the card from the operation of reporting whether or not a card is satisfactory. Thus "test (input card)" is decomposed into operations which we might call "check (input card)" and "report (result)". If the total program is to be expressed in terms of these concepts, then the level of description at which such an expression is given will contain "check (input card)" and "report (result)" as operational concepts. Notice, also, that a new concept has been introduced, that of "result". Some means of communication between the action of "check (input card)" and "report (result)" must be found. Thus, although the decomposition of a "problem" at one level of description may be carried out according to the properties required of that "problem" regard must be paid as to how that decomposition may be expressed in the context of lower level concepts. In this case, we expect that it will be an easy matter to implement the necessary communication and so decompose "test (input card)" as described. In general, however, it may not be possible to evaluate a decomposition of a problem with any

great confidence because of a lack of knowledge either of the properties required of the high-level "problem" or of the relationships that such a decomposition will require at some lower level.

If a level of description is characterized on the basis of the decomposition of "test (input card)" suggested above, then this level (denoted as level 2) is related to level 1. The operation "test (input card)" at level 1 is elaborated at level 2 by an expression involving operations "check (input card)" and "report (result)". This elaboration may be represented as a piece of program at level 2.

Suppose that the next "problem" chosen is that of deciding what information to retain about an "input card". An analysis of the properties of an "input card" and the requirements of the communication between "read (input card)" and "check (input card)" suggests that it is necessary to retain all 10 "values" of any input card. Each of these 10 values must be identifiable separately and in the proper order. Thus the problem of retaining an "input card" may be decomposed into the problems of retaining a "value" and of structuring several "values" into an ordered "sequence". Notice now that, if the total program was represented at a level of description reflecting this decomposition then it is not sufficient merely to incorporate an expression of an "input card" as being "a sequence of 10 values". In addition expressions are required which describe how the operations "read (input card)" and "check (input card)" are carried out in respect of the decision taken as to the representation of an input card. Thus, in order to give a meaningful representation of the program at this new level of description (denoted as level 3) decompositions must also be given for "read (input card)" and "check (input card)" in terms of, for instance, "read (value)" and "check (value)". In order to understand the program at

level 3, therefore, it is necessary to understand several individual elaborations, although each may be described hierarchically.

The development of the program may continue in a manner similar to that described above. A choice is made from amongst a set of possible "problems" that remain. An analysis of the properties required of the chosen problem suggests a decomposition of that problem into a set of "sub-problems". This decomposition forms the basis for a level of description at which the program (or a part of the program) may be expressed. However, this expression may require further concepts or decompositions before it can be understood to satisfy the properties required of the problem. Alternatively, the ordering of the separate tasks may be different as we discuss below. However, the notions of decomposition, expression and choice are relevant whatever ordering is followed.

3.2.2 A discussion of methods

The relationship between approaches to program construction based upon a decomposition of the overall task and the ideas of levels of description discussed in the previous section draws attention to a number of factors. The programmer must choose a particular "problem" to investigate further. When he has made a choice, he must decide on a suitable decomposition of that problem and how the piece of program for that problem will be expressed in terms of this decomposition. The influences upon his choice and his determination of a suitable decomposition and expression are, to a large extent, based on any actual method he may be following. A number of well known methods are discussed below. This discussion is itself based upon two observations concerning program construction. The first is that the order in which a program is developed plays a crucial part in the form it eventually takes. A simple example is only an illustration of this observation.

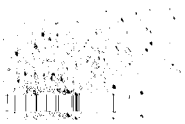
example:

> At an early stage in the development of a program it is realized that certain data must be retained and made available during subsequent processing. If a decision is taken at an early stage as to how this data is retained (i.e. according to a particular mapping between the abstract data structure and actual storage according to an expression in a programming language) then this decision determines to a considerable degree how operations upon this data are implemented. At the time the decision is taken the full extent of such operations will, most likely, be unknown. If the decision is delayed until as much information as possible is available about how the data will be used, then a more appropriate representation might be achieved.

The second observation concerns the evaluation of decisions and expressions made by the programmer. Although Chapter 4 is devoted to a consideration of program correctness and testing, the necessity of evaluating a program at various stages in its development has an extremely powerful effect upon the practical application of certain programming methods and therefore warrants comment at this time. If a particular method allows the programmer to obtain information about the worth of his work then this can act as a means of guiding his future work in particular directions. Mannheim (1966) describes a "method" for the design of highway routes which is based almost completely upon the idea of repeated evaluations. The method depends upon the designer providing "cost estimates" applicable to design choices at a particular level of description, and then uses Bayesian decision theory to suggest the cheapest route on the basis of these estimates. Although the actual mechanism of evaluation might not be practical in a programming situation there are techniques which have a similar background and which can be used. A number of these are

described below.

It is interesting, before considering programming methods in detail, to note the work of Alexander (1966). Alexander seeks to derive a method of design which we may describe as being determined from a direct consideration of properties of the problem. His technique is based upon the formation (by the designer) of a matrix of values to represent the relationships between all of those properties which are not acceptable in any solution to the problem under investigation. (In Alexander's particular case, he was interested in problems of environmental planning). Certain properties have a strong inter-relationship whilst being relatively independent of the remainder. Alexander proposes that the set of unacceptable properties may be grouped according to the strength of their mutual relationships. There will then be certain relationships existing between the groups themselves. These groups can therefore be aggregated into larger groups and the process repeated until all unacceptable properties are categorized into one single group. These various groupings form a hierarchical structure. The designer uses this structure and the properties of the individual groups to form his complete design for the solution to the problem. Thus the only problem facing the designer is the expression of this solution in appropriate terms. Whilst such an approach has a certain appeal, there are, however, a number of difficulties which restrict its applicability in a practical situation such as programming. Randell (1971) points out a number of these. In particular there is the problem of constructing the matrix of values relating unacceptable properties. This requires that the problem being tackled is well-specified and that the programmer is able to appreciate, more completely than is usual at the outset of any programming activity, the way in which the concepts of the



problem are related to the primitives of the programming language. Alexander's method appears to disallow, to a considerable extent, the freedom for a designer to reappraise his design on the basis of the way that design is developing and in the light of a better appreciation of the task with which he is confronted. This may be satisfactory in certain design situations where problems are well-specified and where there is no difficulty in representing the final design. However, these are two aspects of design in general which are not characteristic of programming. A programming method needs to allow the programmer the opportunity to learn about his task as he carries it out. Thus any ordering of the development cannot and should not be determined precisely at an early stage.

The programming methods which we now discuss rely on an ordering of the development of a program, but not one which has the inflexibility apparent in Alexander's method. Rather, they may be described generally as trying to balance the need for some ordering of the programmer's intellectual effort against the usefulness that information gained during the development process can have upon the way in which that development proceeds.

The essence of bottom-up programming is the construction of concepts, which are expected to be of use, from others which have less immediate attraction or applicability. The construction process is represented by a decision to provide a certain concept which will enable a representation to be given of a program (or piece of a program) in terms which are more closely related to the problem than are those of any available level of description. This decision is followed by an activity in which the appropriate elements of some already defined level of description (e.g. a programming

language) are combined in some way to represent the implementation of the new concept. This basic construction process is repeated, building further concepts in a hierarchical fashion until a set of concepts is constructed which is sufficient to allow the representation of the program for the overall problem at a level of description close to that at which the problem is described and understood.

A design ordering which is purely bottom-up is unlikely to be of any practical use because it takes no account of the posed problem to limit the space of concepts which are provided at each stage. However, it is more often the case that bottom-up programming forms part of a wider design method in which an initial design stage is carried out. This will take the form of a problem analysis process which decomposes the overall programming task into a hierarchy of sub-components. This hierarchy may then be implemented in a bottom-up manner to construct the total program. Methods similar to this have been used in programming a number of large systems (e.g. Scherr 1973).

As a program which is constructed in a bottom-up manner can always be represented at the level of a programming language, use can be made of the underlying hardware at any stage of the development for the purposes of evaluation and testing. It is possible to derive physical measures of resource utilization (e.g. execution time, storage requirements) during the development process and to demonstrate certain properties of pieces of program. It is, however, not possible to relate any individual measures to those which constrain the total program because this can only be achieved when the whole program is complete.

Bottom-up program construction is clearly exemplified in the description of the T.H.E. operating system given by Dijkstra (1968b). Each level of the design is built from the one beneath it, masking out

unwanted features and constructing others which are required.

Most contemporary programming languages, and particularly extensible languages encourage a bottom-up programming style by the provision of mechanisms such as procedures and data structures (see section 3.1.4 above) and compilers which enable programs to be tested on hardware. The use of separately compiled procedures is often helpful in testing programs at higher levels of description. Many of the publications concerned with SIMULA 67 include examples of bottom-up construction, (e.g. Dahl, Myhrhaug and Nygaard 1968, Dahl and Hoare 1972, Birtwistle 1973).

Amongst other reports exemplifying this approach is a paper by Naur (1969). This describes the idea of an "action cluster" whereby a representation is made for the innermost loops of a program before the remainder of the program is constructed.

It would appear that the construction of a program following bottom-up techniques is always likely to involve a compromise. The problem analysis phase cannot pay sufficient attention to the specific difficulties which will occur during the later implementation of the concepts specified during that phase. Thus problems will arise during implementation which would be best resolved by a further consideration of the overall design. It is often the case, however, that it is not possible to carry out the necessary redesign because of the effort which has already been invested. In this case, any implementation problems must be overcome in some unsatisfactory manner so that the original design is maintained. It may even be the case that it is not possible to meet the original design specifications but equally it is not possible to change these specifications. A program constructed under such conditions will not, therefore, be likely to meet its overall

design specifications.

Top-down programming is an ordering of the development of a program whereby the derivation of a suitable decomposition proceeds together with the determination and representation of an appropriate piece of program. Design commences with a description of the problem at some level of description. Using the concepts of this level a solution process may be described. These concepts are programming problems because they will not, in general, be directly representable in a programming language. The development of the program proceeds by considering these various problems in turn. Solutions for each may be expressed in terms of lower level concepts (which will not generally be those of a programming language) following an analysis and decomposition of the properties required.

example:

> A solution process may be described using the operation "test an input card". The problem of constructing a representation for the operation in terms of a programming language is tackled by analysing the required properties of the operation, decomposing it into the lower level operations "check an input card" and "report results" and giving an expression of how these operations may be combined to fulfill the action of the operation "test an input card" (i.e. "check an input card and then report results".)

The process continues until the representation of solutions to all problems can be given (by composition) in terms of the programming language.

Each decomposition is an invention of a new level of description enabling a description of the program (or part of the program) to be represented. Successive levels of description are related by elaboration until, finally, the "invented" level coincides exactly with the programming

language.

There have been numerous reports which discuss top-down program construction (e.g. Zurcher and Randell 1968, Mills 1971, Wirth 1971b, Baker 1972). Of particular interest is the report on "structured programming" (Dijkstra 1972a). This report introduces the concept of a "pearl" as a unit of program development. A pearl encapsulates many of the notions of a level of description together with the representation of elaborations of higher concepts.

The process of top-down programming differs from methods based upon a bottom-up ordering by the stress placed upon solving the problem of giving a representation to a program or piece of program. Just as a blind bottom-up design and encoding method is unhelpful because it takes no account of knowledge of the original problem, so a blind top-down approach is impractical because it cannot take account of the requirements of any actual programming language.

A particularly obvious manner by which the properties of the programming language can influence the development at higher levels is through notation. The programming language provides a level of description which may be characterized by sets D_{PL}, F_{PL}, C_{PL} and S_{PL}. Each level of description derived during the construction process may be characterized by sets as D_{LD}, F_{LD}, C_{LD} and S_{LD} (for instance). One approach is to restrict the relationships between the sets of level PL and those of various levels LD in certain ways. For example, the following relationships could be maintained.

$$C_{LD} = C_{PL}$$

$$S_{LD} = S_{PL}$$

$$D_{LD} \supset D_{PL}$$

$$F_{LD} \supset F_{PL}$$

By $D_{LD} \supset D_{PL}$ etc. we mean that the data concepts of the programming language are available at all levels of description LD, although other concepts of data may be present at levels other than that of the programming language. Other interpretations could be placed upon this relationship, either limiting or expanding the set of concepts available at various levels.

If, in addition, other characteristics of the programming language (i.e. its textual nature, its particular syntactic forms) are suitably generalized and applied to the notations used at higher levels, then the flavour of the base language will permeate the design process and encourage the program to be developed in a consistent manner towards a given programming language.

Baker (1972) describes a top-down approach based upon a similar scheme, with the further constraint that the mechanisms used to relate the various levels of description (i.e. the expressions of solutions at each level) should be those mechanisms of the programming language which structure concepts hierarchically. In his scheme, $D_{LD} = D_{PL}$, $C_{LD} = D_{PL}$ and $S_{LD} = S_{PL}$ for all LD and the base language is PL/1.

The more general scheme described above is recognizable as the generalized or "bastard" programming language often used by programmers during program development (see Chapter 2, section 2.3.3.).

Most programming languages can be used in a restricted manner to provide a number of levels of description derived top-down. The use of such notations purely as representational devices is almost neutral as to the ordering of the development (see section 3.1.4. above).

example:

> A program can be represented in a programming language as merely a sequence of calls to procedures which have not been developed. Mills (1971) and Baker (1972) use an approach similar to this (see below).

Design evaluation in a top-down method cannot rely upon knowledge of the eventual form of the program in a programming language until the program is almost complete. Thus the only measures of the "correctness" (or suitability) of a particular program development which can be determined in the early stages are relative to the programmer's intention for high-level concepts. Equally, no measures can be given of the utilization of actual hardware resources when the program is represented in terms of abstract concepts divorced from considerations of execution speeds or storage requirements.

However, though these observations are generally true of top-down development methods, it is possible to improve on this situation if certain restrictions are made. The method described by Mills (1971) and Baker (1972) is an example. The programmer is allowed only to represent his program at levels of description which are derivable within a given programming language. He may represent his program in terms of procedures which are not implemented, for example. Because the program is still represented in the programming language, it may be presented to a compiler and executed

with "dummy" procedure bodies providing suitable support for the yet to be designed procedures. Thus, a certain amount of program evaluation can be done with mechanical assistance.

More generally, it is possible to make use of simulation techniques to overcome problems of design evaluation in top-down developments. Simulation can be used to model the typical behaviour of processes without actually creating a representation for them. This possibility was recognized in papers by Parnas and Darringer (1967) and Zurcher and Randell (1968). In the latter case, the term "multi-level modelling" is introduced to describe the particular design method advocated. At any particular time during its development, a program may be represented in terms of concepts which are not those of the base programming language. Simulation techniques may be used to model these concepts and thereby allow useful design evaluation to be carried out.

According to the multi-level modelling design method (and also that described by Parnas and Darringer) such simulations form the basis for program development. Initially the highest level of design is simulated in order that it may be evaluated. The concepts simulated at this level are then implemented in terms of lower level concepts. These concepts are in turn simulated to provide a mechanism for evaluation. When this evaluation is completed, the cycle is repeated. Multi-level modelling has received further attention in papers by Aslanian and Bennett (1971) and Graham, Clancy and DeVaney (1973).

This discussion of programming methods has stressed particularly the role played by the ordering of the development activity. As both Gill (1969) and Naur (1972) point out, a strict adherence to either a top-down ...

or a bottom-up ordering is neither natural nor practical. As we described above, however, the separation of the task of analysing a problem from the task of embedding the appropriate concepts in a program can lead to programs which may not meet their specifications or which are unnecessarily complex. There would seem, therefore, to be an attraction in the parallel development of these tasks so that each may influence the other and allow a closer assimilation of the program text with its purpose. In order that this be possible without the need for constant redesign or reimplementation, we believe that it is necessary that programming methods be used which are based upon a top-down ordering. This is not generally the case at present. We suggest that this is primarily because the tools available to a programmer encourage him to encode his design in a programming language at a very early stage. The subject of later chapters is to describe certain programming aids which take an opposite point of view.

3.3 Conclusions

This chapter has been concerned largely with the way programs are developed. The basic premise was that program design and development is an extremely complex problem solving activity involving the representation of complex information in specialized notations. Our thesis has been that design must proceed in well-disciplined ways and that a hierarchical structuring of the representation of the program and of the development process were aims to be achieved. To these ends we introduced the notions of a level of description and of abstraction and elaboration relating such levels.

We believe that programming methods based upon a top-down ordering of the development have several advantages over other methods. This ordering combines both the derivation of suitable decompositions of a programming task and the expression of the program in terms of these decompositions within a single development structure. This allows full use to be made of information gained from such expressions in the evaluation of design decisions in order to influence future development. The use of simulation techniques enables useful information to be obtained about the properties of a program, even though this program may not be completely developed and represented in its final form in a programming language.

Chapter 4:

Correctness, debugging and other considerations

In the previous chapters, programming has been considered from the general standpoint as being a special form of problem solving activity. We have discussed many of the issues involved in the derivation of a program as a piece of text representing a set of computational processes from this point of view. However, little attention has been given to the problem of ascertaining whether a program will, in fact, fulfill the expectations of the programmer. We mentioned, briefly, some related ideas in discussing ways by which designs may be evaluated. In this chapter we describe some of the difficulties that have to be faced if a programmer wishes to be certain (or at least have a justifiably high degree of confidence) that a program is "correct". Often, as will be seen, a major problem is that of defining what is meant by "correct". We do not attempt to give a formal definition, but rather we discuss the specific difficulties inherent in describing, or even ascertaining, the relationship between a statement of a problem and a program written in response to that problem. We discuss various techniques whereby the programmer can demonstrate confidence in a program. These techniques include program proofs, constructive programming techniques, debugging and program testing and various other mechanical tools which are available. A major aim of these discussions is to draw attention to the influence that an overt concern for program correctness can have on the programming activity and to suggest the form of useful programming aids.

4.1 What is meant by correctness, and redundancy

It is very difficult to define precisely what is meant by the "correctness" of a computer program. We may sometimes say that a program is correct because we can "see" from its text that it obviously solves the given problem. This is equivalent to proving a theorem in mathematics by the axiom, "obvious", and has similar dangers. If we claim that we can see that a program solves some problem then we are making two very powerful assumptions. One is that we have completely understood the problem and the second is that we understand fully how the various programming constructions are related and represent a process to carry out the solution to the problem. In the previous chapters we have described some of the difficulties associated with such understandings. Except perhaps for the case of extremely simple programs solving trivial problems, the technique of "seeing" the correctness of a program is bound to be unsuccessful. In real world problems and programming situations, it is often the case that the problem is only fully appreciated by an attempt to write a program for it.

The correctness of a program is defined ultimately by whether or not the results of its execution are always those desired and expected. (Whether or not this includes all intermediate results is dependent upon the form of any actual definition of correctness which may be adopted). One way in which such a criterion may be checked is by running the program under all possible inputs and under all possible conditions. Even if we disallow the possibility of such things as asynchronous interrupts, then clearly it is likely to be necessary to run the program an extremely large number of times. Moreover, this approach becomes completely uneconomic when we realise that whenever a modification is made to the program, many of the previous tests have to be re-run. Well-structured programs can help reduce the number of

test-cases required (Dijkstra 1970), but that is all. Even the choice of the test-cases themselves may be an almost impossible task, the very complexity of a design making it difficult to ascertain whether or not certain program paths have been rigorously tested. Hetzel (1973) lists some approaches which have been followed in the field of automatic generation of test data. However, if a satisfactory "proof" of a program is required, then examining the executions it invokes is never likely to be a success.

> "Program testing can be used to show the presence of bugs, but never their absence".
>
> -Dijkstra (1970)

If we wish to ascertain absolutely that a program does what we believe it should, then we must rely on the program text alone. If it is possible to give a "proof" that the processes defined by a program will always produce an effect which can be recognized as being what is required, then we have indeed managed to provide some degree of confidence in the program. However, as we shall see it is by no means an easy matter to give such a "proof", and even then, the "proof" may be based upon a number of assumptions, some of whic are quite likely to be invalid. Thus there is likely to be a continued requirement for program testing techniques in order to improve program comprehension and increase confidence levels.

As is probably clear, establishing that a progam is correct is likely to require a considerable effort from the programmer. Much of this effort is expended in supplying redundant information which can act as checks within the program. In many current programming languages the programmer must provide information which is strictly redundant. The

declaration of variables as being of a particular type is an example. Checks can be made (e.g. type checks of operands and parameters) on the program text, which would not otherwise be possible, because the necessary information is available. Likewise, if it is desired to construct a proof of the correctness of a program from its text, then additional information must be supplied to specify the purpose of the program and against which the proof may be constructed. Program testing relies on the availability of redundant information. If this was not the case, then there would be no criteria by which to judge the results of such tests. We will give examples of such redundancy in the course of this chapter.

Of course, as human beings, we rely heavily on redundancy to allow us to achieve a better understanding of complexity. Many of the points we made in Chapters 2 and 3 concerning the design and representation of solutions are ultimately founded upon this idea. Hierarchical structures represent redundant information. The processes of abstraction and elaboration are exploitations of this fact.

Unfortunately, the provision of redundant information is not always acceptable to the programmer. If he is unable to see how he may gain from it or if it involves him in a considerable amount of additional work, then his natural inclination will be to refuse the task. For similar reasons, documentation is often badly done, or not done at all. The programmer himself considers he will get no benefit from it, or certainly that he will get no return worth the effort involved. However, if he can be given tangible benefits from such extra work in proportion to the work he expends, then he may be attracted. A reasonable aim, therefore, should be the provision of an environment in which a programmer is rewarded for his extra effort in supplying information in order that a higher degree of confidence can be placed in his programs. At the present time, the satisfacto

achievement of this aim would seem to be some time in the future. In the remainder of this chapter we investigate some of the questions which arise and how these questions are related to specific programming methods and tools.

4.2 The text of a program

In this section we will describe some approaches that have been made for the verification of the behaviour of a program by consideration of its text rather than from any properties which may be deduced from executing the program on a machine with particular test data.

4.2.1 The meaning of a program text

There are two obvious requirements to be met before we can prove the correctness of a program from its text. One is that there should be some means by which an exact understanding may be gained of what processes are represented and what are the effects of such processes. The other is that there should be some means for specifying those processes which the program text should represent (i.e. what is the intent of the programmer in writing the program). The latter requirement is dealt with in section 4.2.2.

A significant amount of work has been carried out attempting to define the meanings of the elements of programming languages and their combination into programs (see Steel 1966, de Bakker 1969 for example). Many workers have expressed the meaning of programs in terms of an interpretation on abstract, formal machines (e.g. van Wijngaarden 1966, McCarthy 1966, Lucas, Lauer and Stigleitner 1968). Such methods do not, in general, allow a single interpretation to be given for a program text which encompasses all processes which that text can represent. This is because it is necessary to specify an initial state of the abstract machine for any interpretation which thereby allows a meaning to be assigned to a program text only in the context of a particular set of input data.

Another approach to the derivation of the meaning of a program text is by the use of axioms and rules of inference. Hoare (1969) describes how axiomatic schema can be given which define primitive elements of programming languages by transformations of predicates over the variables of a program, and suggests a possible notation.

example:

the axiom of assignment
$\vdash Po \{ x: = f \} P$
where f is an expression,
x is a variable identifier,
and Po, P are predicates, Po being obtained from P by systematically replacing occurrences of x by f.

It is generally the case that the meaning derived for a program text by the use of such a scheme will be conditional upon a separate determination of the property of program termination. However, the fact that a meaning can be derived which is independent of particular values of input data (being expressed in terms of predicates relating the properties of the input and output variables of the program) gives the axiomatic approach a considerable attraction.

Hoare and Wirth (1972) give an axiomatic definition for a major part of the language PASCAL (Wirth 1971a). Dijkstra (1973) uses an axiomatic basis to define predicate transformations exhibiting certain properties in order to derive equivalent programming language primitives possessing similar properties.

4.2.2 Expressing the intention of a program

We may be able to derive a meaning for a program by consideration of its text by the approaches described in section 4.2.1. However, in order to ascertain whether or not this derived meaning satisfies the purpose for which the program was written, it is necessary to have a means by which the programmer can express his intent or understanding of his program. One obvious way in which this can be done is by the programmer stating that, whenever a process which is represented by the program, terminates, then certain values will have been produced. As a trivial extension to this idea, the programmer may express his understanding of parts of the program by statements which declare that, at particular points in any such process, certain intermediate values will have been produced. These various statements are known as assertions. The use of assertions in the proof of the correctness of a program was suggested independently by Naur (1966) and Floyd (1967a), although the idea of an assertion is much older and may be seen in writings from the early days of modern computer programming (Goldstine and von Neumann 1947, Turing 1949).

4.2.3 Proving a given program correct

Naur (1966) and Floyd (1967a) both propose that the use of assertions provides the basis for a technique by which programs may be "proved to be correct". The technique requires that at specific points within a program, the programmer makes assertions about the current values of the program variables. Each assertion is that, when the program execution reaches these points, then the named program variables will have the stated values. In particular, the assertion at the end of the program represents the expected result of the execution, whilst an assertion at the start of the program specifies the conditions under which the program will achieve this

desired result. Using a certain minimum set of such assertions and some suitable scheme for defining the semantics of the programming language, the program can then be checked statically to see if these assertions will actually hold. Notice that it is also necessary to prove that the program execution will actually reach those parts of the program annotated with assertions, in particular that it terminates.

There is, as we have stated, a minimum set of assertions required for this process to be carried out. It has been shown (King 1969) that it is sufficient if every loop of the program contains at least one assertion.

It should be noted that this technique may also be applied to programs which are interesting even though they loop (e.g. some processes in operating systems). The technique is used simply to demonstrate that, when a process reaches a certain point in the program describing it, then certain conditions apply with respect to the program variables. Terminating programs are merely a particularly interesting special case.

It is important to appreciate the role of redundancy in this technique, and also its fallibility. The provision of a sufficient set of assertions is no more than a second writing of the program. Indeed there is a strong requirement that the "assertive program" uses a similar notation to the program itself as they need to be checked against each other. As usually the same person who writes the program also supplies the assertions, there is ground for believing that any misconceptions he may have had when writing the program will also find their place as similar misconceptions in the assertions.

Another difficulty is the question of what to do when an inconsistency is discovered. Basically what the technique does is to compare two representations of the same object. When a mismatch occurs, all that may be concluded is that there is probably an error in one of the two versions.

Provided it is possible to decide in which version the error is, then progress may be made. It could be that the error occurs in an assertion. If so, then the programmer has an incomplete understanding of his program and his intention and so he should improve this understanding by trying to correct the assertion. If the error is in the program, then the programmer has also learnt something; namely that the program does not do what he thought it did, and again he has to discover what it does.

Finally there is one important point to be made about assertions. If a complete check is made between the program and the set of assertions, then this does not mean that the program is correct. All that may be properly claimed is that it is correct "relative to the assertions that were applied", and also "on the assumption that the model assumed for the programming language semantics was correct". The program might still fail to solve the problem.

The basic technique outlined above has been applied by several workers and considerable experience has been gained. London (1972) and Elspas, Levitt, Waldinger and Waksman (1972) both give lengthy surveys. The experience gained has not been confined solely to programs using only integers or other particularly well-understood concepts for which axioms can be derived without excessive difficulty. Hull, Enright and Sedgwick (1972) apply similar principles to the problems of the correctness of numerical algorithms. In addition Clint (1970) demonstrates that assertions can be used to prove properties about programs which use floating point arithmetic. Ashcroft and Manna (1971) and Lauer (1972) have recently investigated ways of extending Floyd's original approach to the problems of co-operating sequential processes.

A number of practical difficulties have arisen, not the least being the complexity of the proof of the theorems which arise. A theorem needs to

be proved for each path in the program which is bounded by assertions. These theorems are generated by "pushing" an assertion through the program, modifying it in accordance with the semantic definition of the programming constructions used, until another assertion is encountered. It is then necessary to prove a theorem concerning the compatibility of the modified assertion with the one encountered. Such theorems are generally known as verification conditions.

It is obvious that the form these verification conditions take is dependent upon the form of the assertions applied. Whilst it is unlikely that there will be much choice in the form of the assertions at the beginning and end of a program, the assertions made within the program are dependent upon the techniques employed in the program. Therefore, the programmer has some control over the complexity of the verification conditions by suitable choice of program and assertions. However, we believe that the programmer is unlikely to have sufficient understanding of his problem to use this as an absolute criterion governing the design of his program. It is still a useful exercise, however, to anticipate a requirement for a program proof during program construction. This is likely to have some affect upon a design.

Another difficulty that can arise is in the formation of the assertions to apply to a given program. This is particularly apparent in the case of assertions which are within a loop of the program. In some sense these assertions represent the meaning of the loop itself. It has been pointed out (King 1969, Good 1970) that there is an analogy between loop assertions and inductive hypotheses in mathematics. Elspas, Green, Levitt and Waldinger (1972) have suggested that difference equations might be used to establish loop assertions. Otherwise, it would appear that they must be supplied solely on the basis of a programmer's intuition. In view of the important roles

that the choice and form of assertions play in the generation of verification conditions, there is an argument for guidance being available to a programmer so that the assertions he makes are suitable.

Machine assistance may offer a solution to many of the problems described In theory it is possible to reduce the complexity of the verification conditions by supplying more assertions. This has, however, the effect of increasing the number of theorems to be proved. King (1969) describes a system which, given a program annotated with assertions, will generate the verification conditions and use an automatic theorem prover to prove their correctness. However, the program must be written in a special language which is, of necessity, limited. The basis of this limitation is the need to be able to describe the semantics of the language in a way that allows a theorem prover to be able to generate the necessary proofs. An interactive system is described by Deutsch (1973). Deutsch claims that this system is more powerful than King's, due largely to advances in the techniques of automatic theorem proving. He does, however, remark that the set of programs which can be automatically proved by his system also appears to be limited. Elspas, Levitt, Waldinger and Waksman (1972) describe many of the difficulties which have to be overcome in the design of a theorem prover suitable for proving the theorems which are generated in proofs of correctness of programs. It has been conjectured (Elspas, Green, Levitt and Waldinger (1972)) that it is unlikely that a resolution based theorem prover will ever be capable of proving such theorems. They suggest that a deductive theorem prover working interactively might offer the best approach.

Good (1970) describes a system which makes use of the human being to carry out the proofs whilst employing the computer in those places where it can be of great assistance at little cost. His scheme does not employ an automatic theorem prover and hence is capable of a wider application. There is no need to limit the form of the assertions to a particular system, and

in fact Good's scheme allows assertions to be written in a free form. The machine assistance provided is in the construction and simplification of the verification conditions and the maintenance of clerical information indicating which theorems have been proved (by the programmer) and which theorems still remain to be proved.

The "free form" assertion does have the advantage that it allows the programmer more scope in formulating assertions, but the danger is that there is then no restriction on its abuse.

Various other automatic systems have been proposed, several of which have been constructed (see London 1972) - systems which produce verification conditions alone are popular. We believe that such systems are useful in that the production of verification conditions provides an illustration of the complexity of a program and may suggest ways in which improvements may be made to the program or how more appropriate assertions may be provided.

4.2.4 Partial proofs and some effects of proof techniques

Although it may not be practical to prove the correctness of a complete program, the same techniques may be applied in order to prove (again in a relative sense) that a program satisfies some particular function. For example, it may be possible to prove that a program does do something, whether or not it is practical to prove that it does all that is required of it. We may call such a proof, a partial proof of correctness.

example:

> It may be critical that a real time system always produces, correctly, a certain set of values. The program may, in fact, produce other results but these are irrelevant if we can place no confidence in the values of the critical results. Proving that the program does generate these properly may be feasible (and thus desirable) whereas proving that the program generates all of its results may not.

Even if it is not feasible to prove the correctness of a program completely or even partially as suggested above, then it is very often

a useful exercise for the programmer to try to formulate assertions about his program, and possibly to construct proofs about properties of some of the more complex parts. In doing this, the programmer is forced to write down, in a semi-formal way, what he thinks his program does. It is the author's experience that such an exercise can lead to a better appreciation of the real essence of a difficulty, and thus to a more reliable program.

To illustrate, we give two examples, neither of which offers a formal proof, but rather some arguments taken from particular cases.

Example 1:

This example arises from a piece of program that was to assign to a variable "OK" the value true or false according to a set of conditions. The program was of the following form. We have added two assertions (P_1 and P_2) for later discussion.

assertion P_1 - - - - - - ->

```
OK := true ;
  if pf ≠ 0 then
   { if pd ≠ 0 then
      { . . . . . ;
          OK := (i ≠ op);
            while i ≠ 0 and OK do
              { . . . . . ;
                  OK := (i ≠ op) }
      }
   } ;
```

assertion P_2 - - - - - - ->

The variable "OK" is only set to false if a particular value (op) of i is found and both pf and pd are non zero.

The meaning of this piece of program can be defined in terms of two assertions P_1 and P_2. (The actual form of P_1 and P_2 is immaterial to this discussion). As such the complete piece of program is well-defined. However, the conditional statement

```
if pf ≠ 0 then . . .
```

does not have a meaning of its own. In order to give any meaning to it, it is necessary to include the preceding statement:

```
OK := true
```

As given above, it is not too difficult to prove that the program is consistent with appropriate assertions P_1 and P_2. However, in practice problems arose when the program was modified to cater for a wider class of possibilities. These modifications entailed additional statements and, unfortunately, these were added between the "OK := true" and the "if pf ≠ 0 then . . ." statements. Proving the modified piece of program was now much more difficult, as what had previously been a unit of meaning, was now separated into two sections. As a result an error was committed and was not uncovered by the informal attempt at proving the modifications. When, subsequently, the error was detected, the true meaning of the original statements was fully appreciated and was then better expressed in the following indivisible form:

assertion P_1 - - ->

```
if pf ≠ 0 then
 { if pd ≠ 0 then
    {. . . . . ;
       OK := (i ≠ op);
          while i ≠ 0 and OK do
             {. . . . . ;
                OK := (i ≠ op)}
    }
   else OK := true }
else OK := true
```

assertion P_2 - - ->

In this form, the meaning of the piece of program between assertions P_1 and P_2 is that of an indivisible unit. The moral of this example is that where a piece of program is complex, then it should

(a) be given a meaning by the use of assertions

and (b) be indivisible in a syntactic sense.

Example 2:

This example serves to discuss the relative merits of two common language constructions for expressing iteration, namely while . . . do . . . and repeat . . . until . . . The former allows zero or more iterations, whilst the latter will carry out the iteration at least once. Figures 4.1(a) and 4.1(b) describe these constructions in typical flowchart form. They have been annotated with some general assertions.

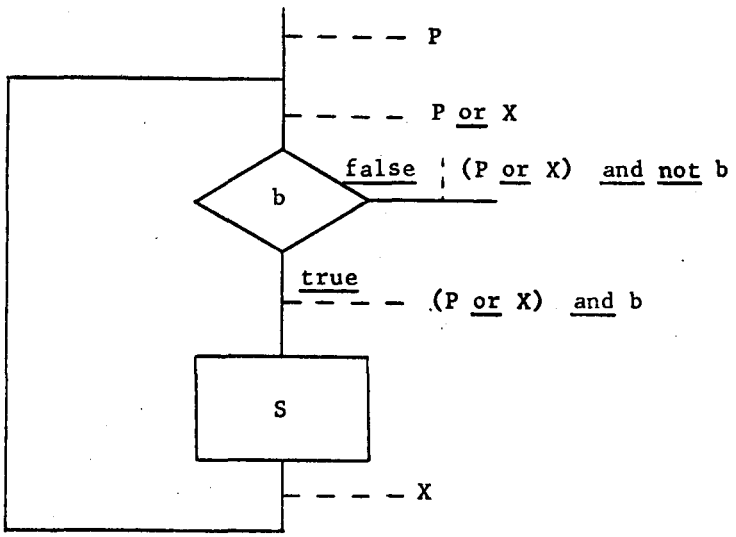


Figure 4.1(a) while b do S

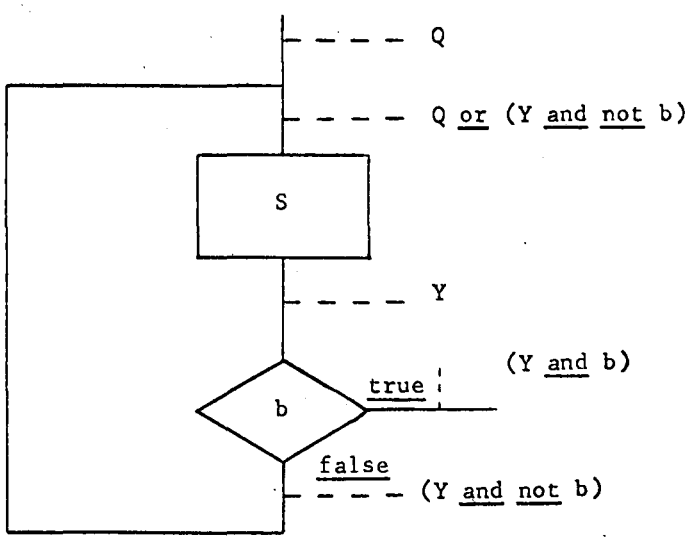


Figure 4.1(b) repeat S until b

The important assertions that describe the meaning of the iterations are those immediately preceding the body of the loop, (i.e. immediately preceding S) which we call the loop assertion and those expressing the result of the terminated iterations (at the exit from the loop).

In the case of while . . . do . . . the loop assertion always implies the truth of the condition governing the loop.

i.e. (P or X) and b $\Rightarrow$ b

This is not true for repeat . . . until. Only after one iteration has been carried out does the assertion need to contain any reference to the controlling condition.

i.e. Q or (Y and not b) $\not\Rightarrow$ not b

This special case treatment for the repeat . . . until iteration appears (certainly from the author's experience) to make it harder to specify the actual intention of the loop via the loop assertion. It is quite possible to execute the body of a repeat . . . until . . . iteration with the terminating condition already realised. As a result it can be more difficult to specify what the loop assertion should be, because this assertion does not necessarily follow from the condition controlling the loop.

These two examples represent abstractions from particular instances in the author's experience where a concern for a proof of a piece of program led to certain styles of programming. It would be difficult to give actual cases in detail because of their complexity. Equally, simple contrived examples do not suffice because their very simplicity tends to hide the problems they are supposed to illustrate. Thus, an appeal is made to the reader to relate these abstract examples to his own experience.

4.2.5 Constructive use of assertions

We introduced the notion of assertions from the point of view of giving a method by which the programmer's understanding of a program could be expressed. The flavour that we hope has been imparted is one of

"write your program, then prove it is correct".

If care is not taken it is likely that the principles of proving correctness will be divorced from the major problems of program construction. There does not seem to be anything particularly sensible in designing and writing a large piece of software, and only then proving (or, more disastrously disproving) its correctness. As Dijkstra has said (Dijkstra 1968a, 1972a), what we should really strive for is a way of maintaining correctness rather than of obtaining it. A concern for a later proof may have an effect on the way a program is written, but this is different from constructing a program and proving that this construction process is correct.

We may make use of assertions as a means of expressing an intent. Indeed, we have already noted that it is possible to view a set of assertions as a program. While we do not agree that this is a particularly appropriate

result in theorems that cannot be proved, in a program construction situation it may simply result in a program which is significantly different from that which was intended.

example:

> Hoare (1971a) describes the construction of a program called FIND. Hoare "proves the correctness" of the derivation of this program without making use of one particular assertion. This specifies that the vector constructed by the program must be a permutation of the input vector. In the derivation, Hoare uses his own knowledge of this fact without explicitly writing it down. If this assertion was not included in the set of assertions describing what such a program should do, then the program derived by, for instance, an automatic program synthesizer might well be different to that anticipated. It would, of course, be perfectly correct with respect to the information given by the programmer.

What is missing from the use of assertions in this manner is the necessary redundancy of information which enables checks to be made regarding the properties of the program.

Assertional methods can, however, play an important role during the development of a program. This role is particularly related to the approaches to program design discussed in Chapter 3. Recall that assertions allow an expression of the programmer's intent. Used as such assertions can represent a decision to develop a program along particular lines without the actual program being written. From such assertions the programmer may be able to evaluate different possible decisions. Having made some particular decision, as represented by some set of assertions, he may then proceed to construct a piece of program which he can prove will satisfy his intent as expressed by this same

set of assertions. Such techniques are to be seen in a number of papers, particularly in the "action clusters" of Naur (1969), in programs developed by Dijkstra (1968a, 1972a) and by Wirth (1971b), and in the techniques described by Mills (1971) and Baker (1972). In all of the cited references, the "statements of intent" are given in an informal manner. The justification of their correctness and the proof that the piece of program satisfies the intention are often given as a discursive argument embedded in the program design documentation. Hoare (1971a) and Allen and Jones (1973) give examples of a similar nature except that the statements of intent are given in a formal notation (e.g. predicate calculus, set theory) which allow rigorous proofs to be made. Indeed, in the case of Allen and Jones, the whole development process is carried out in such a system. An actual programming language is only used to represent an algorithm which has been otherwise completely developed.

Hoare introduces the idea of an "invariant" into the process of program development. His technique is describable in terms of levels of description. At a particular level of description his program makes use of certain properties of concepts from that level of description. (e.g. properties of a data type, or properties of a control structure). These properties are described in terms of invariants at that level of description. When, at a lower level of description, an elaboration is given for a concept so described, it is a necessary part of the process of justifying the correctness of this elaboration that these invariant properties are proved to be maintained.

Unfortunately, there is one non-trivial drawback to a more formal development of programs. This is that there is a not inconsiderable dependency upon the programmer's ability to prove the

theorems and lemmas which occur in the justification process. It is perhaps useful to illustrate this point by comparing the developments of the algorithm FIND given by Allen and Jones (1973) and by Hoare (1971a).

A program has to be written whose purpose (Hoare 1961) is to find the element of an array A[1:N] whose value is fth in order of magnitude, and to rearrange the array such that this element is placed in A[f] and further, that all elements with subscripts lower than f have lesser values than this element and all elements with subscripts greater than f have greater values than this element.

In both cases, the program which is evolved represents about 30 lines of a high-level programming language. However, in Hoare's development 18 separate lemmas must be proved. Allen and Jones require the proof of some 16 theorems and a number of lemmas in a development which is described in approximately 40 pages of manuscript. Of course, in both cases a number of proofs are trivial, but some are not. It is apparent that the form these theorems take depends upon the development process chosen. Man-machine systems may be an answer for the trivial proofs, but whether we are prepared to allow ease of proving theorems to have a considerable effect upon the actual development of a program is a debatable question. If possible we would expect that the proof of the necessary theorems was something that could be left on one side during each stage of the development, to be taken up as and when the programmer felt that formal justification was necessary.

It is, however, the author's belief that a suitable grafting of some of the above ideas for expressing intention and criteria relating to the correctness of programs onto the design methods described in Chapter 3 is likely to be of significant worth. The difficulty lies in deciding how much of such a facility should be provided, and the form it might take.

In Chapters 5 and 6 we describe one possible approach.

4.3 Information from Program Execution

The traditionally accepted methods of evaluating a program are based upon exercising it under a set of data values known as test cases. Although such exercises cannot hope to be exhaustive, it is possible to use these techniques to the point where a high degree of confidence in the behaviour of a program can be gained. If this was not the case, then the rapid growth in the use of computers that has occurred over the last twenty years would not have been possible. It is probably true to say that at this time at least 99% of programs being written will be evaluated by the use of techniques based on test case execution. We should, therefore, investigate some of these techniques, their limitations and their influence on programming methods.

4.3.1 Writing programs to be tested

It is important that any information generated by the execution of a program can be easily related to the actual text of that program. A single program text will, in general, map onto a number of different computational processes dependent upon the input data. However, even with a knowledge of the input data, the mapping from a given process onto the describing program text is generally ill-defined.

example:

> When, for example, a FORTRAN program fails, in many systems it is a difficult task to find out where this failure occurred.

Even if it is possible to relate information from a process to a particular line of program text, this is likely to be insufficient. Several events in a process will often be related to the same line in a program.

example:

> A statement in the body of a loop will be "used" in the process several times.

In order to be helpful it is necessary to identify an event in the execution of a program uniquely. In order to do this more information is needed (e.g. trace information). It will be obvious that to relate information about events in a process to the program text in a manner which is useful to the programmer, then it is necessary to start from a point where the exact relationship between text and process is known. In general, this point will be the beginning of the program. This says, therefore, that the programmer can only relate an arbitrary event in the execution of a program to the program text by knowing the sequence of events that have occurred since execution commenced. This is, in general, unacceptable because of the sheer amount of information that this represents. Dijkstra (1968c) has suggested that, if the program is structured in a particular way, then the necessary information could be maintained by use of a simple stack. The particular structuring is consistent with our earlier discussions of well-structured program design, in that it is necessary that the relationships that may be exhibited amongst the control structures of the program must follow a hierarchical discipline. Much has been made of the fact that this requirement does not allow the uncontrolled use of 'goto' statements (e.g. Rice 1968). What we feel is important to stress is that the relationships between the program text and the computational processes it represents are particularly important ones from the point of view of the comprehension of the program by a human being. The way in which Dijkstra demonstrates how

such relationships could be obscured by the use of uncontrolled jumps, serves as an illustration of several of the points we made about hierarchies and relationships in Chapter 3.

Consideration for relating run-time information to the program text has led to other programming methods. Amongst these the simplest may be described as "defensive" programming. Additional tests on the values of program variables are placed in the program to give a close relationship between text and process. It may be that a rigorous examination of the program text would reveal that such tests will always be satisfied. However, the programmer may have neither the desire, nor even, in general, the ability, to carry out this rigorous check. The simple expedient of inserting a test ensures that when the program is run, the knowledge that it has passed (or failed) the test should be available, whereas without the test this knowledge is less likely to be easily obtainable.

The sheer size and complexity of large programs has led to such notions as modular programming (see ICL (1971) for example). By decomposing a program into separable units, each of which may, initially, be partially tested in isolation, a higher degree of confidence can be placed in the total program. Of course, the choice of a particular modularization may not be made solely on the grounds of ease of testing. The fact that a set of modules may have been well-tested individually does not guarantee that they will work together as a group. However, we believe that, if, during the design of a program, due consideration is paid to the requirements of program testing, then modular techniques can be of some help in increasing the reliability of that program.

As a general philosophy, it is probably useful to appreciate, at the time a piece of program is designed and written, when tests will

be necessary to exercise it. As a program is designed, various decisions are taken. The testing of the program is an aid to ensuring that these decisions are actually reflected in the program code written down. The obvious time to design those tests which pertain to a particular piece of program is when the decision is made and the piece of program written. There is, of course, an even greater attraction in carrying out these tests then as well, but this may not be generally possible.

4.3.2 The information fed back to the programmer

There are essentially two sources from which the programmer can expect information about the progress of the execution of his program. One is from explicit statements in the program itself. At selected points in the program, the programmer may insert statements which will print out information such as the fact that execution actually reached this statement or a display of the contents of selected program variables.

Such a method can be attractive if it is relatively easy for the programmer to insert these statements without making alterations to the program under investigation. They must usually be removed once the programmer is satisfied with the way the program behaves during execution. (This, in itself can sometimes be a source of errors. It is not unknown for simple testing statements to mask out bugs which then appear when the statements are removed). A number of high-level languages cater specifically for these methods with special language forms (e.g. the AT statement of FORTRAN, dynamic tracing facilities using subroutine calls, programmer controlled exception handling in PL/1). By using such facilities the programmer may include testing statements which can be invoked by suitable input data. The statements do not, therefore, have to be removed. (Another example is described by Satterthwaite (1972). We will say more of this below).

The other source of information is the machine which is executing the program. In figure 4.2 we extend the figure of Chapter 2 (figure 2.1) to include the transfer of information from an executing program to the programmer.

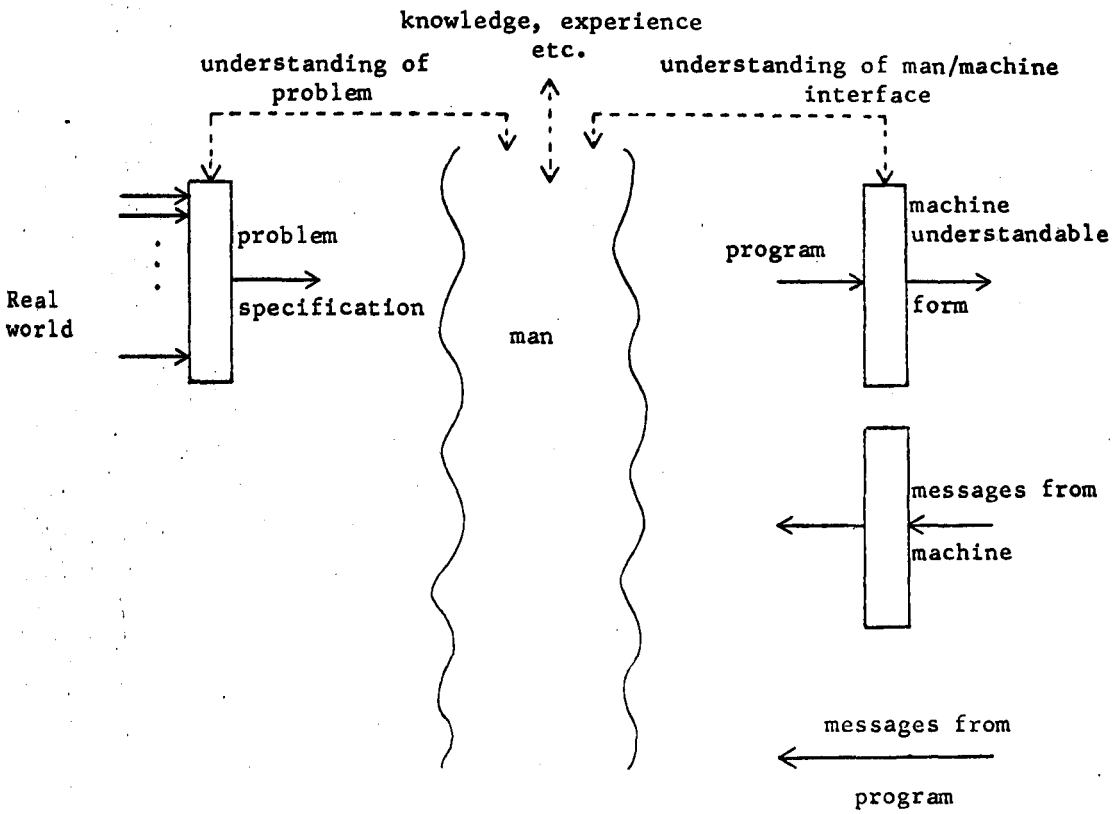


Figure 4.2

What we believe is a characteristic of many contemporary programming systems is that the man/machine interface is divided into at least two parts. One is the interface which accepts information from the programmer and transmits it to the machine. The other is the interface which accepts messages from the machine (in a form appropriate to it) and transmits these to the programmer. All too often it would seem

that this interface serves only as a relay station doing little to interpret the messages to the programmer's view of his program written in a particular programming language. Barron (1971) gives several examples.

A further reference to figure 4.2 may be helpful to explain the problems to be overcome. When a program is written in a particular language (distinct from the order code of the machine), there is some mechanism which physically represents the interface between man and machine. If this mechanism is a translator then the actual executing machine is conceptually separate from the interface. The original program is translated from its form in the programming language to a form understandable to the executing machine. Thus the executing machine has no knowledge itself of the original form of the program. It, therefore, cannot phrase messages to the programmer in terms of the original programming language. If a suitable interface is created to intercept these messages and make use of the original translator then it is possible to translate messages from the machine back into a form related to the original program (i.e. "source-language debugging"). This mechanism can be seen in the Alcor Illinois 7090/7094 post mortem dump system (Bayer, Gries, Paul and Wiehle 1967) and other debugging systems (see for example Evans and Darley 1966, Balzer 1969, Satterthwaite 1972).

Of course, if the man/machine interface is very closely tied to the actual execution machine (as for example, in the case of a software interpreter), then it is an easier matter to relate information about a program execution to the original source language form.

It is also important to consider what information should be made available to the programmer and when. It is obvious that one time when information is required is when the executing machine is asked to perform some function which it cannot do.

It is hoped that the executing machine will at least report this fact to the programmer. However, it is useful if the machine gives a little more information regarding the possible cause of the error and the current state of the execution process. Most contemporary programming and computer systems provide some such feature, though those which relate this information to the source language form of the program are less numerous.

There are other occasions when it is useful to supply the programmer with information. The tracing facilities of several high-level languages (COBOL, FORTRAN, PL/1) are examples.

It is also useful for the programmer to be given some statistics regarding salient features of a program execution. The evaluation of a program design is not simply a case of finding as many "bugs" as possible. Satterthwaite (1972) describes a system which generates a "profile" of a program execution in terms of frequency counts of the executions of various portions of the program. In Satterthwaite's system this information is neatly related back to the original program text, thus enabling the programmer to see where the bulk of the work is being performed. He can then pinpoint areas where it would be useful to improve the design. In such ways, program evaluation may be extended beyond the realms of being merely "correct" to allow comparisons between different versions of a "correct" program.

We will return briefly to some other aspects of debugging systems in section 4.4.

4.3.3 Program Testing as part of program design

Whilst a programmer cannot hope to test a program completely through observation of its behaviour under all conditions, observation of its

behaviour in particular cases can be instructive. Very few people are willing to accept their comprehension of a program purely from its text because of the immense intellectual effort required to appreciate the effect of the processes the program describes. Few people, therefore, will have complete confidence in their program without testing it. We believe that it is unlikely that the techniques of static program proofs will ever completely remove the need for test runs and evaluations. There is, therefore, a place to be found for tools which improve the information given to a programmer when a program is being tested. There is, indeed, a place for such facilities throughout the design process. The problems that are to be faced in appreciating a program from its text alone are equally likely to be encountered at any time in the design process. Thus any assistance which a programmer can obtain from experimental evaluations of partial designs will be invaluable. He is then able to obtain information about his design in terms of the process he is describing at the current level of description. Experiments can be made in "real" situations and designs may be tested as they are formulated rather than when they are ultimately realised in a conventional programming language. It may even be possible to make observations pertaining to program efficiency if the tools are sufficiently powerful. As we described in Chapter 3 (section 3.2.2.), in a design methodology based upon levels of description the concepts of multi-level modelling (Zurcher and Randell 1968) have an obvious application.

4.4 Some further machine aids and influences

In this section we will look briefly at a few other machine-based tools which can help the programmer in the construction of a program and which may enable him to have more confidence in his work.

4.4.1 Interactive systems

A feature of the recent growth of time-shared terminal systems has been the rise in popularity of languages and other facilities which make specific use of the fact that a human being is physically in communication with a program during its execution. Such systems range from interactive debugging schemes to complete programming systems such as BASIC and APL.

One of the particular characteristics of such interactive systems is the ability of the programmer to continuously monitor the execution of a program. It therefore becomes even more essential that the form of communication between the programmer and the machine is easily related to the program text. In online debugging systems for programs written in languages which are not specifically classified as "interactive languages" there is often a question of efficiency to be taken into account (see Balzer 1969 for some further discussion on this point). Approaches akin to the scheme described by Satterthwaite (1972) whereby use is made of efficient machine code wherever possible with source language interpreters being invoked if needed, would seem to have some attraction. Mitchell (1970) describes a system based upon the technique of incremental compilation which is similar.

Generally, interactive programming systems (e.g. BASIC, APL) make use of an interpreter for program executions. Such a system is therefore able to maintain overall control of program executions and communicate with the user about such executions in terms of the source program. By choosing to use such a system, a programmer deliberately sacrifices some of the power (e.g. execution speed, storage and input/output facilities) of the underlying hardware which could otherwise be obtained through a more conventional programming language system. However, in many circumstances, this sacrifice is more than outweighed by the benefits to be gained through

interactions between programmer and program executions.

The PILOT system (Teitelman 1970) was designed to allow particularly close co-operation between the user and his programs written in LISP. The user can direct PILOT as to what actions to take when error conditions arise (e.g. a spelling corrector). He is able to interact with PILOT as part of any error correction activity that may be undertaken. Other facilities are available which allow the programmer to give directions about the operation of his program.

It is the author's belief that, although the simple fact of having the programmer so closely involved with his program will not in itself guarantee better programs, it can help because of the increased understanding that is likely to accrue.

In order that such benefits may be achieved, an interactive system must possess certain properties. These we may classify generally under the heading of human engineering. Potential human users must not be distracted from obtaining the benefits of machine assistance because it is awkward. The well thought out design of the notation used in APL is a good example. This notation is extremely easy to use following some experience, and concise enough to be attractive for a human being at a typewriter terminal. Whilst it may have some drawbacks from the point of view of representing solution processes, as a means of immediate man to machine communication it can have few peers. The human engineering aspects of interactive systems are not specific to interactive programming systems. As Engelman (1968) points out, computers are very good at doing certain things which human beings find difficult. This ability is heightened in an interactive environment if the computer can do what is required just when it is required and particularly if such use is convenient.

Human engineering has received considerable attention in several . .

man-machine systems (e.g. that described by Engelbart and English (1968), MATHLAB (Engelman 1968), Hansen (1971a, 1971b), Mitchell (1970)). The interested reader is referred particularly to Hansen (1971b), or to Mitchell (1970) which is more relevant to the design of interactive programming systems.

4.4.2 Generation of syntactically correct programs

Hansen (1971a) describes how a programmer can be guided to construct only programs which are at least syntactically correct. He demonstrates how a text handling system based on hierarchical relations between pieces of text can be tailored to accept only text satisfying certain predefined rules. In particular he uses the production rules of PL/1.

The rules are applied in a constructive manner. The system (called EMILY) displays the current text (or portion of it) and advises the programmer which syntactic form of text string he may use to replace a non-terminal symbol present in the text displayed. The text which is constructed is certain to satisfy the syntax of the programming language, although logical errors may be present.

4.4.3 Program skeletons

Systems have been described (for example Bequaert 1968, Dutton and Minto 1971) in which programs are written by adding code to pieces of pre-written code called program skeletons. These program skeletons carry out various commonplace processing functions which are not specific to any particular application. This differs from the normal practice of programming such functions individually as required.

example:

> Many data processing systems require functions for data input, data updating and data retrieval.

It is possible that a programmer, by incorporating code which is already written in a general fashion and specializing it to suit his own requirements will produce more reliable programs in a shorter time. He will, for example, be able to concentrate more fully upon those design points which are relevant to the job in hand.

The method whereby the program skeletons are actually used in the construction of a program varies. In the system described by Dutton and Minto (1971), the skeletons are written in COBOL and each skeleton has exit points where the programmer can supply further statements (also in COBOL) which are specific to his purpose. Bequaert (1968) describes how the program skeletons can be specialized for particular applications on the basis of the programmer's response to questions generated by the system.

The way in which programs are developed using such systems is obviously dependent upon the availability and form of skeletons. These factors will exert an influence over the actual design of programs in a way which is similar to that exerted by a programming language. In applications areas where there are likely to be many programs requiring similar functions these systems should prove to be of some worth. There still remains, of course, the task of designing the total program and of constructing the necessary code to interface in a suitable manner with the skeletons.

4.4.4 Automatic error correction by a translator

Language processors are generally unconstructive when they detect errors in programs submitted to them. Usually an error message is supplied which gives the context of the error and some indication of what specific error has been found. It is rare that any action is taken to suggest how the error might be removed. The PL/C system (see, for example, Conway and Gries 1973) constructed at Cornell University, however, goes one stage

further than this by attempting to automatically "correct" program errors discovered at compilation. Whilst a large number of punctuation errors can be corrected with confidence, the correction of many other syntactic and semantic errors is unlikely to recreate the programmer's intention. It is claimed by the system designers that even in these latter cases, the effect is to allow the provision of further diagnostic information which will increase the programmer's chances of removing errors from his program.

A danger of the approach of automatic error correction would appear to be the likely encouragement of sloppy habits in a programmer. He will omit semi-colons because he believes that the system will insert them in the right places. Of course, system corrections should be checked by the programmer because no guarantee can be given that all such corrections maintain the original intention. The author conjectures that, unless there is some explicit mechanism to motivate a programmer to check all corrections carefully, then a number of erroneous "corrections" will not be appreciated as such. Even if the proportion of such misconceptions is small, it is surely worthwhile to demand some additional work on behalf of the programmer to make it more likely that he appreciates exactly what processes are represented by the program he has written. Unfortunately, it might be difficult to design a mechanism which would provide the desired effect.

4.5 Summary: Towards a Program Building System

The major question we have discussed in this chapter has been that of establishing the correctness of computer programs. We introduced this in terms of a requirement to increase a programmer's confidence in the worth of a particular program, or piece of program, as a solution to some problem. This involves the programmer in the comprehension of what he has written down (the text of his program) as a specification of a computational process.

We have discussed how it is possible, using the program text and suitable information regarding the meaning of programming language constructions, to obtain a high degree of confidence in a program. Many of the techniques employed in this appreciation centre around the provision of redundant information in the form of assertions, declarations etc. We also described how such information may be used in a constructive manner thereby ensuring a high degree of confidence in a program arising from the methods used in its construction.

In a similar way we have seen how we may improve the understanding we have of a program by observing its execution. Whilst this method cannot hope to give complete certainty as to how a program will behave it is possible to use testing criteria to aid in the process of program development.

On a number of occasions in the course of the above discussions we encountered situations where the co-operation of man and machine was likely to be useful. As examples we cite program proving systems (Good 1970, Elspas, Green, Levitt and Waldinger 1972), interactive debugging systems (e.g. Balzer 1969), interactive programming systems (e.g. APL, BASIC), interactive program construction (Hansen 1971a) or other non-programming endeavours (Engelman 1968, Engelbart and English 1968).

A natural successor to these schemes would be a single system concerned with providing a set of computer-aided tools to help a programmer in the development of a program. In addition to some of the particular techniques we have described, such a system would provide clerical aids organizing the information of the design development for the programmer. It would also impart the necessary discipline upon the programmer so as to affect

the overall structure of the final program.

The remainder of this thesis describes one particular such interactive "program building system" which the author has designed and implemented.

This system, in fact, concentrates primarily on providing facilities for program design rather than on testing or proving completed programs. This emphasis is only as a result of the particular emphasis it was thought desirable to demonstrate in an actual implementation. Thus, for example, whilst the evaluation of actual programs by execution on test data can be a powerful technique, it has already received a significant amount of attention elsewhere. It was thoughtmore appropriate to concentrate attention on those facilities which could guide the programmer in the development of a well-considered program, reflecting the care taken in its design, and exhibiting a good, elegant and appropriate structure. Because of the experimental nature of the implemented system, there are, of course, a number of deficiences and limitations. In the description which follows, these will be explained and their remedies, where appropriate, described. The actual system can act, therefore as a study of the feasibility of some of the ideas that have been discussed in Chapters 2-4.

Chapter 5:

Basic construction of programs using Pearl

In this chapter we will describe the basic ideas behind the Pearl (Program Elaboration and Refinement Language) program building system. This will entail a demonstration of how a program design may be built up into a complete program using the concepts of levels of description and of a particular design strategy. (see Chapter 3). In Chapter 6 we describe other features provided by the system in the form of machine assistance in the maintenance of the design and in its evaluation. In both Chapter 5 and Chapter 6 we will incorporate discussion on particular points as appropriate. More general discussion concerned with experience gained from using the system will be found in Chapter 7. In no sense will these chapters attempt to be definitive. Appendix B contains a summary of system facilities whilst appendix A gives a formal description of the syntax of the notation used to represent designs. Appendix C gives a few notes on system implementation. A number of examples are given in the following chapters; complete texts from which these were drawn may be found in appendices D, E and F.

5.1 Bases

The Pearl system acts as a specialized management system for a particular set of information; namely a program design. The system accepts texts in a particular notation representing parts of a program design. Each new piece of information is first checked in a number of ways before being incorporated into the total design. This ensures that the new information is itself reasonable and that it is consistent with the design already present. A number of conventional data base facilities are provided in Pearl to allow access and manipulation of design information (see Chapter 6).

The basic notion behind the system is that of describing processes to solve sub-problems in terms of varying levels of description. A program at a particular level of description is thought of as representing the action to be invoked on a hypothetical "machine" possessing attributes which characterize that level of description. The word "machine", although not intended to have all of its more generally assumed connotations, is chosen advisedly. In the current context a machine is an abstract entity capable of performing some action described by a program indicating the sequence of operations that define that action. A machine is considered to possess (or to understand) certain attributes and to operate within some environment. These attributes relate to the functions that the machine is capable of performing, or the types of objects to which it can apply these functions. The reader will appreciate that these machines have much in common with Dijkstra's "pearls". (Dijkstra 1972a).

Pearl provides a generalized programming language to be applied in any such machine. The programmer may specify an ideal machine for his purpose by particularizing this general language. The design task then becomes one of implementing those features introduced by the programmer which are non-primitive in the underlying actual machine (the base machine). This task may be carried out by the introduction of further ideal machines each suited to a particular purpose.

Each machine is considered to exist in an environment of other machines according to its purpose in the design. This environment provides a partial particularization of the generalized programming language and augments the set of concepts available within a machine. A description of the form the environment takes is given in section 5.2.3 whilst a discussion of the implications of particular environments can be found in Chapter 7. It should be noted that the rules describing what

environment is available to a particular machine are closely tied to the desire to encourage top-down development of programs. It is not impossible to follow a bottom-up method, but in general the user will find this a devious thing to attempt.

Each machine introduced by the programmer represents a decision. A machine is limited to carry out one and only one program. Machines therefore differ from Dijkstra's pearls in this respect. (Further discussion related to this point is given in Chapter 7).

We have described one relationship that may exist between a pair of machines (i.e. one machine implements a feature introduced by another machine). This relationship suggests that the set of machines used in a program development may be represented as a tree. However, as will be described in succeeding sections, other relationships are also allowed amongst machines. These tend to structure the set of machines into a directed graph rather than a tree.

The representation of a developing program using a generalized programming language is related to the concepts of extensible programming languages (e.g. SIMULA 67, Dahl, Myhrhaug and Nygaard 1968), Algol 68 (van Wijngaarden 1969), ECL (Wegbreit 1971). However, there are differences as we hope will become apparent. In particular, in Pearl emphasis is placed on the way in which programs are constructed. From this point of view the actual syntactic forms of the Pearl notation may be considered immaterial. From others, however, (e.g. readability, comprehensibility) they are important and have been designed following the discussions of Chapters 2 and 3. Additionally, features of the notation allow the description of redundant information which is then available for certain checks to be made concerning the "correctness" of programs. Several of these have been outlined in Chapter 4.

The generalized language is based upon a particular programming language. This programming language (the base language) provides a level of description whose characterizing concepts are considered primitive. By this we mean that this set of concepts is understandable to an existing machine in much the same way as the concepts of Algol are understandable to an Algol machine. In any program development all other levels of description will be higher than the level of the base language.

The concepts of the base language are available in all machines introduced by the programmer (i.e. they are "pushed through" the various levels of description characterized by programmer introduced machines).

These concepts are as follows:-

1. Data types - integer, string

2. Operations

(i) upon integer . . . +, -, *, /, =, <, >, &, |, ¬ >=, <=, ¬ =

(ii) between strings . . . || (catenate)

substring selection

(iii) some i/o operations . . . readint (integer),
writeint (integer),
nlcr,
prsym (integer)

(iv) declaration

(v) assignment (:=) is available between instances of similar data types.

3. Control structures

(i) sequence	. . .	S1; S2
(ii) alternative	. . .	if E then S1 else S2
(iii) conditional	. . .	if E then S1
(iv) repetition	. . .	while E do S
		repeat S until E

S, S1, S2 are statements and E is an integer valued expression. The value 1 is taken to be "true", any other value as "false".

4. Data structures . . . vector (together with subscription).

A definition of the language is included in Appendix A.

Programmer controlled generalizations of the base language are confined to data types and operations.

The concept of a data type is generalized to allow any data type the programmer wishes to identify. The concepts of declaration, subscription and assignment are all generalized in the obvious ways. The assignment operator (:=) is used to represent all assignments, with the restriction that its operands must both be of the same data type. The meaning implied by this operation is that an application serves to make the value of the left-hand operand the same as the value of the right-hand operand.

The concept of an operation is generalized by providing a standard form in which operations may be written. This is in prefix form:-

⟨name⟩(operand 1, operand 2, . . ., operand n)

or, if there are no operands to be named, simply

⟨name⟩ (e.g. nlcr)

The generalizations allowed correspond to the notions of representational abstraction and operational abstraction as described in Chapter 3.

5.2 Constructing a program (using the *build command)

The nature of the Pearl system is that it is driven by commands issued by the user at an online terminal. One of these commands allows the user to build up a complete description of a machine and add it to the data base describing his developing program. This command is the "*build" command. For the remainder of this chapter we will describe and discuss how the user "builds a machine", what assistance he can obtain and what restrictions are imposed to encourage the structuring that we have described.

5.2.1 The specification of a machine

Each machine introduced by the programmer is given a name. This name serves to label the machine as a complete unit, covering both its specification and its action.

Machines are introduced to carry out a particular function. They thus represent a conscious design decision made by the programmer. Provision is made for the programmer to document this decision in the form of a comment. Together with the machine name and some punctuation this serves as a heading for a machine.

example:

```
cardprocessor:  'read each card and then process it'
```

The identification of the particular concepts understood by this machine now follows. A new data concept may be introduced by a type statement, and a new operation concept by an operation statement.

examples:

```
type        cardimage
operation   print (cardimage c)
```

It is worth stressing that in both of these cases no indication is given of either how a cardimage is to be represented, or how the print operation is to be carried out. The names given to the concepts will no doubt have a meaning for the programmer.

The form of the operation statement is not unlike a procedure heading in Algol-like languages. Each formal operand may optionally be specified as "vary". Only operands so specified are subject to a change of value as a result of the action of the operation.

example:

operation read (cardimage c vary)

This mechanism will be further described in section 5.2.6.1.

5.2.2 Describing the action of a machine

The specification of a machine and its environment serve to describe a particular programming language. The desired action of the machine may be described by writing a program for the machine in this programming language.

Figure 5.1 gives an example of a machine which processes cardimages. This machine represents the first stage in the process of constructing a program to solve the problem we described in Chapter 3. As this problem will be used as an example throughout this chapter, it is repeated here.

> "Write a program which reads 10 input cards and tests these same 10 input cards for the following conditions. Each of the first 9 values on each card should be within certain limits. The 10th value should also be within these limits and, further, should be a check upon the preceding 9 values on that card".

```
build
ardprocessor:'read each card and then process it'
egin type cardimage;
     operation read(cardimage c vary);
     operation process(cardimage c);

rogram:
   declare cardimage c;
   declare integer i;
   i:=0;
   while i<10 do
    (i:=i+1;
     read(c);
     process(c)).
nd
ND OF CHECKING
O ERRORS WERE DETECTED.
```

Figure 5.1

Most of the features shown in Figure 5.1 have been described. Of those that have not, only one requires extensive discussion. ("program").

The name "program" expresses the function carried out by the machine as an elaboration of a concept introduced by some other machine. The concept "program" is provided by the system and is thus the standard "starting point" for any design.

The labelling of the program part may appear to make the machine name redundant. In a completed program this is probably true. However, the use of separate machine names allows a greater flexibility both from the point of view of the user as a reference mechanism, and also to enable alternative machines to be described elaborating the same concept. (In the present implementation of Pearl this is not allowed, but it is much more in the spirit of Dijkstra's "necklace of pearls". (Dijkstra 1972a). Some discussion of this idea appears in Chapter 7).

5.2.3. The environment of a machine

The environment within which a machine may be defined is specified in terms of the operations and types that are available to it. The rules governing the introduction of machines and therefore the introduction of operations and types are framed to accord with the philosophy of top-down program construction.

Machines are introduced in a specific time sequence. At the commencement of a program construction (i.e. at the time the concept "program" is unelaborated) the set of operations available in the environment is that provided by the base language. Each machine introduced by the programmer may modify the environment, and this modified environment is then available for subsequent machines. Thus the set of operations and types is considered global to all machines subject to one or two restrictions which are explained below.

A machine may augment the environment by the introduction of data types and operations not already present. It may modify the environment by elaborating a concept that exists within the environment but which is not already elaborated. A machine may not elaborate a concept introduced in that machine. If a machine elaborates a data type then this data type is subsequently removed from the environment (see also section 5.2.5). Thus programmer elaborated data types are unavailable later in the construction sequence. Some discussion on the effect that this particular rule has had is given in Chapter 7.

The above rules encourage top-down program construction. It is a reasonably simple matter to envisage different rules governing the environment which would encourage other construction strategies. (see, for example, section 7.1.2). The rules chosen are extremely simple and have proved to be quite satisfactory once their devotion to a top-down philosophy is appreciated.

It will be noticed that use of concepts introduced into the environment will, in general, destroy a purely hierarchical arrangement of machines. It would of course, be possible to describe rules which would not allow this. Such a scheme, if enforced rigidly, would bar such notions as concept sharing between machines of different sub-trees in the hierarchy. One possible relaxation of this would be to use machine names themselves to make concepts available. This last suggestion is similar to the referencing of block attributes in SIMULA 67. (Dahl, Myhrhaug and Nygaard 1968).

For the purposes of the present system, however, it was considered that the simple approach implemented offered a reasonable degree of power with only an occasional frustration.

5.2.4. Elaboration of an operational concept

A conceptual operation is added to the environment by an operation statement. As pointed out above, this introduction corresponds closely to a procedure heading. We continue the analogy in describing how such a concept is subsequently elaborated by another machine.

The action part of a machine is associated with a particular operational concept by labelling it with the name of that operation. This may be seen as physically linking the text of the procedure heading to the text of the procedure body. Thus each may appear at the appropriate time in the construction process.

Figure 5.2 (part of the card processing program) gives a sequence of operation elaborations. It extends Figure 5.1 by a further 2 machines.

The program describing how an operation is carried out will normally reference its operands. Thus the label that is applied to the action part of the elaborating machine should display these operands. If it was allowed to use the name of the operation alone (which is sufficient), then understanding the elaboration of the operation is likely to involve the human reader in considerable cross-checking between machine descriptions.

example:

In a machine A an operation 'swap' is introduced as:-

operation swap (integer x vary, integer y vary).

At some later stage in the construction, machine B is introduced to elaborate swap. It is sufficient to write:-

swap: declare integer z;

z: = x; x: = y; y: = z.

```
+*build
 cardprocessor:'read each card and then process it'
 begin type cardimage;
       operation read(cardimage c vary);
       operation process(cardimage c);

 program:
     declare cardimage c;
     declare integer i;
     i:=0;
     while i<10 do
      (i:=i+1;
       read(c);
       process(c)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 processor:'check the values and the check'
 begin operation checkcard(cardimage c, integer ok vary);
       operation rejectmessage;
       operation writeout(cardimage c);

 process(cardimage c):
     declare integer ok;
     checkcard(c, ok);
     if ¬ok then rejectmessage;
     writeout(c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 checker:'check the values, then and only then, the check'
 begin operation checkvalidity(cardimage c, integer ok vary);
       operation checkcheck (cardimage c, integer ok vary);

 checkcard(cardimage c, integer ok vary):
      checkvalidity(c, ok);
      if ok then checkcheck(c, ok).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

Figure 5.2

However, when confronted by this piece of text, what meaning should a reader associate with the variables x and y?

The actual implementation does, in fact, include a mechanism for supplying the programmer with the original operands if necessary. (An example is shown in Figure 5.3).

It is possible to introduce a conceptual operation without using the operation statement. The symbol ":=" is used to denote assignment of the value of one variable to another variable irrespective of the type of these variables but provided the two variables are of the same type. If this type is primitive (i.e. not programmer introduced) then the operation denoted is also primitive. If, however, the type is not primitive, then the operation denoted is conceptual. As such it will require further elaboration when the data type in question is elaborated.

When the symbol ":=" is used (in a machine) to denote a conceptual assignment operation it is considered exactly as if an operation statement had been used to introduce it. A system generated name is used to denote the operation together with some formal operands.

example:

If in a machine we have the following:

```
declare  value (temp1,  temp2);

         . . . . . .

           temp1: = temp2;
```

then the ":=" represents a conceptual operation of assignment of a "value". The system treats this exactly as if the programmer had explicitly stated:

```
operation  value__assign (value value1 vary, value value2).
```

```
+*build
 checker:'check the values, then and only then, the check'
 begin operation checkvalidity(cardimage c, integer ok vary);
       operation checkcheck(cardimage c, integer ok vary);

 checkcard:
 *** WARNING, ORIGINAL HAD PARAMETERS
 WILL USE ORIGINAL PARAMETERS AS FOLLOWS
 CARDIMAGE C
 INTEGER OK VARY
      checkvalidity(c,ok);
                 . . . .
```

Figure 5.3

This mechanism allows the user to properly label the elaboration of the assignment operator acting between operands of a conceptual type.

This particular approach was chosen for its simplicity. Some further discussion on the whole problem of the generalized assignment operator is given in section 7.1.5.

5.2.5. Elaboration of data types

Machines may be introduced, not only to indicate how an operation is carried out, but also to give a representation of a conceptual data type. The two functions are similar.

Instances of a particular data type are created using a declaration. Thus

declare integer i

allocates a certain amount of a resource (called memory) and marks it as an integer to be referenced by the name "i".

In exactly the same way

declare cardimage c

may be considered as allocating a certain amount of memory which will be considered as a cardimage and be referenced by the name "c". In both of these examples, the effect of the declaration is an allocation of a "certain amount of memory" together with a reference to its type and name. The actual amount of memory is dependent upon the representation of the data type in terms of the memory elements themselves.

In the case of the primitive data types, the representation is defined. For conceptual types, the amount allocated will depend upon

the structure subsequently given to the data concept by the programmer.

The introduction of a new conceptual data type can be thought of as the introduction of an unelaborated operation upon memory. The two primitive types are thus primitive operations on memory. The creation of an instance of a data type is thus a call upon the relevant operation. The elaboration of a conceptual data type is thus similar to the elaboration of a conceptual operation.

Figure 5.4 shows a further machine from the cardprocessing program elaborating the data concept "cardimage".

Instances of a data type may be initialized by incorporating the necessary operations in the program of the machine elaborating that data type. Figure 5.5 shows this in a modification of machine "cardrep" of figure 5.4. The variable "i" is local to the inner block. The names of variables declared in the outermost block of a program elaborating a data type are available to machines elaborating operations having operands of that data type.

Pearl enforces a rule that such operations must be elaborated immediately the data type is elaborated (see also section 5.2.6.3). This rule is a recognition of the strong relationship that exists between a data type and operations upon instances of that data type. Once the necessary machines have been entered, the elaborated data type is removed from the environment thus disallowing the recursive definition of data types. This is aimed at encouraging top-down program development.

The sequence of machines shown in figure 5.6 is an illustration of the elaboration of operations related to an elaborated data type.

The data type "cardimage" has previously been introduced together with the operations "read", "writeout", "checkvalidity" and "checkcheck", each having an operand of type "cardimage" and still being unelaborated.

```
                    .
                    .
                    .
+*build
 cardrep:'a card is 9 data values and a check'
 begin type value;

 cardimage: declare vector(9) value data;
            declare value check.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

                    .
                    .
                    .
```

Figure 5.4

```
                    .
                    .
                    .
++build
 cardrep:'a card is 9 data values and a check'
 begin type value;
       operation clear(value v vary);

 cardimage:
         declare vector(9) value data;
         declare value check;

         (  declare integer i;
            i:=0;
            while i<9 do
            (  i:=i+1;
               clear(data(i)) );
            clear(check)   ).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

                    .
                    .
                    .
```

Figure 5.5

```
++build
 cardrep:'a card is 9 data values and a check'
 begin type value;

 cardimage: declare vector(9) value data;
            declare value check.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

++build
 cardreader:'reads in the 10 values'
 begin operation readvalue(value v vary);

 read(cardimage c vary):
      declare integer i;
      i:=0;
      while i<9 do
      (i:=i+1;
       readvalue(data(i) of c) );
      readvalue(check of c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

++build
 cardwriter:'writes out values anyway'
 begin operation writevalue(value v);

 writeout(cardimage c):
       declare integer i;
       i:=0;
       while i<9 do
       (i:=i+1;
        writevalue(data(i) of c) );
       writevalue(check of c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

Figure 5.6

```
++*build
 validity check:'checks the 9 values'
 begin operation checkvalue(value v, integer ok vary);

 checkvalidity(cardimage c, integer ok vary):
          declare integer i;
          i:=0; ok:=true;
          while i<9 & ok do
          (i:=i+1;
           checkvalue(data(i) of c, ok) ).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

++*build
 checker:'make sure check is ok'
 begin operation combine(value v vary, value w);
       operation comparevalue(value (u,v), integer ok vary);

 checkcheck(cardimage c, integer ok vary):
       declare value temp;
       declare integer i;
       i:=1; temp:=data(1) of c;
       while i<9 do
        (i:=i+1;
         combine(temp, data(i) of c));
       comparevalue(temp, check of c, ok).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

Figure 5.6 (continued)

Machine "cardrep" is defined giving a representation for the type "cardimage" in terms of the type "value". Once this machine has been accepted by Pearl, the programmer is constrained as to what he may subsequently enter. He may only enter machines which elaborate operations having an operand of type "cardimage" until such time as no such operations remain unelaborated. Thus the set of machines "cardreader", "cardwriter", "validitycheck" and "checker" (in any order decided by the programmer) must be entered before a machine which elaborates some other concept either of operation or of type. (e.g. the type "value"). Only when this particular set of machines has been accepted is the type "cardimage" no longer available in the environment.

A special operator (of) allows reference to particular elements of the elaboration of a data type during the elaboration of the related operations.

The enforcement of this strategy calls for a few comments. However, beyond noticing that the set of machines providing the representation of the data type and the elaborations of all related operations has much in common with the class concept of SIMULA 67 (Dahl, Myhrhaug and Nygaard 1968) we postpone discussion until Chapter 7.

5.2.6. Correctness considerations

In addition to what we have described above, there are a number of features provided by the system which allow the programmer to increase the confidence he is willing to place in what he has written down.

We saw in Chapter 4 that the provision of redundant information is a powerful method of increasing the understanding that may be gained of a program. The various features to be described allow the programmer using Pearl a number of ways of saying what he understands by what he has written in his program, or what he intends to write. As a by-product

of this, there are a number of occasions when the redundancy can be checked by the system in an automatic or semi-automatic fashion.

There are two main areas of interest. Firstly, because the actions of machines are described by programs, then these programs are subject to problems of comprehension much as traditional programs are. Secondly, in a development of a program in multi-level fashion there is a requirement to ensure that descriptions given at different levels of description are mutually consistent.

5.2.6.1. Assertions etc.

There are several features provided which are best classified as being of a miscellaneous nature.

An "assert expression" is provided as part of the base language.

example:

assert x = a & y = b before . . .

Assertions may be made about the state of a computation at any point within an individual program for a machine. In the system as implemented, these assertions are not used to generate verification conditions or for the automatic proof of program correctness, but rather act as run-time checks.

As an aid towards maintaining the correctness of an elaboration, some restrictions are applied to the mechanism used in parameter passing. This mechanism is known as "call by reference". (Note, there is nothing equivalent to a global variable common to several machines). Operations act upon their operands. This effect, when seen in procedures in high level languages, is often known as side effect in that it is possible, by a procedure call, to alter the value of a variable without explicitly making use of an assignment statement. Indeed it is

possible for a single procedure call to change the value of many program variables. In several current languages those parameters of a procedure whose value may be changed by calling that procedure are not distinguished syntactically. In Pearl, the vary attribute is provided. In the elaboratic of an operation, only such operands (or their components) as have been given the attribute vary may appear on the left-side of an assignment operator, or as an actual operand which itself has the vary attribute. Thus, when an operation is introduced, the programmer must specify which of its operands will be changed in value by that operation. The system will ensure that his specifications are not violated by later constructions. The vary attribute partitions the operands of an operation into two groups in a manner similar to that described by Hoare (1971b). Further discussion on the vary mechanism is given in section 7.1.4.

There are also some restrictions which prevent the programmer from doing things which may be considered unreasonable.

example:

> It is not possible to change the value of any (non-local) variable as part of the evaluation of a logical expression despite the fact that the base language is an "expression language".

Finally, all operands of operations are checked to ensure that they are of the type specified in the introduction of the operation or in the base language.

5.2.6.2 Meanings of conceptual operations

In section 5.2.1 we introduced the operation statement whereby new operations could be introduced. In order to allow the programmer to indicate the effect that an abstract operation has upon its operands withou

describing how the effect is achieved, the operation statement is extended to express the "meaning" of the operation being introduced. This takes the form of a pre-condition and a post-condition described by assertions over the operands of the operation. Thus the syntax of the operation statement may take the extended form:-

operation	⟨name⟩ ⟨operand list⟩
provided	⟨pre-condition⟩
yields	⟨post-condition⟩ onexit

Both the pre-condition and the post-condition are logical expressions, but certain restrictions apply to the latter in order that the meaning of the operation may be deterministic. A discussion of some of the implication of this restriction is to be seen in Chapter 7, whilst an argument for its presence in the current system may be found in Chapter 6. To ensure a fully deterministic meaning for an operation in a fairly trivial manner, logical disjunction is disallowed in the post-condition. Also in the post-condition the usual symbol for logical conjunction (&) is replaced by a comma so that the post-condition can be expressed as an atomic list of assertions about the operands using a comma to separate the elements.

example:-

```
operation  swap (integer (x,y) vary, integer (a,b))
provided   x = a & y = b  yields
           x = b,  y = a  onexit
```

5.2.6.3. States

To enable the expression of assertions about variables of non-primitive type, a further concept is introduced; that of "state". States are a means of indicating a condition in which an instance of a data type may be found. They are derived directly from the need to express the result of an operation on a conceptual data type. However, they may also be used in conjunction with the primitive types integer and string.

States for a type may be introduced at any time that an operation using an operand of that type may be introduced. Their introduction is part of the machine specification and is effected by the states statement. The form of this statement is similar in form to an operation statement, but without a meaning part.

example:

```
states empty (queue a)
```

Once a state is introduced, it may be used to define the meaning of an operation.

example:

```
operation  clear (queue  a  vary)
provided ¬ empty (a)  yields
           empty (a)  onexit
```

States may also be used as logical functions which may be tested in a program.

example:

<u>while</u> ¬ empty (a) do

A state may be undecidable in addition to being either true or false.

States may be elaborated in a similar fashion to the elaboration of operations, and the restrictions which apply following the elaboration of a conceptual data type are extended to cover the elaboration of states of that data type. (see section 5.2.5).

Two ways in which elaborations of states may be used are given below. Both exemplify a different stress applied in the derivation of a design.

If a state is used to express the meaning of a particular operation, then the elaboration of the state may serve as a check on the elaboration of that operation. This is illustrated in figure 5.7. The machines presented there are taken from a modified development for the checking problem used earlier in this chapter (see figure 5.2).

An additional operation "initial" is introduced to ensure that the variable "c" is in the correct state for the first "read" operation. The operation "process" is defined to yield a cardimage in the state "processed". However, from its elaboration in the machine "processor2", it is seen that, as a result of the application of the "writeout" operation, the cardimage will, in fact, be in the state "written". The elaboration of the state "processed" as meaning "written" restores the correctness of the program.

From a different point of view, states may be used to specify a program development. A program may be defined by giving the states necessary to fulfill the requirements of the program. The program development takes the form of defining an operation which satisfies

```
+*build
 cardprocessor2:'as cardprocessor, plus states for checking'
 begin type cardimage;
       states readin(cardimage c),
              processed(cardimage c);
       operation initial(cardimage c)
         provided true yields processed(c) onexit;
       operation read(cardimage c vary)
         provided processed(c) yields readin(c) onexit;
       operation process(cardimage c)
         provided readin(c) yields processed(c) onexit;

 program:
      declare cardimage c;
      declare integer i;
      i:=0; initial(i);
      while i<10 do
      (  i:=i+1;
         read(c);
         process(c)  ).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*build
 processor2:'as processor plus states'
 begin states passed(cardimage c), written(cardimage c);
       operation checkcard(cardimage c, integer ok vary)
         provided readin(c) yields passed(c) onexit;
       operation writeout(cardimage c)
         provided passed(c) yields written(c) onexit;
       operation rejectmessage;

 process(cardimage c):
      declare integer ok;
      checkcard(c,ok);
      if ¬ok then rejectmessage;
      writeout(c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*build
 statrel:'explain processed versus written'
 begin

 processed(cardimage c):
      written(c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

Figure 5.7

these states. Subsequent elaboration of the states thereby provides further specifications to be met by elaboration of the operations introduced to satisfy given state transitions.

example:

A program might require that an object of a data type "t" satisfies some predicate "p1". An operation to express this would be introduced as:-

operation op1 (t x vary)

provided true yields p1(x) onexit

Next the predicate p1 is elaborated as being some relationship rel between two other predicates p2 and p3.

Thus

p1 (t x) : p2 (x) rel p3 (x)

In order to satisfy this relationship two further operations could be introduced.

operation op2 (t x vary)

provided . . . . . yields p2 (x) onexit,

operation op3 (t x vary)

provided . . . yields p3 (x) onexit

These are then used to elaborate op1 so that the relationship between p1, p2 and p3 is met.

Thus the elaboration of states may be used either to drive the program design process, or be caused as a result of the design process. The particular stress applied is dependent upon the programmer himself and the problem he is solving.

5.2.6.4. Pre- and post-conditions upon programs

When an operation is elaborated, this elaboration may be given pre- and post-conditions. In the same way that pre- and post-conditions given at the time of an operation introduction may be considered as giving a

meaning to the concept of the operation, so the pre- and post-conditions applied to an elaboration may be considered as expressing the meaning of the actual implementation of the operation. A check between the two sets of conditions is provided as an aid towards correct elaboration. Such checking takes the form of a message to the programmer who can take action as necessary as no automatic theorem prover is implemented within the system.

The conditions applied to the elaboration part are in fact assertions although the syntax takes a slightly different form.

example:

Suppose that an operation is introduced.

operation op1 (. . .)

provided P(. . .) yields Q (. . .) onexit

Subsequently op1 is elaborated.

op1 (. . .):

provided R (. . .) then

.
.
.

assert S (. . .) onexit

The programmer is reminded that the following conditions should hold:

$P(\ldots) \Rightarrow R(\ldots)$

and $S(\ldots) \Rightarrow Q(\ldots)$

5.3 Supplementing the design with a new machine

Under the control of the *build command, the programmer can enter the text for a new machine into the system. Once this text has been satisfactorily checked the new machine is added to the program design. This necessitates modifications to the current environment as described in section 5.2.3.

In addition certain relationships are noted as to the place of the new machine within the total design structure. These relationships are expressed between the machines representing the design.

For the purposes of later discussion we introduce the notion of a machine being "dependent upon the existence of" another machine.

A new machine M1 is dependent upon the existence of another machine M2, present in the design structure, if:-

(i) M2 introduced the concept type, operation or state elaborated by M1.

(ii) M2 introduced some concept which is used anywhere within M1. (e.g. declaring an instance of a type, invoking an operation or using a state).

(iii) M2 elaborated a data type and M1 gives an elaboration for an operation or state upon that data type. (In this case M1 may make use of the representation of the data type as given in M2).

As a point of interest it should be stressed that relationships are expressed between machines only, and not between machines (or parts of machines) and individual concepts. Further discussion on the implications of this decision is given in Chapter 6, sections 6.1.1. and 6.1.2.

5.4 Discussion of the notation

There are a number of issues which require discussion with regard to the contents of this chapter. At this time we will deal only with those specific to the notation used to specify machines and their programs. Other discussion is left until Chapter 7.

5.4.1 Omissions

Whilst we cannot hope to give a complete list of those things which might reasonably be expected to appear in the Pearl notation but which do not, an attempt is made to cover the most glaring omissions.

(a) procedures or subroutines.

The procedure or subroutine is a most important structuring feature of most contemporary high-level languages. In a limited way the notion of an operation in Pearl serves a similar purpose whilst restricting the more general concept in a number of ways. By viewing a procedure purely as a particular form of control structure there would seem to be no strong argument for its omission. However, it was felt that its inclusion as such would add an unnecessary additional complexity to the description of the program of a machine as well as possibly allowing the programmer to build potentially large machines representing a set of design decisions instead of the one decision intended.

(b) functions

The notion of a function is almost entirely absent. A state covers a limited set of those conventionally available to a programmer. (Namely, boolean functions of a single argument). The omission is most noticeable when there is a need to express a relationship between a number of variables as a boolean function of n arguments. Such functions occur naturally as (for example) the conditions in alternative or iterative control structures. It was expected that the expression language nature of the base language would make unnecessary an explicit provision for such relationships. However, in our opinion the use of such concepts as "block expressions" detracts quickly from the clarity of programs and is, in general, a poor construction.

From experience, it is probable that there is a strong argument for the inclusion of explicit boolean functions of more than one argument. However, the further generalization to n-place functions of any type is of more doubtful value. The same effect can be obtained by an extension of the

function into an operation by an additional assignment to a vary operand. In these cases there appears to be no syntactic argument against such a construction.

(c) data types and structuring facilities

The two primitive types (integer and string) were chosen for their general usefulness and for the fact that the concepts they represent are reasonably well understood. It may be considered that lower level, more basic types should have been chosen in view of the fact that all data concepts must eventually find a representation in terms of the system provided types. However, if the base level is chosen too low, then it is less reasonable to ignore such complications as storage management primitives. In the current system the storage allocation is handled within the base level machine and the programmer has no way of altering the mechanism.

Experience has suggested that a further primitive type (the boolean or logical) should have been provided. The control structures which conventionally rely upon boolean expressions (e.g. the alternative and iterative constructions) instead use integer expressions using the value 1 as being equivalent to true and any other value as false. Similarly the relational operators (=, $<$, $>$, etc.) and "boolean" operators (&, |, $\neg$) take integer operands and produce integer results. Such use of the integer type is not, of course, unique. (It is to be seen in APL and XPL for example). It is acceptable until we consider the definition of the operators &, | and $\neg$. In order to give a definition for these operators over all (possible) integer values, it is necessary to assume a representation for integers. That chosen for the system as implemented is 16-bit 2's complement as the interpreter was to use half-word arithmetic on an IBM System /360 machine.

examples:

7 & 3 = 3

-2 & 5 = 4

Thus the base language is itself making assumptions about how one of its concepts is represented. It does not, therefore, truly represent a single level of conception. With hindsight, it is preferable to introduce the type boolean as a primitive type.

There are two structuring relationships that may be expressed amongst data elements.

One is that represented by the elaboration of a data type into a set of components. This relationship represents an abstraction relationship between two distinct levels of description used in the design.

The other relationship is that provided by the vector form. This serves to exemplify one of many possible such relationships which may be formed, which do not necessarily characterize a different level of description. Other possibilities including arrays, powersets and sequences are suggested by Hoare (1972a) of which several are available in the language PASCAL (Wirth 1971a). Whilst the provision of a single example of such a structuring relationship was considered sufficient for the purposes of the current work, it is likely that any practical system would require such other examples as we have suggested.

(d) control structures

The control structures represent a simple, self-contained set of elements to describe the sequential flow of program control. They do not allow the complexity of the for statements of Algol or the connectivity

presented by the goto. No attempt was made to include facilities for the description of either parallelism or co-routines.

(e) Operations

Given the choice of basic data types, the set of operations provided is representative of the set of possibilities, whilst allowing useful concepts to be described at the base level for the purposes of exemplification.

(f) Correctness facilities

The requirement that the definition of operation meanings be deterministic is a particular limitation. Additionally, as states are purely one place predicates, there is no way of specifying meanings as abstract relations. Further discussion on these points is given in Chapter 7 for consideration in possible extensions.

The system provides no automatic scheme for proving the correctness of either an individual program for an ideal machine or of the consistency of the overall design. The relevant sections of Chapter 4 deal with this point. One possibility that could have been implemented is the automatic generation of verification conditions. This was not done, purely for reasons of time and not because of the lack of belief in the practical utility of such a tool.

Likewise, there are undoubtedly several other ad hoc features that could be included to catch possible program errors.

example:

> It is possible to check, in some cases, that the logical expression controlling a loop may not be altered by computation within the loop.

The usefulness of such checks is of doubtful general worth and, again, time precluded any investigation.

5.4.2. Generalization of control structure elements

The base language of Pearl is an "expression language" (see Wirth and Weber 1966, for example). Statements of the language potentially have values and may be used as operands in the formation of expressions. In some ways this allows a simplification of the concepts of the programming language and so should reduce its inherent complexity thereby increasing the chance of comprehension by the programmer.

examples:

(1) The well-known conditional expression

a: = if E then b else c is derived from the use of the general alternative control structure element of the language as the right operand of the assignment operator.

(2) It is possible (as in CPL for example, Barron, Buxton, Hartley, Nixon and Strachey 1964) to write (if E then b else c) : = a where the same alternative control structure is used as the left operand of the assignment operator. (The parentheses are needed to achieve the correct precedence of alternative over assignment).

(3) The semi-colon may also be used in this way.

a: = x + y + z; a/2

has the value a/2

Unfortunately it seems to be the case that such generalization allows the programmer too much freedom and can lead to unnecessary complexity. Indeed it may well be that it encourages the programmer to attempt devious program constructions. It is not an impossible task to conjure up programs that when unravelled are quite sensible, and yet are textually insanely complex.

A conclusion which may be reached is that elements of a programming language whose purpose is to express a flow of control should in general be distinguished from elements whose purpose is to identify particular actions to be carried out. As Wilkes (1968) has suggested, there are benefits to be achieved if it is possible to separate the notions of control flow from consideration of particular operations or data types, although this may be a difficult task.

5.4.3 States, values and generalized constants

The idea of a state was introduced to allow the programmer to express the result of an operation.

There is a very close analogy between the notion of a state and the abstraction of a value, or set of values.

example:

> The state "even(i)" where i is an integer, represents the abstraction from all possible integer constants which are even.

States may be considered as representing conceptual values of conceptual data types.

example:

> Given the data type "queue" the state "empty (queue q)" may be thought of as expressing one particular value that a queue may take.

However, such analogies, whilst useful, do not express the full intention of the general notion of states in the present experimental system. It is possible to define an operation which changes the state of one of its operands even though that operand does not possess the attribute vary.

example:

operation print (cardimage c)

provided ¬ printed (c) yields printed (c) onexit

This example serves to illustrate the intended use of states in allowing the programmer a formal means of expressing his intention without necessarily committing himself to particular implementations of that intent. The operation introduction expresses a clear intent. The operation "print" will not change the value (in a primitive sense) of the operand, but its application is an event with significance which is to be recorded. This use of a state has proved to be of benefit in expressing the use of a variable (see for example the development given in Appendix D).

It is possible that the additional insight given by an investigation of the application of 'invariants' for a data structure will suggest better how a state is related to the various notions discussed above.

As was described earlier, (section 5.2.6.3.) it is possible to conceive of the definition and elaboration of states as driving the program design. It is interesting to speculate whether it is more helpful to think of a program being developed in terms of the operations and data structures necessary to describe the required process (with states being used to validate the program so developed) or whether in fact states are indeed the way in which the necessary operations and structures are determined. The view taken by Schwartz (1970), that there are various advantages to be gained when a system is built through consideration of its data, would seem to support the latter approach.

5.5. Some comparisons with other programming notations

There is some similarity between the scheme presented above and extensible programming languages. However the extension

mechanism is unusual. In ECL (Wegbreit 1971) or Algol 68 (van Wijngaarden 1969), the extension is made outwards from the actual objects present in the base language. It is necessary when introducing a new concept, to give its representation. In Pearl, the extensibility is based upon a generalization of a programming language together with a separate mechanism for relating concepts to a representation in the base language. This allows a greater freedom of expression and, in general (although not in Pearl as implemented), the possibility of a variety of design strategies (including bottom-up for example).

SIMULA 67 (Dahl, Myhrhaug and Nygaard 1968), whilst also exhibiting an extension mechanism which is primarily bottom-up, does provide a neat encapsulation of the relationship between a data concept and the set of operations associated with that data concept. The language itself probably suffers, however, from its historical derivations and resultant overall complexity. We have earlier discussed the role of SIMULA in the context of the representation of program designs on many levels (see Chapter 3).

Pearl is unusual in its enforcement of a particular design discipline. We have earlier discussed how programming notations influence program development. In the design of Pearl an attempt has been made to take advantage of this fact in order to encourage design in particular ways. By way of contrast, although the AED-0 language (Ross 1969) and the AED philosophy itself (Ross 1967) are based upon a similar recognition, the programmer is given immense freedom and facility to build models and designs. This freedom allows the careful programmer a wide range of expression, but in doing so opens the way to unbridled complexity. The Pearl philosophy may be stated more in terms of giving the programmer enough rope to do something constructive, but not enough to hang himself. Whether it would be possible to maintain this philosophy if additional power was added (e.g. in the number of conceptual relationships that could be represented) is an open question.

It is the author's belief that it would, provided the additional complexity was constrained to be used in particular ways which did not result in the connectivity of substantially different concepts being increased beyond some reasonably low bound.

In its provisions for the specification and maintenance of correctness criteria, Pearl is by no means unique. An equivalent form of assert expression is to be seen in, for example, some implementations of Algol W (Algol W 1972) and in the language Nucleus (Good and Ragland 1973). The provision of a means of giving meanings to conceptual operations is less common. There is a similarity with assertional languages such as ABSET. (Elcock, Foster, Gray, McGregor and Murray, 1971).

5.6. Summary

In this chapter we have concentrated on one particular feature of the Pearl system; namely the manner in which it assists in the actual construction of a program. This necessarily entailed a description of the bases of the system for describing and checking a multi-level design using one design strategy in particular. In the next chapter we will describe the other facilities provided by the system for the editing, interrogation and interpretation of the information contained within the data base of the program design.

Chapter 6:

Extended Facilities of Pearl

In this chapter we describe the facilities provided by the Pearl system which allow the programmer to carry out design modification and design evaluation, and to request information about the state of a design.

Chapter 5 described how the programmer can construct a program using a particular notation together with some machine assistance. One important aspect of the assistance provided is the construction of a data base representing the evolving design. It is not difficult to visualize the programmer developing his design in the way described, using the machine to check each piece in much the same way as a conventional compiler might do, but not making use of the machine to maintain the design at all. The medium in which the design is stored may then be represented as a pile of paper.

example:

> It is possible to develop a program written in Algol in a similarly structured manner. Each individual Algol text may be checked by an Algol compiler, but the relationships existing between individual texts will not be recognized and stored by anyone other than the individual programmer.

The drawbacks of such a medium are obvious when consideration is given to the functions which may be applied to discover information pertaining to any particular level of description. One effect of the awkwardness of information retrieval is that errors are "corrected" by patching those texts which are easily available (generally the base level program) rather than by a proper modification to the design at the

appropriate level.

> "It is the patching of partially correct programs that makes them obscure".
>
> (Henderson and Snowdon 1972).

In the Pearl system, the computer itself is used to maintain the information representing the design as a data base and facilities are provided to enable easy access to this information so that proper modifications can be made.

Other tools may obviously be provided to act upon the information in the data base. One such is an interpreter enabling the run-time evaluation of the program under development. In the current implementation, this interpreter is limited in the facilities it provides. For example, the primitive type string has not been implemented whilst error checking and reporting facilities, whilst being available, are not as extensive as some of those described in Chapter 4. Other tools which could be provided in an extension of the current system readily suggest themselves. We give as examples:-

- automatic or semi-automatic program prover.
- an automatic means of checking for correct construction.
- powerful debugging aids.
- translator into an existing language or to machine code.

The system is used interactively from a terminal (although it can be used in batch mode) with the various tools being invoked by a set of commands. The *build command was introduced in the previous chapter. The

majority of the remaining commands will be introduced in the following sections. (For a complete list, see appendix B). Examples will be used where appropriate. Several of these are taken from the program developments shown in appendices D, E and F.

6.1. Modification of the Design

There are two commands which allow the programmer to modify an existing design. These both use a "machine" as the unit of editing. Modification may be carried out locally by a replacement command, whilst more drastic alterations may be carried out by invocation of a deletion command.

6.1.1. Replacement

The *replace command is designed to allow the replacement of a single named machine by another machine. (As an extension we might consider the replacement of a set of connected machines by a different set of connected machines).

The replacement of a machine is not dissimilar to the original introduction of a machine using *build. However, it is necessary to

(a) re-construct the environment of the machine being replaced, and

(b) impose certain additional restrictions upon the replacement machine so as not to violate the currently existing environment or its development.

It is a reasonably trivial matter to ensure that condition (a) can be achieved, whilst use is made of the dependency relationships that are defined between machines (see Chapter 5, section 5.3) to construct the necessary restrictions in (b).

In particular it is required that the specification part of the replacement machine should include the specification part of the machine

being replaced to the extent that individual concepts are re-introduced. This requirement is imposed because no means are provided (except for exhaustive search) within the system by which to ascertain whether or not an individual concept introduced in one machine has been used by any other machine. A different implementation would ease this requirement. (Appendix B contains a complete definition of the restrictions).

Operation meanings may be changed provided the programmer accepts that the new meaning implies the old meaning. This is an instance of the fact mentioned above that it is non-trivial (although possible) to discover whether a particular operation meaning may have been made use of in some machine.

Figure 6.1 illustrates a part of a Pearl session in which a machine is replaced by another, and operation meanings are checked. It is based upon the development given in appendix D.

It will be appreciated that the action provided by the replacement command is limited. Figures 6.2 and 6.3 may help to clarify the command further in view of the restrictions given.

Figure 6.2 shows a design built from 5 related machines M1, M2, M3, M4, M5. Each machine is represented as a node. The full lines linking two machines represent the elaboration of a concept introduced by the machine nearer the root by the machine further from the root. The labels on these lines identify particular concepts. Thus the concept c is introduced by M1 and elaborated by M2. The dashed lines between machines represent other dependency relationships. Thus M5 is dependent upon M2 through use of the concept d introduced in M2.

```
+*build
 liner1:'we print an image by printing its lines'
 begin
   states lineprinted(line 1),linebuilt(line 1);
   operation lineprint(line 1)
     provided linebuilt(1) yields lineprinted(1) onexit;

 print(image i):
       declare integer j;
       j:=21;
       while j>1 do
       (j:=j-1; lineprint(1(j) of i)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

              .
              .
              .

+*replace liner1
 liner11:'we print an image by printing its lines'
 begin
   states lineprinted(line 1),linebuilt(line 1),
          lineempty(line 1);
   operation lineprint(line 1)
     provided linebuilt(1) | lineempty(1)
     yields lineprinted(1) onexit;
 ? DOES
 LINEBUILT ( L )
 IMPLY
 LINEBUILT ( L ) | LINEEMPTY ( L )
 yes
 ? IS
 LINEPRINTED ( L )
 IMPLIED BY
 LINEPRINTED ( L )
 yes
 print(image i):
       declare integer j;

              .
              .
              .
```

Figure 6.1

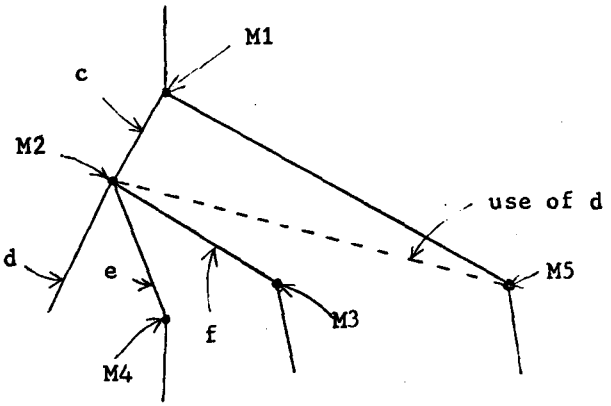


Figure 6.2

Suppose that a decision is taken to replace M2. The replacement machine must fit into the position occupied by M2 in the structure of figure 6.2. Figure 6.3 shows the structure that is left if M2 is removed.

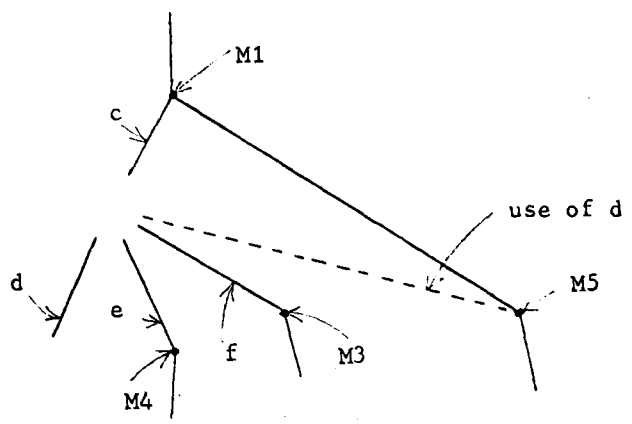


Figure 6.3

Thus the replacement machine must

(i) elaborate the concept c introduced in M1, and

(ii) provide (introduce) concepts d, e and f.

The replacement command is intended to illustrate how the programmer can make slight perturbations to a design without discarding previous work. Obviously, other similar tools could be provided, whilst different implementations of the system could relax the restrictions that apply.

The replacement of a machine which elaborates a data type imposes an additional constraint upon how design may proceed. Following such a successful replacement, the programmer must provide replacements for machines which give elaborations for operations and states which use an instance of this data type as an operand or parameter. (This constraint is equivalent to that imposed when a machine giving a representation of a data type is first entered; see section 5.2.4.). In this way the programmer is protected from overlooking the consequences of a different representation of a data type.

Figure 6.4 shows an example where this restriction applies. (Taken from the example of appendix D).

The machine "longrep" gives a representation for the type "line". Machine "longrep1" elaborates the operation "lineprint" using this representation. Subsequently a different representation for a "line" is thought more appropriate. The machine "shortrep" replaces "longrep" to carry out this change. The programmer is now constrained to give a replacement for "longrep1" reflecting the altered representation of a line. This he does using machine "shortrep1".

(A facility is provided to circumvent this constraint. The programmer may indicate that he wishes to "leave" the original machine).

6.1.2. Deletion of machines

By using the command *delete, the programmer may remove a named

```
+*build
 longrep:'a line is simply a vector of 20 symbols(integers)'
 begin

 line: declare vector(20)integer symb.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*build
 longrep1:'print a line by using prsym'
 begin

 lineprint(line 1):
       declare integer j;
       j:=0;
       while j<20 do
       ( j:=j+1; prsym(symb(j) of 1));
       nlcr.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

            .
            .
            .

+*replace longrep
 shortrep:'include a count of symbols to be printed with line'
 begin

 line:  declare integer f;
        declare vector(20) integer symb.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*replace longrep1
 shortrep1:'print f symbols using prsym'
 begin

 lineprint(line 1):
       declare integer j;
       j:=0;
       while j<f of 1 do
       ( j:=j+1; prsym(symb(j) of 1));
       nlcr.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

Figure 6.4

machine from the data base completely. The net effect is to leave the design as if the machine had never existed. To achieve this, the command is more powerful than it might at first appear.

If a machine is deleted, then all of the data types, operations and states which are introduced by that machine are also deleted. It is necessary, therefore, to delete, in addition, all those machines which depend upon the existence of a machine being deleted. Deletion of these machines causes deletion of further dependent machines and so on. In a highly connected system of machines, it is easy to see that the explicit deletion of one machine can have a drastic effect upon the remainder of the structure. Of course, as the data base represents a set of machines which must all be connected directly or indirectly to the initial ideal machine elaborating the "program" concept, it is a trivial matter to delete the whole program design. The delete command should obviously be treated with care.

Figure 6.5 offers an illustration

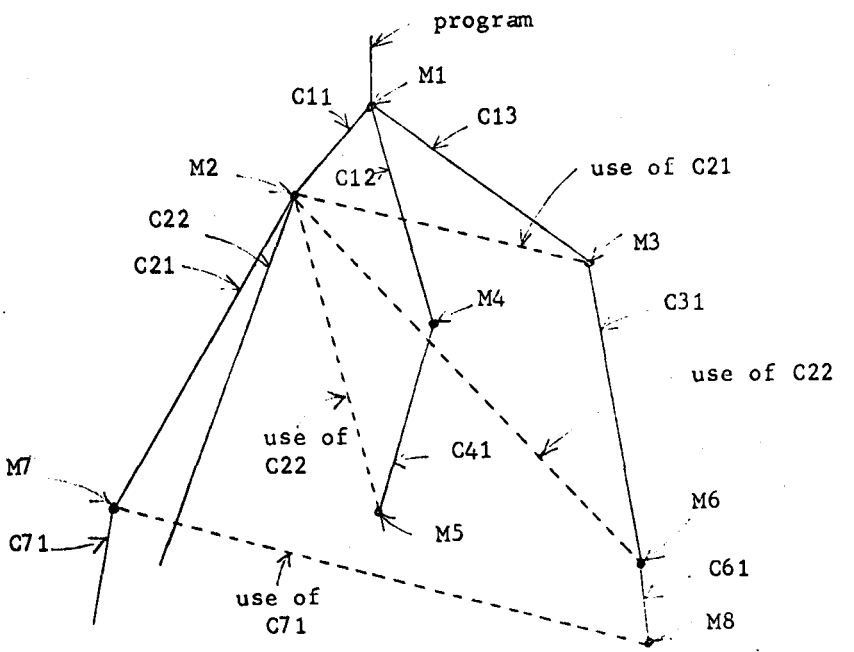


Figure 6.5

If the command

*delete M8

is issued, then only machine M8 will be removed.

If the command

*delete M2

is issued, then this will cause deletion of machines M2, M7, M5, M6, M8, M3.

6.2 Interrogation of the design

A command is available (*list) by which information may be retrieved from the data base and presented in readable form to the user. The command may be parameterized according to simple rules so that the user can request that specific information is displayed. A full list of the options available is given in appendix B.

Figure 6.6 shows an example of the use of the *list command based upon the program development of appendix D. This particular example illustrates the formatting feature provided for the display of text. Any formatting information present when text is input to the system is destroyed. Standard formatting is applied when text is displayed by the system for the user, thereby making textual input a less laborious task than might otherwise be the case.

6.3 Design evaluation - program execution

The *execute command invokes an interpreter to execute the program under design. This interpreter has a number of features. Perhaps most interesting is its ability to execute a program which is not complete. This allows some evaluation of a program design at any stage in its

```
+*build
 jscanner:'setmarks. put each of the 40 marks into image'
 begin operation addmark(integer j, image i vary)
 provided j>0 & j<=40 yields true onexit;
 setmarks(image i vary):
 declare integer j;
 j:=0;
 while j<40 do
 (j:=j+1; addmark(j,i)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*list jscanner
 JSCANNER IS A MACHINE
 JSCANNER:'SETMARKS. PUT EACH OF THE 40 MARKS INTO IMAGE'
 BEGIN
 OPERATION ADDMARK(INTEGER J, IMAGE I VARY)
         PROVIDED J>0&J<=40 YIELDS TRUE ONEXIT;
 SETMARKS(IMAGE I VARY):

        DECLARE INTEGER J;
        J:=0;
        WHILE J<40 DO
            (  J:=J+1;
               ADDMARK(J, I)).
 END
```

Figure 6.6

construction. The interpreter also allows a limited amount of interaction between the executing program and the programmer sitting at a terminal. Some provision is made for error checking and error reporting, the latter in a language appropriate to the error condition encountered.

For a number of reasons, but mainly that of time, the interpreter provided in the current implementation is incomplete and experimental. It was considered desirable for some form of interpreter to be provided within the system, particularly to demonstrate the feasibility of carrying out some program design evaluation before the level of the base machine had been encountered. Thus an interpreter providing some of the more unusual features was developed, whilst those features of a more mundane nature were either omitted or not developed completely.

The form of the command is:-

*execute machinename

The action invoked may be considered as "switching on the power" to the named machine. This machine will then carry out the action described by its program part, in general involving the invocation of other machines to carry out elaborations of any concepts it requires. (Recursive invocations of machines are handled in the obvious manner; recall that recursive definition of data types is not allowed). As implemented, the machine named in the command must be the initial ideal machine elaborating the "program" concept, although an obvious and attractive extension is to allow the command to apply to any machine. There would then, of course, be a need for some form of initialization of any operands.

6.3.1. The basic execution process

In a completely elaborated program, execution flow is similar to the flow of a program written in a contemporary programming language equipped with a procedure mechanism (e.g. Algol 60). As described in

section 5.2.5 the declare statements are regarded in a similar fashion to operations whose concern is the allocation and formatting of memory. If an operation is defined with a pre-condition or a post-condition these are checked prior to execution of the elaboration and after it respectively.

We will deal in more detail with the execution of programs when not all of the necessary machines have been designed and entered by the programmer. Three possible approaches are considered.

6.3.2. Simulation or temporary machines

A straightforward approach is one which clearly parallels the ideas of multi-level modelling (Zurcher and Randell 1968). The programmer includes in his design, "dummy" machines which merely simulate the necessary effect to produce acceptable results. Then, as design proceeds, each simulating machine is replaced by a proper machine designed to overcome the difficulty being simulated. Of course, this may involve the use of further dummy machines simulating the new set of primitive notions. The simulating machines can, of course, be powerful making use of any information which may be available. Aslanian and Bennett (1971) describe a system which provides a comprehensive set of simulation concepts which substantially increase the descriptive power available to the programmer. No such concepts are provided in Pearl.

However, the facilities provided in Pearl make it a reasonably simple matter to develop a program using dummy machines to allow test executions early in the development.

6.3.3. Programmer assistance

By reason of the interactive, online nature of Pearl, it is possible to make use of the programmer to supply values as the result of unelaborated operations. Use is made of such information as the type of the operands and whether they are vary or not before deciding whether

programmer assistance can be invoked. Figure 6.7 gives a short illustration.

The technique has a number of advantages and disadvantages. Among the advantages is the reduction in the amount of text that must be entered by the programmer in order to test a program. He does not have to explicitly code and enter machines (even "dummy" machines) to provide an "implementation" of operations which he has not properly designed. However, experience suggests that the programmer sitting at a terminal requires information about the state of a computation when prompted for values and is often surprised by what he is asked to do. This difficulty is, of course, closely tied to the problem of relating a program text to the actual program execution. It may be that better human engineering could alleviate the difficulties somewhat but there appear to be limitations to this interactive approach.

6.3.4. Using operation meanings

Consider the two machines described in figure 6.8. (It is assumed that no other machines exist). The obvious intention of the programmer is to construct a program which yields, as its result an object named "page" of type "image", which is in the state "printed". (This is the assertion supplied as the result of the initial machine "compfirst").

The command

*execute compfirst

activates "compfirst".

An object of type "image" with name "page" is created. As no representation is given, a standard one is used. Control now moves to carry out the operation "build" upon the conceptual object "page". First, the pre-condition is checked and found true. "Clearfirst" is now activated

```
++build
 display:'display values of a function of integers 0-9'
 begin operation f(integer x,integer y vary);

 program:
       declare integer (x,y);
       x:=0;
       while x<10 do
       ( f(x,y);
         writeint(y);
         x:=x+1).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
++execute program
 *** UNELABORATED OPERATION
 F(INTEGER X,INTEGER Y VARY)
 BEFORE OPERATION
 X WAS 0
 Y WAS 0
 PLEASE PROVIDE VALUES FOR
 Y
 72
 72
 *** UNELABORATED OPERATION

                .
                .
                .
```

Figure 6.7

```
+*build
 compfirst:'store image of page before printing'
 begin type image;
       states built(image i), printed(image i);
       operation
         build(image i vary)
           provided true yields built(i) onexit,
         print(image i)
           provided built(i) yields printed(i) onexit;

 program:
     declare image page;
     build(page); print(page).

 assert printed(page) onexit
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*build
 clearfirst:'expand build.we will empty the image first'
 begin states blank(image i);
       operation
        clear(image i vary)
          provided true yields blank(i) onexit,
        setmarks(image i vary)
          provided blank(i) yields built(i) onexit;

 build(image i vary):
     clear(i); setmarks(i).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*execute compfirst


 EXECUTION SUCCESSFUL
```

Figure 6.8

to carry out the elaboration for "build". The first action is to "clear" the object "page". The pre-condition is satisfied, but there is no machine available to carry out the operation. Thus it is performed symbolically using the post-condition of the definition of "clear" as the statement of the result of the operation. As a result the object "page" is deemed to satisfy the predicate

"blank (page)".

and the pre-condition of the next action, the operation "setmarks" is met. In a similar manner, there being no machine elaborating "setmarks", the object "page" will subsequently satisfy the predicate

"built (page)".

The action of "clearfirst" thus being completed, "compfirst" is resumed. A check is made that the elaboration of "build" was carried out successfully by evaluating the post-condition given in the definition of "build". The "print" operation is carried out in a similar, conceptual fashion. As a result it is determined that the object "page" is "printed" and thus the final assertion is met.

This example indicates how incomplete programs may be executed in a meaningful way with some expectation of discovering inconsistencies. Obviously there are limitations. Some are described in the next few sections, whilst others, possibly more far-reaching, are discussed in Chapter 7.

6.3.5. The use of meanings and states

In the example of section 6.3.4 we described how the post-condition of an operation definition could be used as a statement of its result. In the cases given, the actual post-conditions consisted of a single state. States are the only elements of post-conditions which may be used in this manner. Figure 6.9 illustrates one reason for this. (This figure is hypothetical for illustrative purposes).

```
++build
 n1:' ... '
 begin
   operation makezero(integer i vary)
     provided true yields i=0 onexit;
   operation generalop(integer i,integer (j,k) vary)
     provided i=0 yields i>j,j>k,k>i onexit;

 program:
       declare integer (a,b,c);
       makezero(a); generalop(a,b,c).
 end

            .
            .
            .

++execute n1
```

Figure 6.9

When "n1" is activated, it will follow its program and declare three integer variables a,b and c. An application of the "makezero" operation follows. Suppose that, as it is unelaborated, we make use of the post-condition to act as a statement of the result. Thus the integer variable a is given the value zero and execution of "n1" continues. The precondition of "generalop" is satisfied and an attempt is made to fulfill the post-condition. One possibility is to use the known value of a to determine a value for b so as to satisfy $a > b$. Thus if b is assigned the value -1, then this condition will be met. The second condition now requires $-1 > c$, and thus c is assigned the value -2. This results in the obvious contradiction $-2 > 0$ from the final condition. Of course it is not possible to choose a set of integer values to satisfy these conditions because of the theorem

$$i > j \;\&\; j > k \;\Rightarrow\; i > k$$

Unfortunately it would require an automatic theorem prover to discover whether a given post-condition could be satisfied at all.

The restriction of the use of post-conditions in this manner only to those post-conditions which are states, offers a partial solution to this problem. However, there are still a number of rules to be observed. The first we have hinted at above. If the post-condition of an unelaborated operation consists of more than one element, then the list of elements is processed left to right.

example:

if an operation has a post-condition as

. . . yields s(x), ¬ s(x) onexit,

then this would be treated as if it was

. . . yields ¬ s(x) onexit

despite its obvious contradictory nature.

The other rules are less trivial. The next section is devoted to a discussion of them.

6.3.6. Rules for the use of operation meanings in the execution of incomplete programs

(i) The pre-condition of an operation meaning is always a test.

(ii) The post-condition of an operation meaning is used either as a test, or as a statement expressing a result. The only element possessing this duality is the state. Whether a state is used as a test or as a statement is dependent upon whether or not the operation itself is elaborated (e) or unelaborated (u), and also whether the state itself is elaborated (e) or unelaborated (u). Figure 6.10 gives a table showing which particular use a state is put to. The table is described in terms of an operation b and a state bd where

operation b (. . .)

provided . . . yields bd (. . .) onexit

	b	bd	Test or statement
Case 1	u	e	test
Case 2	u	u	statement
Case 3	e	e	test
Case 4	e	u	statement

Figure 6.10

Neither case 2 nor case 3 from this table call for much comment. If both the operation b and the state bd are unelaborated, then the state acts as a statement; if both are elaborated the state is a test upon the consistency of the elaborations.

In case 1 the programmer has provided an elaboration of the state but not of the operation. Presumably the ultimate elaboration of b will reflect the given elaboration of bd. If the execution process was to interpret bd as a statement then, because bd has been elaborated, it would be necessary to ensure that the elaboration of bd was also true. It is not difficult to see how this could lead to either non-deterministic or contradictory situations.

example:

In section 6.3.5. an example was discussed to show why only states may exhibit the dual role of statement and test. This example is exactly similar to the situation we are now discussing when viewed as follows. Suppose a data type D is introduced together with a state S. S is used to define an operation F as:-

operation F (D x) provided . . . yields S(x) onexit

Subsequently D is elaborated as having 3 integers components i, j, k, whilst S is elaborated as:-

S(D x):

(i of x > j of x) & (j of x > k of x)

& (k of x > i of x)

An execution of a program invoking the unelaborated operation F(x) must not use the state S(x) as a statement because the elaboration of S(x) cannot possibly be satisfied.

In the absence of a tool able to resolve inconsistencies of the nature of the example, the obvious course is taken of insisting that an elaborated state always implies a test.

Case 4 of figure 6.10 arises when the operation has been elaborated but its meaning is still expressed at the higher, unelaborated level of description. It is therefore meaningless to test the state, as it should not have been changed as a result of the action caused by the operation elaboration. The state is thus interpreted as a statement.

Both case 1 and case 4 are good illustrations of the "close" conceptual relationship that exists between operations and states. (see section 5.2.6.3). These two cases are examples of the difficulties that are liable to arise if an operation is described at a different level to the state which defines it (or which is defined by it). The relationship between an operation and the states used to define its

meaning is similar to that which exists between a data type and the operations allowed upon instances of it. It will be appreciated, therefore, that individual machines do not represent individual levels of description; there will generally be several machines within that single level. The cases we have discussed above therefore represent not only the execution of a program not completely defined in terms of the base language, but also a view of such an execution even when closely related concepts of the program are themselves developed to differing degrees of detail.

6.3.7. Error reporting and debugging facilities

The structural nature of the set of machines enables the occurrence of any run-time errors to be meaningfully related to the original program text as described in Chapter 4. In the current implementation of the interpreter there are a few automatic checks carried out at run time. These include subscript checking and arithmetic overflow but not such features as ensuring a variable has been assigned before its value is used. In addition a number of features of the notation enable the programmer to specify explicitly that checks be made. (e.g. assert expressions, pre- and post-conditions on operations and elaborations etc.).

When an error is detected, the execution process provides certain information to the programmer. This consists of a message appropriate to the fault, followed by material indicating in which machine it occurred and at which point in its program. This is given in the form of a source listing with a pointer. Next the programmer is given a trace of machine activations so that he has some additional contextual material upon which to base his investigations. Finally the values of any pertinent variables are listed. Figure 6.11 shows an example.

The trace of machine activations and the textual pointer referencing a failing machine are particularly related to the structure of programs

```
+*build
 insert:'insert an integer value into a list'
 begin type integerlist;
       operation insertvalue(integerlist s vary,integer i);

 program:
       declare integerlist s;
       declare integer i;
       i:=0;
       while i<10 do
       ( i:=i+1;
         insertvalue(s,i)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*build
 listrep:'a list has 9 elements'
 begin

 integerlist:
       declare vector(9) integer element;
       declare integer count;
       count:=0.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*build
 listinsert:'insert according to vector representation'
 begin

 insertvalue(integerlist s vary,integer i):
       count of s:=count of s+1;
       element(count of s) of s:=i.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
+*execute program
 *** ERROR: SUBSCRIPT ERROR
 CURRENT MACHINE IS LISTINSERT EXECUTING OPERATION INSERTVALUE
 ERROR AT:
 COUNT OF S := COUNT OF S+1;
 ELEMENT(COUNT OF S) OF S := I.
                             |
 LISTINSERT WAS CALLED FROM INSERT EXECUTING OPERATION PROGRAM
 VALUE OF INDEX IS 10
 DECLARED SIZE IS 9
```

Figure 6.11

written in Pearl. The implementation does not provide either frequency counts (Satterthwaite 1972) or a complete correlation between the static and dynamic representations of a program. (Dijkstra 1968c). Both would be worthwhile and, probably, non-complex additions.

The system does not, as implemented, include the more attractive features of online debugging systems such as the interrogation of named variables or the program counters of machines. As is described in Chapter 4, features such as these can increase the understanding that a programmer has for a program and thereby raise his confidence in it.

Further comments related to possible extensions to the current Pearl implementation (and thereby to similar systems) are given in the next chapter.

6.4 Summary

This chapter has described those facilities of the Pearl system which enable modification, interrogation and evaluation of program designs.

Pearl represents an attempt to provide a unified environment for the development of computer programs. This environment is provided by means of a design notation in which a developing program may be described, together with a set of tools to help the programmer to a realization of a program in which he can place a high degree of confidence. Many of these tools are already available in contemporary computing systems, but Pearl additionally provides some which are unusual as well as exerting a particular influence over the complete design process. This influence may be traced from the earliest conception of a program design throughout, and even beyond, the normal life of the program itself. Pearl is able to do this because

it represents an information system for the total design process.

There are a number of points of discussion that the design of Pearl raises. It has deficiencies and weaknesses of its own besides being based upon some philosophies about program design that are, to say the least, the subject of considerable discussion. In the next chapter we attempt to give an evaluation of the experimental system including suggestions as to how noted deficiencies could possibly be corrected and the power of systems like Pearl extended. Also we discuss how Pearl as a system relates to contemporary programming systems.

Chapter 7:

Discussion and Conclusions

Experience in the use of Pearl has shed some light upon the contribution that such systems may make to the programming activity. We will discuss some points in this chapter, particularly those which we see as relevant in the light of likely future developments.

In this context, it is important to relate Pearl to certain other tools and techniques that are currently available to the programmer, or which have been proposed.

Finally we give some indication of the success of the current investigation. Several points of argument are raised, even in the underlying design philosophies. It is an important result of this work to decide whether decisions taken on the basis of these philosophies have been substantiated.

7.1 Some deficiencies and limitations of Pearl

In Chapter 5 (section 5.4) we described certain omissions and deficiencies in the notation of Pearl. We have said little about equivalent inadequacies in the system as a whole. Some of these have been recognized from the outset in that certain influences upon the design of the system (e.g. human engineering) were not catered for specifically, or that the experimental nature of the implementation made it impossible to include several desirable features. Other deficiencies have been revealed by use of the system.

7.1.1 Machines and levels of description

In Chapter 3 we developed the idea of a level of description being characterized by four sets of concepts (D, F, C and S). Pearl is built around the notion of allowing the representation of a program at a level of description chosen by the programmer. This freedom is provided by the . .

mechanism of the 'ideal' machine whereby the programmer specifies elements for the characterizing sets. There is, therefore, a very close correspondence between an ideal machine and a level of description. However, this correspondence is not one to one. A single machine (together with its environment) need not completely represent a single level of description. This is a natural result of the concept of a machine as carrying out a single elaboration. The specification part of each machine thereby introduces only those concepts necessary to carry out this elaboration, whether or not these concepts characterize a complete level of description in the sense that all related concepts are introduced.

It is arguable whether or not it would have been better to have made a one to one correspondence between machines and levels of description. For a number of reasons it was decided not to do this.

1. A machine represents a single design decision.
2. A machine was chosen as the basic unit of information in the system. As such it should be neither textually unwieldy or potentially complex. The uniform nature of all machines as the only building block was considered important.
3. At any stage of design, the programmer should not need to specify more than is necessary to represent a particular decision.
4. The system could take care of the need to gather together a set of machines representing a complete level of description.

Unfortunately, it is often the case that a programmer finds it necessary to introduce machines of not inconsiderable complexity (see for example the development shown in appendix E). Stoy and Strachey (1972) have remarked that in programming there is a certain requirement for a conjunction of autonomy and hierarchy. This is apparently reflected by the complexity of certain individual machines.

However, we believe that the concept of a machine is satisfactory in an environment such as Pearl, although it is possible that a smaller unit of information might be desirable. It would then be possible to re-investigate the relationships between the idea of a machine and the characterization of levels of description with a view to obtaining a closer correspondence.

There is a not inconsiderable problem here of course. The concept of a machine does provide a useful, self-contained, structural unit of design which enables the imposition upon the programmer of a significant design discipline. Smaller information units, whilst arguably having some advantages, have some potential dangers. It would be necessary to represent more relationships amongst such units than is the case with a machine. The programmer would have to be aware of these and the imposition of a satisfactory design discipline becomes a more difficult problem. Naturally this raises the question once more of how much freedom the programmer should be allowed. It must be said, however, that it is the author's belief that it is better to err on the side of too much restriction than too little, for programmers will generally abuse it if they are given too much.

7.1.2. Machine environments and design strategies

At any time in the development of a program, the environment available in a machine represents the information contained in the design as it has been developed to that point. A subsequent inspection of the environment will indicate that the design has progressed, but without indicating exactly how it has progressed. Thus a notion of progressive design is available from an inspection of the changing environment.

The strategy for design progress provided in Pearl is only one of a number of possibilities. In Chapter 3 (section 3.2.2.) we discussed a number of basic design strategies and concluded that one which allowed the

programmer to develop his program starting from the level of the problem description would have most advantages. The rules representing the design strategy of program development in Pearl are based upon this conclusion. Thus the environment available to a new machine represents the total design as developed and there are rules dictating how a new machine can interpret and modify this representation. The particular representation and set of rules has proved to be adequate in the context of the design strategy they encourage.

The particularly rigid discipline enforced by Pearl has proved to be of benefit in the programming task in that it has led to a better appreciation of difficulties inherent in the problem being tackled rather than problems of choice amongst possible representations in some programming language. As a result, the programmer is encouraged to try to understand what he is really trying to do, before he does it, rather than the more commonly encountered situation of the programmer trying to understand why he has done something already.

On occasion it has been found that the discipline is too rigid. In Pearl the level of the base language is fixed and all programs must be elaborated down to this level. There is no reasonable way by which the programmer can, even in a consistent fashion, raise the level of the base language concepts (i.e. a bottom-up strategy). Perhaps a system such as Pearl should be neutral as to the direction in which design proceeds. However, it should be noted that such a relaxation introduces a further degree of freedom into the design process, with consequent methodological issues raised. In Pearl it was decided to take a particularly rigid viewpoint and limit this freedom quite extensively. It will be appreciated that even in Pearl, there are situations where the programmer still has considerable

freedom of design.

example:

the number of operands in an operation, in particular the freedom to define an operation which alters the values of any number of its operands.

As alternatives which could be implemented in a system based upon Pearl the following are suggested for consideration.

1. Program development may additionally proceed in a bottom-up fashion. This is allowed by a relaxation of the rule that a machine may only elaborate a concept that was present in the environment before the machine was defined.

example:

A machine X could both introduce and elaborate a new concept Y.

If this extension is to discipline the programmer to a pure bottom-up development, then the elaboration of the concept Y must only make use of concepts which have a definite representation in the base language. This is probably an unnatural restriction. It would appear more reasonable to allow elaboration in terms of concepts which may or may not have a definite base language representation. The programmer may then develop programs, not only from the top or from the bottom, but also from the "middle" (up or down).

2. In suggestion 1, the design is still represented by a single environment. It is possible to change the representation of a design by adding structure to the environment. We can envisage a structure based upon scope rules for machines, so that only a sub-set of the total set of design concepts is available to any one new machine.

example:

Suppose there is a concept c present in the total environment. A new machine N may not wish to be told of this concept so that, although it has a requirement of a concept with the properties of c, it can, for instance, introduce a concept c with similar properties. Of course, the design may be such that the new machine N should not know of the existence of c. Such situations may arise in programs being developed by several people where the single, global environment of Pearl could have some disadvantages. Individual designers should be able to derive their own parts of the program in a manner which is unimpeded by others.

3. As an extension of 2, different representations of the same concept could be allowed to co-exist in the design. This could be of use as a means of archiving the development process, or for the implementation of versions developed by two or more programmers each influenced by their own design requirements, or of versions developed to cope with expected variations for program use, or to cater for program portability.

examples:

(a) It may be possible to develop a compiler such that by changing the representation of the concepts of code generation new compilers could be produced easily for different machines.

(b) Some decisions made during development of a program may be based upon predictions of input load. Versions allow different representations to be developed according to a set of differing predictions.

Such a scheme is suggested by Dijkstra (1972a) with the notion that a program is constructed by making a necklace of pearls chosen from amongst a set of potentially useful pearls. The ultimate extension to such a scheme is to allow the choice of a particular version of a representation to be left until the program is actually executing.

Experimentation with the 3 alternatives suggested (and others) may well be an interesting exercise. However, such experimentation is likely to be subjective according to individual programmers and situations and thereby be difficult to evaluate.

7.1.3. Extended notion of states

The concept of state as described in chapters 5 and 6 has proved to be a useful tool in the expression of the meanings of a conceptual operation. A state allows the programmer a means of writing down his intentions in a formal yet descriptive terminology which relies heavily upon an interpretation from natural language. The rigid nature of the syntax imposes a requirement for careful expression and contributes to considered design in much the same way as does the construction of assertions about a program (see Chapter 4).

The very usefulness of the technique draws particular attention to its other limitations and restrictions. There is therefore an attraction in investigating how these restrictions may be relaxed.

The basis for the restrictions lies in the desire to use operation meanings to define the effect of unelaborated operations at run time. The two main restrictions are:-

(i) Meanings should be completely deterministic, and

(ii) it is not possible to use states (reasonably) to express a relationship between operands of an operation.

Relaxation of restriction (i) introduces, in theory, combinatorial problems, as an operation may produce a number of different results, each of which must be considered in a state test of a program. Henderson and Quarendon (1974) describe some work in this area. Figure 7.1 however, gives an example of the additional power of expression which can be gained if the potential combinatorial explosion can be overcome.

The operation "read" has a result which is determinable only when supplied with additional information. The program illustrated in the figure is one which occurs frequently in practical programming situations and yet there is no way of representing it in Pearl without including a deterministic mechanism explicitly in the form of flags or other testable relationships.

Figure 7.2 illustrates the difficulties to be overcome if relationships between variables are to be represented. The specific problem is to determine whether or not the final assertion is satisfied. If not, then what relationship does hold between the cards a and b? On the other hand if the assertion is satisfied, then how is such a relationship maintained or, perhaps of greater importance, how can it be detected that some relationships cannot possibly be maintained? It is the author's belief that a programmer should be able to write down such things as non-deterministic results, or results which are relationships. In many cases there is no reason why this should not be possible provided the programmer is in full control of what he is doing. The difficulties arise because it is necessary to take precautions to warn a programmer when he has lost sight of his purpose. Again, it is a matter of deciding how much freedom programmers should be allowed.

```
type card;
type cardreader;
states readin(card c), eof(cardreader r);
operation read(card c vary, cardreader r)
provided ... yields readin (c)
              or eof(r) onexit;
program: declare cardreader r, card c;
         read(c,r) ;
         while ¬eof(r) do
         ( ...
           read(c,r) );
         ...
```

Figure 7.1

```
type card;
relation samecard(card(a,b));
operation read(card c vary);
operation copycard(card a vary, card b)
provided readin(b) yields samecard(a,b)
                   onexit;
program: declare card(a,b,c);
         read(a);
         copycard(b,a);
assert samecard(a,b) before
         read(c);
         copycard(b,c);
assert samecard(a,c)  ...
```

Figure 7.2

7.1.4. The "vary" mechanism

One concept that has been found wanting is the "vary" mechanism applicable to the operands of an operation. (see section 5.2.6.1).

The idea of an "invariant" as suggested by Hoare (1972b) has a great attraction as a more powerful mechanism which serves a similar purpose. The invariant allows the actual representation of a data object to change without affecting the abstraction of the object as used at a higher level of description. The vary mechanism prevents any such changes, whether or not they are visible to the higher level. It is not possible, therefore, to change the representation in any helpful manner during program execution (the "benevolent side effect" Hoare 1972b).

In order to allow an invariant to apply to a data object, it is necessary to provide a means for variable initialization. The condition of the invariant is then established before any operations may be carried out.

This may be done by an extension of the type statement as illustrated below. The example is based on one from Hoare (1972b).

```
type     smallintset invar limited;
states   limited (smallintset s);
```

The state "limited" is intended to refer to a bound on the number of elements in a "smallintset". The invar clause is equivalent to the post-condition of an operation statement. If an instance of a "smallintset" is declared in a machine program, then this instance should invariably satisfy the condition of being "limited". If an elaboration is given for "smallintset", then it is the responsibility of this elaboration to correctly ensure that the

appropriate operations are carried out so that instances of "smallintset" will be initialized to the state "limited". A check that this has been done may be carried out during execution in a straight forward way by using the invar clause as a test. For this test to be meaningful, it would be expected that the terms occurring in an invar clause would also be elaborated. Thus, "smallintset" might be elaborated as:

```
smallintset:  declare vector(100) integer A;
              declare integer m;
                  m: = 0.
```

and "limited" as:

```
limited (smallintset s): m of s >= 0 & m of s <= 100.
```

The test of the correct initialization of "smallintset" is thus expressed in terms of the elements of its representation.

If an elaboration of "smallintset" is not given, executions of programs declaring instances of "smallintset" may still be carried out by using the invar clause as a statement as is possible with the post-condition of an operation statement.

7.1.5. The assignment operator

The operation of the assignment of a value to a variable is a basic one in a wide class of contemporary programming languages. The base language of Pearl is a member of this class. It is fitting that this should be so due to the widespread use of languages of this class. The generalized use of the operation in Pearl and the way in which it is handled calls for a few comments.

Although the scheme that has been implemented is satisfactory (see section 5.2.4) it has drawn attention to a number of points. Firstly, the assignment operator symbol is unique amongst other operator symbols, in that its functional representation is context dependent. This can be defended on the grounds of familiarity of use and its common purpose, which is context

independent. This suggests that assignment should be treated as a well understood operation of the base language, much like subscription.

Assignment is a much more complex operation than subscription (which we may view as an operation on a type "address"). How an assignment is carried out is dependent upon the representation of its operands. This may not be sufficient information however. We can envisage situations where further information might be necessary, particularly if the notion of a type is parameterized in a manner equivalent to the parameterization of classes in SIMULA 67.

example:

```
table (integer n):
   declare integer size;
   declare vector (n) line 1;
        size: = n.
```

It is thus reasonable to allow the programmer to give elaborations for any assignment operator used between operands of a conceptual type. Thus the operation of assignment should, in general, be viewed as a conceptual operation just as any other introduced by the programmer.

In the implementation of Pearl the programmer may use the common symbol (:=) for all assignments, but must also provide elaborations for any assignment which may be invoked between operands which are not of a primitive type. This has led to a number of difficulties, particularly in the context of a program modification using the commands *replace or *delete. The usefulness of the flexible viewpoint has not been confirmed in practice and it is doubtful that any significant benefit was derived from the additional engineering effort required in the implementation.

7.1.6. Human Engineering

It is important that, in any system incorporating a human element, the interfaces between that human element and other parts of the system are well engineered. In an interactive system such as Pearl, potential users must not be deterred because of the form of their contact with the system. In the current implementation, whilst not disregarding this issue completely, it has not been explored to any depth. This was done consciously in the interests of limiting the task of implementation. Where possible some attempt has been made to make the system easy to use but on occasion human engineering has been neglected. This has had the effect that there are a number of examples where, in an environment other than one which is purely experimental, potential users might well at first find the system unattractive.

The notation itself is one such example. It was chosen for its simplicity and readability. The similarity between it and other high level languages is not coincidental. Many of the concepts we believe to be desirable are to be found in contemporary languages and it is thus appropriate to take advantage of well understood syntax. Unfortunately, such syntax can be verbose for a human being sitting at a terminal. The requirement for unique names leads to the invention of long identifiers (not in itself a bad thing) thereby adding to the overall textual length of programs.

It may well be that we should develop a language offering two different representations*. One representation is that used for program input (e.g. from a terminal), the other being used when the programmer wishes to inspect his program. Each representation would be derivable from the other in an automatic fashion.

The handling of errors discovered as a machine definition is entered is

*This idea was first communicated to the author by J.D. Ichbiah.

another source of frustration for the user of the current system. In an interactive system, it is particularly attractive to the user to be able to correct immediately errors discovered by the compiler. Because of the parsing method employed by the syntax checking routines of Pearl, this is not possible (see appendix C). Indeed it has been found most appropriate to terminate the checking of input completely once a non-trivial error has been encountered. Naturally this can be extremely frustrating for the user.

In order to make systems such as Pearl more attractive there are facilities which could be considered in addition to the removal of the sources of frustration noted above. Pre-eminent amongst these is an editing system allowing the user to edit existing machines in a microscopic fashion (in comparison to *replace or *delete), so that the large amount of textual input currently necessary may be reduced.

There are obviously many issues involved when the human engineering of an interactive system is considered (e.g. positioning of keys, type of function provided automatically etc). An appreciation of these may be gained from the work of Hansen (1971a, 1971b) or Mitchell (1970).

7.1.7. Miscellaneous

We close this discussion with a section on some other possible extensions to the current system.

An additional method of passing information between executing machines would often be useful. As a candidate for this we suggest a set of global variables, perhaps organized in subsets according to machines, with access controlled in a similar fashion to the access of operation, state or type names. However, it is necessary that the use of such global variables be obvious and restricted (maybe read only) to disallow obscure and complex relationships amongst machines. The class concept of SIMULA 67

could afford a possible solution. Unfortunately the scope rules of that language suffer from their development from those of Algol 60 which can be the source of much devious program construction. The named common areas of FORTRAN may be more appropriate, provided that additional restrictions are imposed to prevent misuse. A possible approach would be to use an explicit named set of variables (implemented by some machine) together with a statement of an invariant to hold over these variables.

It may be possible to develop an extension to Pearl which has a substantially different base language (e.g. one providing storage management primitives) or one which allowed generalization of the control structure elements in addition to operation and type.

The elements of the system itself can be expanded to provide, for instance further interrogation facilities or (as suggested earlier) editing and formal proving tools. Arguments can easily be made for any of these, but it is necessary to beware of allowing the system to become too large or too complex. One answer to this could possibly lie in making the system itself extendable so that a user could build up more complex facilities to satisfy his own requirements. This is a mechanism often seen in the command languages of interactive systems, particularly those of text editing systems. (van Dam and Rice 1971).

The system could also fruitfully be extended to include a mechanism for constructing efficient machine code programs to take full advantage of hardware. Indeed, this could possibly be combined with the interpretation techniques currently used as the basis of the program testing tools provided by the system. The testing and debugging system described by Satterthwaite (1972) is based upon the use of machine code rather than interpretation, whilst the incremental compilation techniques described by Mitchell (1970) are of obvious relevance.

Hopkins (1970) has suggested that, because of the redundancy of the information available in a structured development of a program, it may be possible to carry out a substantial amount of optimization when producing the executable code for the completed program. No work has been carried out on these lines within the implementation of Pearl.

7.2 The fallibility of Pearl - an example

It is appropriate to illustrate that even simple programming errors can be made and may pass undetected when using Pearl. This comment is possibly too strongly worded as the error which we shall describe was eventually discovered and certainly could have been discovered earlier, although for reasons we give below the programmer may be discouraged from making this possible.

In appendix E is shown an example of the design of a program to solve the eight queens problem as posed by Wirth (1971b). The relevant portions of this design are repeated in figure 7.3.

The concept behind the design of the program is that of moving a "pointer" over a "board" and testing the squares pointed at by the pointer. In machine M4, the pointer is represented by two integers, one to point at rows, and one to point at columns. The crucial error has been made by this choice of representation in terms of base language concepts, but more of that below. In M4G, the operation "regress" is elaborated in terms of this representation of a pointer. This involves the two operations "findqueen" and "removequeen", both of which use integer operands to identify the relevant square on the board. During the original design of this machine, the use of one of these operations was specified wrongly insofar as the logical correctness of the program was concerned. In fact the row and column pointers became interchanged. When the program was run it did not, of course, perform satisfactorily. Eventually the error was found by inspection and corrected.

```
m4:'now a pointer points to a column and a row'
begin

   pointer:  declare integer (row,col).
end

                      .
                      .
                      .

m4g:'we regress by using old information'
begin
      operation findqueen(board q,integer row vary,integer col);
      operation removequeen(board q vary,integer row,integer col);

regress(board q vary,pointer p vary):
      declare integer (i,j);
      j:=col of p;
      j:=j-1;
      if j>0 then
      ( findqueen(q,i,j);
        removequeen(q,i,j);
        if i=8 then
        (  j:=j-1;
           if j>0 then
           ( findqueen(q,i,j);
             removequeen(q,i,j))));
      col of p:=j;
      row of p:=i+1.
end
```

Figure 7.3

However, the error need never have occurred if the pointer had not been represented by two objects of the same type. If a pointer had been represented as a "rowpointer" and a "columnpointer", then any interchanged use would have been discovered by reason of the stringent type checking.

There is however a disincentive to carrying out the design in this way which it is particularly instructive to describe. If the design had been carried out according to the second alternative we described, than the two types "rowpointer" and "columnpointer" would each have separately required elaboration. Suppose a rowpointer was represented as an integer. Then the operation "findqueen" for example, which has both a columnpointer and a rowpointer as an operand will require elaboration before a columnpointer can be elaborated. The programmer must introduce a new operation (which we might call "newfindqueen") which now has a columnpointer and an integer as operands. Once he has done this for any similar operations he may give a representation for a columnpointer as an integer and elaborate "newfindqueen" etc. into operations having two integers as operands. The intermediate operations serve no purpose other than transforming one operand of the original set of operations in the next level of representation as a step towards the transformation of the complete set of operands. At each intermediate level, however, it is possible to perform a check upon what is written down although even at the lowest level there is an operation which has two integers as operands and so the error can be repeated. Hopefully the programmer will have a better understanding of what he is doing however. The notation will help as well, as one of these two operands should always appear in the context of the of operator.

The unfortunate disincentive is the large amount of text which needs to be input to prevent such errors occurring. As a human being carrying out the programming task, the author (who was the programmer at fault) was not prepared

to accept this additional work in return for such (seemingly) meagre benefit. The penalty was paid in full.

The moral is, of course, that no matter what tools are provided, the human user may be guaranteed to misuse them or to fail to appreciate their true worth.

7.3. Relationship with other tools and techniques

The Pearl system has much in common with other tools and techniques currently available, some of which have been already noted. This is to be expected as we do not claim to propose any new or startling technique to be applied in the programming task. What has been done is to look at the different tasks involved in programming and to select those approaches which are considered likely to make the whole programming task more comprehensible. The words whole and more in the last sentence are stressed because we have taken the view that programming covers more than the initial creation of a definitive piece of text. Rather, programming is an activity which encompasses the life of a program, from its conception to that time when all physical trace has been lost.

The Pearl scheme is very closely related to the ideas of "structured programming" as described by Dijkstra (1972a). However, Pearl is more rigid in the form that development may take, whilst the use of the computer allows not only a means of enforcing the discipline, but also a way of providing a powerful set of tools to actively assist the programmer during (and after) the development. Thus there is some reward for the programmer who follows the design discipline imposed upon him.

In Chapter 5 we discussed both SIMULA 67 (Dahl, Myhrhaug and Nygaard 1968) and ECL (Wegbreit 1971) as extensible languages. It would, of course, be possible to build a system similar to Pearl around a given extensible language encouraging the necessary design discipline that we regard as lacking in such languages. However, it is the author's belief that the necessary restrictions to apply such

a discipline would have a drastic effect upon the language.

In Chapter 6 we described how Pearl is related to the work of Zurcher and Randell in providing a scheme for the evaluation of incomplete program designs by test executions. Use has also been made of the notions of assertions about programs although more in the manner of maintaining or driving the program design than in generating verification conditions to be proved by an automatic theorem prover.

Although Pearl is designed specifically for the development of one program by a single programmer, it has several similarities with systems aimed at the problems of a team of people constructing a large piece of software. Pearson (1973) and Falla and Burns (1973) give outline descriptions of such systems. Both of these systems and Pearl rely upon the construction of a data base to represent a developing program. However, Pearl differs particularly in its emphasis upon the methodology of program construction. Both of the other cited systems are principally concerned with project control, although attention is paid to the structure of the resulting software at the level of individual module relationships.

The LC^2 system (Mitchell, Perlis and van Zoeren 1968) was designed to see how the computer could be of assistance in the top-down design of programs. It is an interactive system using program execution as the main source of design information. The programmer may enter program text in the form of "parts" which may be likened to procedures. If a "part" is discovered to be missing when execution takes place, the execution process gives the programmer the opportunity to enter the necessary text before resuming.

However, LC^2 gives no other assistance in the enforcement or encouragement of a discipline for design. The programmer has complete freedom to construct programs as he likes and there is, therefore, an equivalent methodological difficulty. LC^2 provides no mechanism at all for the testing of incomplete

programs. If he wishes to do this, then the programmer must provide, explicitly, executable temporary code to implement "parts" which are missing. (This is equivalent to a programmer in Pearl entering "dummy" machines as described in section 6.3.2.).

In a paper given at the I.F.I.P. conference in 1971 (Floyd 1971) there appears an example of a hypothetical man-machine interaction to construct a computer program. Floyd calls for the usual tools of syntax checker, code generator, program executor, prompter and file handler. In addition he suggests that the machine might continually check the consistency of the program against a set of specifications. This would involve a proof of the semantic correctness of the program, a proof of the termination of iteration, and counter-examples to incorrect programs. Of particular note in his example is the apparent hierarchical design strategy and the need for intelligible interaction between man and machine. The interactive program verifier described by Deutsch (1973) is based upon some of Floyd's proposals.

A further proposal is made by Freeman (1973). Freeman describes the tools and techniques that his system will provide as follows:

> ". . . an integrated programming environment . . . in which all the tools needed to develop a program are immediately available at the same level of control: editors, filing systems, compilers, debugging systems, I/O facilities; such a system is usually interactive".

Freeman takes functional programming (Freeman and Newell 1971) as the basis for program design. This scheme is again hierarchical in nature. Conspicuous by its absence in the list presented by Freeman is any tool concerned with checking the logical correctness of a program or part thereof from the text alone.

The implemented Pearl system is much closer to Freeman's proposals than

to Floyd's in that they both lack a complete logic checking tool. Pearl does, however, offer some such capability through the design of the notation and the static checks that are possible. It does not, however, go to such lengths with the debugging facilities as suggested by Freeman, although there is no reason why future developments should restrict themselves in this direction.

These proposals together with the systems described by Pearson (1973) and Falla and Burns (1973) are of particular interest in that they indicate that a unified approach to the process of program writing, development and maintenance is being more widely appreciated and also that there is considerable common ground.

7.4. Conclusions and summary

In the introduction to this thesis (Chapter 1) we expressed concern over the reliability of contemporary software produced using the tools and and techniques generally available. We have tried to investigate some of the causes of difficulty that arise in the programming task. At the bottom of many difficulties is the inherent lack of comprehension due to the complexity of both the problems to be solved and of the tools available for their solution.

The average programmer is unlikely to be able to obtain a sufficient grasp over both the problem and the available programming languages so as to be able to choose the best way of using the machine to solve the problem. Each individual programmer will develop his own way of doing things dependent upon his own experience, ability and environment. Unfortunately his natural resources tend to possess transients and so when he returns to a particular problem at a later date, he is often unable to recall how his program works. It is not really surprising that others subsequently have even greater difficulty.

The desire for powerful constructions is a major source of complexity in programming languages. In view of the cost of hardware this is to be expected. If the programming language removes much of the power of the hardware then it is likely that there will be questions of economics to be answered. Notwithstanding this point of view, we have suggested that the complexity of programming languages must be reduced. It does not follow that there will be a commensurate reduction in the power of the language, the opposite may even be true.

Although we may be able to lessen the impact of complex programming language constructions, there still remain the difficulties posed in the comprehension of real world problems and the evolution of satisfactory solutions. These are two processes which cannot be truly separated. Indeed programming is a particular form of problem solution. An understanding of the process of problem solving can act as a guide to how programs may be developed. We described the use of a generalized notation and particular design strategies to constrain and assist the program designer, using the ideas of a "level of description" and the relationships of "abstraction" and "elaboration".

Many of the conclusions at which we have arrived are, of necessity, subjective. However we believe that the arguments and suggestions put forward are well-founded as basic philosophies to be held about program development and design; only experience can show whether this is truly the case.

A particularly important requirement of these philosophies is that of restriction. We have already made a similar point about programming languages, but it is equally important that the programmer has only a limited set of things he can write down at any time in the program development. It should then be possible to understand program designs and to follow a constructive design strategy. This restriction must not be too oppressive or the programmer will find his natural inventiveness and creativity is hampered,

but it must not be too lax or else the programmer will tend to introduce complexity into his design through a lack of appreciation for the true source of difficulty. We have suggested that a hierarchic program structure, developed basically via a top-down strategy is a reasonable way for a design to be represented, whilst imposing a sufficient restriction to limit the degree of complexity. As the appropriate discussion has indicated, this may well be an over-cautious discipline. However, we must take care if we introduce any relaxations.

We have also suggested that the necessary restrictions are best imposed upon the programmer rather than being self-imposed. This may again be wrong in particular cases, but not, we believe, in general. How many programmers take as much care in the documentation and description of the design of a program as is seen in a recent paper by Naur (1972)? We suspect the number is very few. Yet this is the degree of discipline which is necessary and which, if not imposed by external means, must come from the programmer himself.

Pearl is a scheme which imposes a discipline throughout the design of a program. Although much of its worth comes from its attention to the textual development of a program in a well-structured way, this is only a part of the process of program development. The unification of many tools and techniques in a single environment is aimed at making the whole task of the comprehension of complexity in program design easier for the human being, be he the programmer or any other interested person. We have been able to combine in a single scheme, many techniques, ranging from the hierarchical development and representation of a program, schemes for specifying the intention and understanding of a programmer, facilities for programmer interrogation of designs and proposals, machine assistance in the maintenance of such information and means for checking its consistency, through to the

simple expedient of program development in a interactive system. However, it must be stressed that whilst all of these may make their own individual contribution, they are worth less if the total scheme is not based upon the philosophy of comprehension through simplicity, clarity and ease of use.

We certainly do not claim to have found the panacea for the problems of writing highly reliable software. Indeed the examples given may have been just as easily developed in a conventional way, or would they? Certainly in Pearl, the programmer is provided with means by which he can convey a large amount of information about his program and its design. Whether such a scheme is practical on a large scale program development can only be the subject of speculation. However, it is our hope and belief that most, if not all, of what has been said would apply and be applied with suitable modification in such circumstances.

Further research on the lines suggested by work with Pearl is now being carried out at Newcastle University under a grant from the Science Research Council. The major aim of this continued work is the construction of a further program building system which will additionally incorporate features which received little attention in Pearl. In particular, some effort is being devoted to the human engineering aspects of the new system to enable a closer evaluation of the acceptability of such systems to the programming community than was possible with Pearl.

APPENDICES

Appendices A,B and C contain details of the Pearl system. Appendix A gives a definition in B.N.F. of the Pearl notation. Appendix B details the various commands which a user may invoke, whilst Appendix C gives some notes on system implementation.

Appendices D,E and F show programs developed using Pearl. In each appendix, only the set of machines and a sample execution are included. The actual development of programs such as these additionally involves numerous other interactions between the programmer and Pearl. Text in lower case is entered by the user, whilst that in upper case is written by the system. The system invites the user to enter a command by typing a "+" sign.

Appendix A: Syntax of machine descriptions

The syntax of the notation used for describing machines is given below in Backus Naur Form. Any character or character string not enclosed in angular brackets (< >) is a terminal symbol. In addition <identifer> , <type name> , <number> and <string> are terminal symbols. An <identifier> and a <type name> are character sequences containing between 1 and 255 characters inclusive. The first character must be either a letter or one of the symbols _ # . The remainder may be chosen from these characters plus the digits 0-9. A <number> is a sequence of decimal digits whose value is an integer in the range 0 to 64035. A <string> is a sequence of characters (any characters) enclosed in single quotation marks. A quotation mark within a string is represented by two such marks. The sequence must not contain more than 255 characters.

References are made in the follow definition to notes which follow it.

PRODUCTION	NOTES
<machine> ::= <machine heading> <decision step>	
<machine heading> ::= <identifier> : <string>	1
<decision step> ::= begin <decision option> end	
<decision option> ::= <op elab> | <machine definition> <op elab>	
<machine definition> ::= <machine statement> <;> | <machine definition> <machine statement> <;>	2 2
<machine statement> ::= <type introduction> | <operation list> | <states list>	
<type introduction> ::= type <id spec>	
<id spec> ::= <identifier> | <identifier list> <identifier>)	

PRODUCTION	NOTES
<identifier list> ::= (| <identifier list> <identifier> ,	
<operation list> ::= <operation start> <op specif> |<operation list> , <op specif>	
<operation start> ::= operation	
<op specif> ::= <operation> | <operation> <provided> <valued expression> <result>	14
<operation> ::= <identifier> | <identifier> <parameter list>	
<parameter list> ::= <parameter head> <parameter element>)	
<parameter head> ::= (| <parameter head> <parameter element> ,	
<parameter element> ::= <typing element> | <typing element> vary	

<typing element> ::= <t name> <id spec>
  | vector <t name> <id spec>
  | <head> <size> <t name> <id spec>
	
	3
	4

<head> ::= vector (	
<size> ::= <valued expression>)	14
<t name> ::= <type name>	
<provided> ::= provided	
<result> ::= <yields> <valued expression> onexit	14
<yields> ::= yields | assert	 5
<states list> ::= <states> <op specif> | <states list> , <op specif>	
<states> ::= states	
<op elab> ::= <descrip> : <pre-block>. | <descrip> : <pre-block> . <result>	

PRODUCTION	NOTES
<descrip> ::= <operation>	6,7
| <type name>	8
<pre-block> ::= <block>	
| <entry> <block>	
<entry> ::= <provided> <valued expression> then	14
<block> ::= <expression>	
| <expression> <;> <block>	
<;> ::= ;	2,12
<expression> ::= <valued expression>	
| <valueless statement>	
<valueless statement> ::= <declaration>	12
| <repeat head> <valued expression>	12,14
| <while head> <expression>	12
| <if clause> <expression>	12
<declaration> ::= <declare> <typing element>	9
| <declaration> , <typing element>	
<declare> ::= declare	
<repeat head> ::= <repeat> <expression> until	
<repeat> ::= repeat	
<while head> ::= <while> <valued expression> do	14
<while> ::= while	
<valued expression> ::= <logical expression>	
| <logical expression> := <valued expression>	10,12
| <if clause> <true part> <valued expression>	11,12
<if clause> ::= <if> <valued expression> then	14
<if> ::= if	
<true part> ::= <valued expression> else	
<logical expression> ::= <logical factor>	
| <logical expression> | <logical factor>	12
<logical factor> ::= <logical primary>	
| <logical factor> <conop> <logical primary>	

PRODUCTION	NOTES
<conop> ::= & | ,	12 12
<logical primary> ::= <string expression> | <string expression <relation> <string expression> | ¬<logical primary> | true | false	
<relation> ::= = | < | > | < = | > = | ¬ =	
<string expression> ::= <arithmetic expression> | <string expression>||<arithmetic expression>	
<arithmetic expression> ::= <term> | <arithmetic expression> - <term> | <arithmetic expression> + <term> | - <term> | + <term>	
<term> ::= <primary> | <term> * <primary> | <term> / <primary>	
<primary> ::= <basic primary> | <assertion> <basic primary>	
<assertion> ::= <assert><valued expression>before	12,14
<assert> ::= assert	
<basic primary> ::= <variable> | <(> <block>) | <constant>	9,12
<(> ::= (	
<constant> ::= <number> | <string>	
<variable> ::= <name> | <qualifier> <name>	
<qualifier> ::= <name> of	

PRODUCTION	NOTES
<name> ::= <identifier> | <subscript head> <valued expression>)	12,13,14
<subscript head> ::= <identifier> <(> | <subscript head> <valued expression> ,	

Notes:

1. Any errors made up to this point are recoverable.
2. There are certain additional non-terminals which are necessary for code emission purposes. The code emitted by the compiler (called $\frac{1}{2}$ way code) is interpreted both for listing and for execution.
3. A ⟨typing element⟩ of this form must not be used in a ⟨declaration⟩ .
4. A ⟨typing element⟩ of this form must not be used in a ⟨parameter element⟩ .
5. This form of ⟨yields⟩ is used when giving the post-condition for a ⟨pre-block⟩ .
6. The elaboration of states also takes this form.
7. If the operation or state being elaborated was introduced with parameters different from those given, the system will make the correction or insertion and inform the user.
8. There is a particular problem following the elaboration of a data type concerning the names used for the components of that data type. It is possible that there may be a clash between these and a name of a formal parameter of an operation or state which is elaborated as a result of the elaboration of the data type. This is only discovered when the operation or state is elaborated. It is necessary to change the name of the formal parameter.

9. The scope of variable names is the block in which they are declared except in the case when a data type is elaborated. In this last case, the names of variables declared in the outermost block are available to machines elaborating any operations or states upon the elaborated data type.

10. The type of the ⟨logical expression⟩ and the ⟨valued expression⟩ must be the same and not "undefined" (i.e. the type should be either a primitive type or a user defined conceptual type. An "undefined" type covers all other cases).

11. The type of the ⟨true part⟩ and the ⟨valued expression⟩ must be identical. This type is the type of the whole alternative valued expression and may be "undefined".

12. In post-conditions for operation definitions there are a number of restrictions.

    (i) Conjunction is specified by , rather than &.

    (ii) Parentheses for blocks may only be used within either arithmetic or string expressions.

    (iii) The following may not be used.

    ; repeat while declare

    := if | assert

    (iv) Operations may not be invoked.

13. Subscripts start from 1.

14. In these ⟨valued expression⟩ 's assignments (or use of operations with vary parameters) may only apply to variables local to the ⟨valued expression⟩ .

Appendix B: Commands

There are 7 commands available.

(i) *initialize

This command initializes the data base representing a program. Initialization consists of a machine called "system" plus the following data types and operations.

Data types: - integer, string

Operations:

program	-	an unelaborated operation.
writeint (integer i)	-	to write the value of i.
nlcr	-	to give new line and carriage return character to output device.
prsym (integer i)	-	print a symbol corresponding to the byte value of i on the next available character position of the output device.
readint (integer i vary)	-	read an integer value into i.
substr (string s vary, string t, integer (i,j))	-	assign to s characters i to j inclusive from t (i,j) = 1).

(ii) *build

This command invokes the routines which enable the input of a new machine. The description of the machine follows the command.

(iii) *replace ⟨machine name⟩

This command replaces the machine named with the machine whose description follows. In addition to the form required by *build there are other restrictions on replacement machines.

(a) All of the concepts introduced by the original machine must be re-introduced, at least in name. New operations and states may also be introduced but not (in the current implementation), new types.

(b) The formal parameters of a re-introduced operation must agree both in number and in type (by position) with the formal parameters of the original. It is possible to change the identifier of a formal parameter. It is also possible for an operand of a re-introduced operation to be given the attribute vary, even though this attribute was not present in the original. Removal of the vary attribute for an operand is not allowed. The meaning part of re-introduced operations may also be changed. However, the old meaning should imply the new meaning. The system will request confirmation of this if not in batch mode. Meanings may be added where they were not previously present.

(c) The formal parameter of a re-introduced state should agree in type with the original.

(d) The environment of the replacement machine will be the same as for the machine being replaced. However, additional restrictions are imposed on the choice of names for new concepts to prevent clashes with names present in any later environment.

(e) The concept elaborated by the replacement machine must be the same as that elaborated by the original.

If the concept elaborated is a data concept, then the system will immediately require replacements for all machines which were originally dependent upon the original representation of this data concept.

If desired, it is possible to replace a machine by itself. Instead of providing a replacement machine, the word "leave" may be used.

```
*replace X
leave
```

(iv) *delete ⟨machine name⟩

This command causes the deletion of the named machine and of all machines which are dependent upon it and upon them etc.

(v) *list ⟨option⟩

This command provides means for the retrieval of information from the data-base according to the stated option.

⟨option⟩ :: = ⟨class⟩
| ⟨identifier⟩
| all
| choice

⟨class⟩ :: = ⟨class type⟩
| ⟨class type⟩ full

⟨class type⟩ :: = machines
| operations
| types
| states

Listing a ⟨class type⟩ results in a summary of those objects present in the data-base of the named ⟨class type⟩ . If the keyword full is appended, complete listings are given. The *list ⟨identifier⟩ option gives a "full" listing of information about the named object if such an object is present in the data-base.

The option all is equivalent to requests to list each ⟨class type⟩ without full.

The user can discover if his choice of elaborations is limited in any way following elaboration of a data type (or replacement of a machine elaborating a data type) by the command *list choice.

A full listing of a machine or elaborated concept uses an automatic

indentation algorithm to lay out a program in a neat manner. This can be useful as the layout used on the program input is therefore immaterial.

(vi) *execute <option>

This command causes the execution of a program.

<option> :: = <machine name>
| program

It is necessary that the <machine name> names the machine which elaborates the concept "program".

A description of the execution mechanism is given in Chapter 6.

(vii) *quit

This command terminates the session for the user. Any relevant information is written to the data-base and held on backing store to enable continuation at a later date.

In addition to these commands, an interrupt function is available which will terminate action of any command at an appropriate moment consistent with non-violation of data-base information.

Abbreviated forms of these commands are allowed (e.g. *init for *initialize, *exec for *execute etc.).

Appendix C: Some notes on the implementation of Pearl

The Pearl system has been implemented in an experimental fashion to run under the M.T.S. operating system at the University of Newcastle upon Tyne. This implementation is based upon the existence of two major pieces of software.

The first is the XPL compiler generator system, (McKeeman, Horning and Wortman 1970) which has been used to construct the processor for the input of machine descriptions. The XPL system encourages the construction of such a processor using the XPL programming language. Programs written in this language are compiled into object modules which require a loader of their own. Normally this loader is part of an interface tailored for the particular operating system being used. This interface provides the XPL program with system dependent facilities such as storage control and input/output handling. In Pearl, the opportunity was taken to develop such an interface to provide for the overlaying of XPL programs and to greatly enhance the standard file handling facilities available. These file handling facilities are formed from the second major piece of software which has been utilized. This consists of a set of routines, collectively known as the Newcastle File Handling System (Cooke and Gray 1973), which allow for the construction and manipulation of complex, tree-like data structures which may be stored on disk files. Much use is made of such structures to hold the design information of a program with its complex relationships.

Part of the interface between the XPL program and the operating system is controlled by the user. The commands he supplies determine which particular function of the system will be loaded into the overlay area. All of the major functions of the system are written as XPL

programs which communicate with the file routines and the user via the interface program. The interface program is written in a combination of 360 machine language and PL360.

The interruption handler of Pearl utilizes a feature of the M.T.S. operating system which allows user programs to handle particular forms of interrupt. Using this feature it is a relatively trivial matter to return control to the user interface routine with a request for another command. It is also possible to delay the acceptance of such an interrupt, so that the system is able to ensure that the information held about a program design remains consistent.

The total design structure is represented by a number of interrelated tree structures. Individual trees are used for machines, types and operations, whilst states are stored as part of the tree representing types. The program code for a machine is kept separately from the description of the machine itself, but referenced directly from the machine tree. This code ($\frac{1}{2}$-way code) is in a reverse-Polish form and is such as it may be used to drive an execution process or to regenerate the original source. Symbol tables are additionally required for this latter purpose. The $\frac{1}{2}$-way code contains several operations which are common to both the listing interpreter and the execution interpreter. It thus makes it a comparatively simple process to pinpoint an erroneous statement found by the execution interpreter in the original source listing. This code is also used to retain operation meanings.

The functions invoked by the various commands are combined into 3 separate overlays written in XPL. That for *build and *replace combined in one such program occupies 50 K bytes of code (71 K including data and variables). The interpreter (*execute) is a second, separate program of 23 K bytes of code (55 K) whilst all of the remaining functions are combined into the third program. This has a total of 22 K bytes of code from a total

size of 32 K bytes.

The actual interface program (written largely in PL360) and the set of utility functions (written in 360 Assembler) which provide the file handling facilities require a further 108 K bytes including a large in-core data area of more than 35 K bytes. The whole system at present, including the necessary file buffers, requires approximately 190 K bytes of core storage. This figure could be reduced by limiting the size of the data areas.

Appendix D

This appendix shows a set of machines developed to construct a program for a problem described by Dijkstra (1972a).

A program is to be constructed which will print 20 lines numbered from top to bottom by a y-coordinate running from 20 through to 1. The position of characters on a line is given by an x-coordinate running from 1 to 20. For each of the 40 positions given by

$$x = fx(j) \text{ and } y = fy(j) \text{ for } 1 <= j <= 40$$

a mark has to be printed; all other positions on the page are to be blank.

(This problem is changed from that given by Dijkstra in the magnitude of the dimensions of the page and of the number of marks to be placed).

```
          PEARL PROGRAM WRITING SYSTEM
 COMMANDS MAY BE ENTERED NOW
+*init
 DONE
+*build
 compfirst:'store image of page before printing'
 begin type image;
       states built(image i), printed(image i);
       operation
        build(image i vary)
         provided true yields built(i) onexit,
        print(image i)
         provided built(i) yields
         printed(i) onexit;

 program:
   declare image page;
   build(page); print(page).

 assert printed(page) onexit

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 clearfirst:'expand build. we will empty the image first'
 begin states blank(image i);
       operation
        clear(image i vary)
         provided true yields blank(i) onexit,
        setmarks(image i vary)
         provided blank(i) yields built(i) onexit;

 build(image i vary):
   clear(i); setmarks(i).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

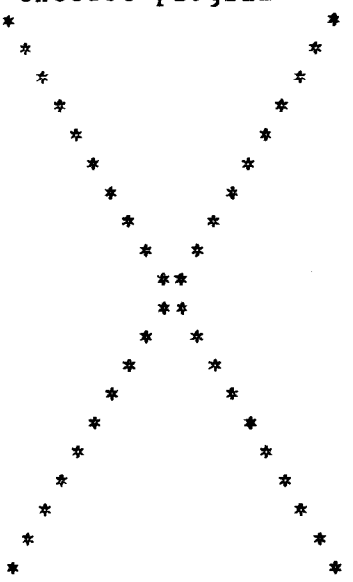
```
+*build
 jscanner:'setmarks. put each of the 40 marks into image'
 begin operation
        addmark(integer j, image i vary)
         provided j>0 & j<=40 yields
         true onexit;

 setmarks(image i vary):
   declare integer j;
   j:=0;
   while j<40 do
    ( j:=j+1; addmark(j,i) ).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 comppos:'calculate the position of the jth mark'
 begin states validx(integer x), validy(integer y);
       operation
        f(integer (x,y) vary, integer j)
         provided j>0 & j<=40 yields
         validx(x), validy(y) onexit,
        markpos(integer (x,y), image i vary)
         provided validx(x) & validy(y) yields
                  true onexit;

 addmark(integer j, image i vary):
   declare integer (x,y);
   f(x, y, j);
   markpos(x,y,i).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 function:'an example of a possible function for f'
 begin

 f(integer (x,y) vary, integer j):
   x:=if j<21 then j else j-20;
   if j>20 then y:=j-20
   else y:=21-j.

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 liner:'an image is a vector of lines called l'
 begin type line;

 image:
   declare vector(20) line l.

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 liner1:'we print an image by printing its lines'
 begin states lineprinted(line l), linebuilt(line l);
       operation
        lineprint(line l)
         provided linebuilt(l) yields
         lineprinted(l) onexit;

 print(image i):
   declare integer j;
   j:=21;
   while j>1 do
    ( j:=j-1; lineprint(l(j) of i) ).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 liner2:'clear out the image line by line'
 begin states blankline(line l), markinline(line l);
       operation
        lineclear(line l vary)
         provided true yields
         ¬markinline(l), blankline(l) onexit;

 clear(image i vary):
   declare integer j;
   j:=0;
   while j<20 do
    ( j:=j+1;
      lineclear(l(j) of i) ).

 en'
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 liner3:'x is a position on the yth line of the page'
 begin operation
        linemark(integer x, line l vary)
         provided true yields
         markinline(l), ¬blankline(l) onexit;

 markpos(integer (x,y), image i vary):
   linemark(x,l(y) of i).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 liner4:'an image is printed if its bottom line is'
 begin

 printed(image i): lineprinted(l(1) of i).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 liner5:'an image is blank if its last line is'
 begin

 blank(image i): blankline(l(20) of i).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 liner6:'an image is built when its last line is built'
 begin

 built(image i): linebuilt(l(20) of i).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 linerel1:'explain relation between linebuilt and other states'
 begin

 linebuilt(line l): markinline(l) | blankline(l).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 longrep:'a line is simply a vector of 20 symbols(integers)'
 begin

 line: declare vector(20)integer symb.

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 longrep1:'print line by using prsym'
 begin

 lineprint(line l):
   declare integer j;
   j:=0;
   while j<20 do
    ( j:=j+1; prsym(symb(j) of l) );
   nlcr.

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 longrep2:'linemark. put a mark in symb(x) of line'
 begin

 linemark(integer x, line l vary):
   symb(x) of l := 92.

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 longrep3:'clear line completely to blanks'
 begin
 lineclear(line l vary):
   declare integer j;
   j:=0;
   while j<20 do
    ( j:=j+1; symb(j) of l := 64).

 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*execute program
```



```
 EXECUTION SUCCESSFUL
+*quit
```

Appendix E

This appendix shows a set of machines based upon the program developed by Wirth (1971b) to find 1 solution to the 8-queens problem. The program described here does not follow that developed by Wirth in all respects, particularly at the higher levels of description.

```
          PEARL PROGRAM WRITING SYSTEM
 COMMANDS MAY BE ENTERED NOW
+*init
 DONE
+*build
 m1:' find a solution to 8 queens problem'
 begin
 type board;
 type pointer;
 states full(board q);
 states toofar(pointer p), offbottom(pointer p);
 operation settofirstsquare(pointer p vary);
 operation trysquare(pointer p, board q, integer safe vary);
 operation putonsquare(board q vary, pointer p);
 operation moveonfornextqueen(pointer p vary);
 operation moveonforthisqueen(pointer p vary);
 operation regress(board q vary, pointer p vary);
 operation print(board q);
 operation clear(board q vary);
 operation failure;
 program:

       declare board q, pointer p, integer safe;
       clear(q);
       settofirstsquare(p);
       repeat
        (  repeat
            (  trysquare(p, q, safe);
               if safe then
                  (  putonsquare(q, p);
                     moveonfornextqueen(p))
               else
                  moveonforthisqueen(p))
           until full(q)|toofar(p);
           if ¬full(q) then
              regress(q, p))
       until full(q)|offbottom(p);
       if full(q) then
          print(q)
       else
          failure.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 m2:'we appreciate 1 queen per column'
 begin
 operation settofirstoffirst(pointer p vary);
 settofirstsquare(pointer p vary):

       settofirstoffirst(p).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m3:'see m2'
 begin
 operation movetofirstofnext(pointer p vary);
 moveonfornextqueen(pointer p vary):

       movetofirstofnext(p).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4:'now a pointer points to a column and a row'
 begin
 pointer:

       declare integer (row, col).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4a:'as a result of m4'
 begin
 settofirstoffirst(pointer p vary):

       row of p:=1;
       col of p:=1.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4b:' see m4a; note possible overflow'
 begin
 movetofirstofnext(pointer p vary):

       col of p:=col of p+1;
       row of p:=1.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 m4c:'see m4a; note possible overflow'
 begin
 moveonforthisqueen(pointer p vary):

       row of p:=row of p+1.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4d:'see m4a; trysquare related to coords'
 begin
 operation trycoord(integer row, integer col,
                    board q, integer safe vary);
 trysquare(pointer p, board q, integer safe vary):

       trycoord(row of p, col of p, q, safe).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4e:'see m4a; mapping straight to coords'
 begin
 operation putoncoord(board q vary, integer row, integer col);
 putonsquare(board q vary, pointer p):

       putoncoord(q, row of p, col of p).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4f:'we may go over row'
 begin
 toofar (pointer p):

       row of p>8.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 m4g:'we regress by using old information'
 begin
 operation findqueen(board q, integer row vary, integer col);
 operation removequeen(board q vary, integer row,
                       integer col);
 regress(board q vary, pointer p vary):

       declare integer (i, j);
       j:=col of p;
       j:=j-1;
       if j>0 then
          ( findqueen(q, i, j);
            removequeen(q, i, j);
            if i=8 then
               ( j:=j-1;
                 if j>0 then
                    ( findqueen(q, i, j);
                      removequeen(q, i, j))));
       col of p:=j;
       row of p:=i+1.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m4h:'we may fall off only in columns'
 begin
 offbottom (pointer p):

       col of p<1.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m5:'a board:  position of queens + squares covered'
 begin
 board:

       declare integer numberon;
       declare vector(8)integer x;
       declare vector(8)integer a, vector(15)integer (b, c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 m5a:'a board is full when there are 8 queens on it'
 begin
 full (board q):

       numberon of q=8.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m5b:'printing is trivial'
 begin
 print(board q):

       declare integer i;
       i:=0;
       while i<numberon of q do
          (  i:=i+1;
             writeint(x(i) of q)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m5c:'no queens and no blockages'
 begin
 clear(board q vary):

       declare integer i;
       numberon of q:=0;
       i:=0;
       while i<8 do
          (  i:=i+1;
             a(i) of q:=true;
             b(i) of q:=true;
             c(i) of q:=true);
       while i<15 do
          (  i:=i+1;
             b(i) of q:=true;
             c(i) of q:=true).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m5d:'using these auxiliaries we can easily compute solution'
 begin
 trycoord(integer row, integer col, board q, integer safe vary):

       safe:=a(row) of q&b(row+col-1) of q&c(row-col+8) of q.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 m5e:'as m5d'
 begin
 putoncoord(board q vary, integer row, integer col):

       x(col) of q:=row;
       numberon of q:=numberon of q+1;
       a(row) of q:=false;
       b(row+col-1) of q:=false;
       c(row-col+8) of q:=false.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m5f:'finding a queen in given column is easy'
 begin
 findqueen(board q, integer row vary, integer col):

       row:=x(col) of q.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m5g:'and so is removing it'
 begin
 removequeen(board q vary, integer row, integer col):

       a(row) of q:=true;
       b(row+col-1) of q:=true;
       c(row-col+8) of q:=true;
       numberon of q:=numberon of q-1.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 m6:'a failure report for m1'
 begin
 failure:

       writeint(999).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*exec program
 1
 5
 8
 6
 3
 7
 2
 4
 EXECUTION SUCCESSFUL
+*quit
```

Appendix F

In Chapters 3 and 5 a problem is described whereby 10 input cards are to be checked for certain properties (see section 3.2.1. or 5.2.2.). This appendix contains a completed program for that problem.

```
          PEARL PROGRAM WRITING SYSTEM
 COMMANDS MAY BE ENTERED NOW
+*init
 DONE
+*build
 cardprocessor:'read each card, and then process it'
 begin
 type cardimage;
 operation read(cardimage c vary);
 operation process(cardimage c);
 program:

       declare cardimage c;
       declare integer i;
       i:=0;
       while i<10 do
          ( i:=i+1;
            read(c);
            process(c)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 processor:'check the values and the check'
 begin
 operation checkcard(cardimage c, integer ck vary);
 operation successmessage;
 operation rejectmessage;
 operation writeout(cardimage c);
 process(cardimage c):

       declare integer ok;
       checkcard(c, ok);
       writeout(c);
       if ok then
          successmessage
       else
          rejectmessage.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 checker:'check the values, then and only then, the check'
 begin
 operation checkvalidity(cardimage c, integer ok vary);
 operation checkcheck(cardimage c, integer ok vary);
 checkcard(cardimage c, integer ok vary):

       checkvalidity(c, ok);
       if ok then
          checkcheck(c, ok).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 cardrep:' a card is 9 data values and a check'
 begin
 type value;
 cardimage:

       declare vector(9)value data;
       declare value check.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 cardreader:'reads in the 10 values'
 begin
 operation readvalue(value v vary);
 read(cardimage c vary):

       declare integer i;
       i:=0;
       while i<9 do
          (  i:=i+1;
             readvalue(data(i) of c));
       readvalue(check of c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 cardwriter:'writes out values'
 begin
 operation writevalue(value v);
 writeout(cardimage c):

       declare integer i;
       nlcr;
       i:=0;
       while i<9 do
          ( i:=i+1;
            writevalue(data(i) of c));
       writevalue(check of c).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 validitycheck:'checks the 9 values'
 begin
 operation checkvalue(value v, integer ok vary);
 checkvalidity(cardimage c, integer ok vary):

       declare integer i;
       i:=0;
       ok:=true;
       while i<9&ok do
          ( i:=i+1;
            checkvalue(data(i) of c, ok)).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 checkchecker:'make sure check value is satisfied'
 begin
 operation combine(value v vary, value w);
 operation comparevalue(value u, value v, integer ok vary);
 checkcheck(cardimage c, integer ok vary):

       declare value temp;
       declare integer i;
       i:=1;
       temp:=data(i) of c;
       while i<9 do
          ( i:=i+1;
            combine(temp, data(i) of c));
       comparevalue(temp, check of c, ok).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 valuer:'values are integers in this case'
 begin
 value:

       declare integer valueof.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 realreader:'values may thus be easily read in'
 begin
 readvalue(value v vary):

       readint(valueof of v).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 validvaluer:'values in the range 0 to 99'
 begin
 checkvalue(value v, integer ok vary):

       ok:=valueof of v>0&valueof of v<100.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 realwriter:'writing values is writing integers'
 begin
 writevalue(value v):

       writeint(valueof of v).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 combiner:'combine is an addition process'
 begin
 combine(value v vary, value w):

       valueof of v:=valueof of v+valueof of w.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
+*build
 checksumer:'checking is purely arithmetic'
 begin
 comparevalue(value u, value v, integer ok vary):

       ok:=(valueof of v=valueof of u).
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 assignment:'assignment of values'
 begin
 value_assign(value value1 vary, value value2):

       valueof of value1:=valueof of value2.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 successwriter:'give "o.k."'
 begin
 successmessage:

       nlcr;
       prsym(214);
       prsym(75);
       prsym(210);
       prsym(75);
       nlcr.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.

+*build
 failurewriter:'give "error"'
 begin
 rejectmessage:

       nlcr;
       prsym(197);
       prsym(217);
       prsym(217);
       prsym(214);
       prsym(217);
       nlcr.
 end
 END OF CHECKING
 NO ERRORS WERE DETECTED.
```

```
**execute cardprocessor

 10
 15
 30
 1
 16
 8
 26
 33
 3
 142

 O.K.


 11
 23
 14
 8
 7
 12
 90
 17
 64
 241

 ERROR


 22
 33
 50
 -5
 77
 13
 17
 20
 46
 283

 ERROR
```

```
77
63
25
14
36
26
82
91
100
424

ERROR


42
13
26
18
91
1
22
81
17
311

O.K.


39
47
29
10
61
41
93
8
26
334

ERROR


66
23
42
85
96
83
2
7
12
426

ERROR
```

```
 23
 49
 9
 13
 25
 13
 31
 41
 99
 303

 O.K.,


 18
 76
 8
 31
 47
 27
 72
 62
 83
 424

 O.K.


 16
 51
 26
 16
 23
 68
 85
 45
 2
 232

 ERROR

 EXECUTION SUCCESSFUL
+*quit
```

References

C. Alexander 1966:
'Notes on the Synthesis of Form'.
Harvard University Press, Cambridge, Mass., 1966.

Algol W 1972:
'Algol W Programming Manual'.
Computing Laboratory, University of Newcastle Upon Tyne, June 1972.

C.D. Allen and C.B. Jones 1973:
'The Formal Development of an Algorithm'.
IBM United Kingdom, Research Report TR.12.110, March 1973.

E. Ashcroft and Z. Manna 1971:
'Formalization of properties of parallel programs'.
in Machine Intelligence 6, B. Meltzer and D. Michie (eds),
Edinburgh University Press, 1971 pp. 17-42.

R. Aslanian and M. Bennett 1971:
'Evolutive modelling and evaluation of operating and computer systems'.
Research report CA-016, Compagnie Internationale pour l'Informatique,
France 1971.

F.T. Baker 1972:
'Chief programmer team management of production programming'.
IBM Systems Journal No. 1, Vol. 11 (1972) pp. 56-73.

J.W. de Bakker 1969:
'Semantics of programming languages'.
in Advances in Information System Science, J.T. Tou (ed), Vol. 2
(1969) pp. 173-228.

R.M. Balzer 1969:
'EXDAMS - extendable debugging and monitoring system'.
AFIPS Spring Joint Computer Conference 1969 pp. 567-580.

D.W. Barron 1971:
'Programming in wonderland'.
Computer Bulletin No.4, Vol. 15 (1971) p. 153.

D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon and C. Strachey 1964:
'The Main Features of CPL'.
Computer Journal Vol. 6 (1964) pp. 134-143.

R. Bayer, D. Gries, M. Paul and H.R. Wiehle 1967:
'The ALCOR Illinois 7090/7094 Post Mortem Dump'.
Communications of the ACM No. 12, Vol. 10 (Dec. 1967) pp. 804-808.

L.A. Belady and M.M. Lehman 1971:
'Programming system dynamics or the meta dynamics of systems in
maintenance and growth'.
IBM Research Report RC 3546, Sept. 17th 1971.

F.C. Bequaert 1968:
'QUIP - A system for automatic program generation'.
AFIPS Fall Joint Computer Conference 1968 pp. 611-616.

G.M. Birtwistle 1973:
'SIMULA - its features and prospects'.
in High Level Programming Languages - the way ahead,
Proceedings of a Conference held at the University of York 1972,
N.C.C. Publications, Sept. 1973 pp. 85-100.

H.M. Brown 1970:
Presentation given at a Conference in Rome 1969. See Software Engineering Techniques, J.N. Buxton and B. Randell (eds), 1970 pp. 53-60.

J.N. Buxton and B. Randell 1970:
'Software Engineering Techniques'.
Report on a Conference sponsored by the NATO Science Committee,
Rome 1969, published 1970.

B.L. Clark and J.J. Horning 1971:
'The System Language for Project SUE'.
SIGPLAN Notices No. 9, Vol. 6 (Oct. 1971) pp. 79-88.

M. Clint 1970:
'An Approach to Floating-Point Function Theory'.
Report of Queen's University, Belfast 1970.

R. Conway and D. Gries 1973:
'An Introduction to Programming - a structured approach using PL/I and PL/C'.
Winthrop Publishers Inc., Cambridge, Mass., 1973.

M. Cooke and W.A. Gray 1973:
'A Redesigned Record Structure for the Newcastle File Handling System'.
Program, No. 1, Vol. 7 (Jan. 1973) pp. 1-23.

O-J. Dahl and C.A.R. Hoare 1972:
'Hierachical program structures'.
in Structured Programming, O-J Dahl, E.W. Dijkstra and C.A.R. Hoare,
Academic Press, London 1972.

O-J. Dahl, B. Myhrhaug and K. Nygaard 1968:
'SIMULA 67 Common Base Language'.
Publication No. S-2, Norwegian Computing Centre 1968.

A. van Dam and D. Rice 1971:
'On-line text editing: A survey'.
Computing Surveys No. 3, Vol. 3 (1971) pp. 93-114.

L.P. Deutsch 1973:
'An interactive program verifier'.
Ph.D. thesis, University of California, Berkeley.
Xerox Corporation Report No. CSL-73-1 May 1973.

E.W. Dijkstra 1968a:
'A constructive approach to the problem of program correctness'.
B.I.T. Vol. 8 (1968) pp. 174-186.

E.W. Dijkstra 1968b:
'The structure of the T.H.E. multiprogramming system'.
Communications of the ACM No. 5, Vol. 11 (1968) pp. 341-346.

E.W. Dijkstra 1968c:
'Goto statement considered harmful'.
Letter to the editor, Communications of the ACM No. 3, Vol. 11 (1968) pp. 147-148.

E.W. Dijkstra 1968d:
Reply to a letter of J.R. Rice.
Communications of the ACM No. 8, Vol. 11 (1968) pp. 538, 541.

E.W. Dijkstra 1970:
'Structured Programming'.
in Software Engineering Techniques, J.N. Buxton and B. Randell (eds), 1970.

E.W. Dijkstra 1972a:
'Notes on Structured Programming'.
in Structured Programming, O-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, London 1972.

E.W. Dijkstra 1972b:
'The Humble Programmer'.
Communications of the ACM No. 10, Vol. 15 (1972) pp. 859-866.

E.W. Dijkstra 1973:
'A Simple Axiomatic Basis for Programming Language Constructs'.
Report EWD372-0, Technological University, Eindhoven 1973.

W.G.P. Dutton and C.S. Minto 1971:
'PM3 - an automatic program generator'.
in Software 71, Proceedings of a Conference held at the University of Kent at Canterbury 1971. Transcripta Books 1971 pp. 143-146.

E.W. Elcock, J.M. Foster, P.M.D. Gray, J.J. McGregor and A.M. Murray 1971:
'ABSET: A programming language based on sets: motivation and examples'.
in Machine Intelligence 6, B. Meltzer and D. Michie (eds), Edinburgh University Press 1971, pp. 467-492.

B. Elspas, M.W. Green, K.N. Levitt and R.J. Waldinger 1972:
'Research in interactive program proving techniques'.
Stanford Research Institute 1972.

B. Elspas, K.N. Levitt, R.J. Waldinger and A. Waksman 1972:
'An assessment of techniques for proving program correctness'.
Computing Surveys No. 2, Vol. 4 June 1972 pp. 97-147.

D.C. Engelbart and W.K. English 1968:
'A research centre for augmenting human intellect'.
AFIPS Fall Joint Computer Conference 1968 pp. 395-410.

C. Engelman 1968:
'MATHLAB 68'.
Proceedings of the I.F.I.P. Congress, Edinburgh 1968 pp. B91-B95.

A.P. Ershov 1972:
'Aesthetics and the Human Factor in Programming'.
Datamation No. 7, Vol. 18 (1972) pp. 62-67.

T.G. Evans and D.L. Darley 1966:
'On-line debugging techniques: a survey'.
AFIPS Fall Joint Computer Conference 1966 pp. 37-50.

A.D. Falkoff 1970:
'Criteria for a system design language'.
in Software Engineering Techniques, J.N. Buxton and B. Randell (eds), 1970.

M.E. Falla and D. Burns 1973:
'Software Development Systems'.
Datafair 73 Conference papers, Vol. 1, Business Papers,
British Computer Society 1973 pp. 166-173.

R.W. Floyd 1967a:
'Assigning Meanings to Programs'.
A.M.S. Symposium in Applied Maths. Vol. 19, 1967 pp. 19-32.

R.W. Floyd 1967b:
'Non-deterministic Algorithms'.
Journal of the ACM No. 4, Vol. 14 (Oct. 1967) pp. 636-644.

R.W. Floyd 1971:
'Towards interactive design of correct programs'.
Proceedings of the I.F.I.P. Congress, Ljubljana 1971 pp. 11-14.

P. Freeman 1973:
'Functional programming, testing and machine aids'.
in Program Test Methods, W.C. Hetzel (ed), Prentice-Hall,
Englewood Cliffs 1973 pp. 49-56.

P. Freeman and A. Newell 1971:
'A model for functional reasoning in design'.
Report CMU-CS-71-107, Carnegie Mellon University 1971.

S. Gill 1969:
'Thoughts on the sequence of writing software'.
in Software Engineering, P. Naur and B. Randell (eds), 1969.

E.L. Glaser 1971:
'Introduction and overview of the LOGOS project'.
Case Western Reserve University, Oct. 1971.

H.H. Goldstine and J. von Neumann 1947:
'Planning and coding problems for an electronic computing instrument'.
in John von Neumann, Collected Works Vol. 5. Pergamon Press 1963 p. 80.

D.I. Good 1970:
'Toward a man-machine system for proving program correctness'.
Ph.D. thesis, University of Wisconsin 1970.

D.I. Good and L. Ragland 1973:
'NUCLEUS - A language of provable programs'.
in Program Test Methods, W.C. Hetzel (ed), Prentice-Hall, Englewood
Cliffs 1973 pp. 93-117.

R.M. Graham, G.J. Clancy Jnr. and D.B. DeVaney 1973:
'A software design and evaluation system'.
Communications of the ACM, No. 2, Vol. 16 (Feb. 1973) pp. 110-116.

W.J. Hansen 1971a:
'Creation of hierarchic text with a computer display'.
Ph.D. thesis, Stanford University 1971.

W.J. Hansen 1971b:
'User engineering principles for interactive systems'.
AFIPS Fall Joint Computer Conference 1971 pp. 523-532.

P. Henderson and P. Quarendon 1974:
'Finite state testing of structured programs'.
Colloque sur la Programmation, CNRS, Paris 1974.

P. Henderson and R.A. Snowdon 1972:
'An Experiment in Structured Programming'.
B.I.T. Vol. 12, (1972) pp. 38-53.

W.C. Hetzel 1973:
'Principles of Computer Program Testing'.
in Program Test Methods, W.C. Hetzel (ed), Prentice-Hall Inc. 1973 pp. 17-28.

I.D. Hill 1972:
'Wouldn't it be nice if we could write computer programs in ordinary English - or would it?'.
Computer Bulletin No. 6, Vol. 16 (June 1972) pp. 306-312.

C.A.R. Hoare 1961:
'Algorithm 65 Find'
Communications of the ACM No. 7, Vol. 4 (1961) p. 321.

C.A.R. Hoare 1969:
'An axiomatic basis for computer programming'.
Communications of the ACM No. 10, Vol. 12 (1969) pps. 576-580, 583.

C.A.R. Hoare 1971a:
'Proof of a program: FIND'.
Communications of the ACM No. 1, Vol. 14 (1971) pp. 39-45.

C.A.R. Hoare 1971b:
'Procedures and Parameters: an axiomatic approach'.
in Symposium on Semantics of Algorithmic Languages, A. Dold and B. Eckmann (eds), Springer-Verlag 1971.

C.A.R. Hoare 1972a:
'Notes on Data Structuring'.
in Structured Programming, O-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, London 1972.

C.A.R. Hoare 1972b:
'Proof of correctness of data representations'.
Acta Information Vol. 1, (1972) pp. 271-281.

C.A.R. Hoare and N. Wirth 1972:
'An axiomatic definition of the programming language PASCAL'.
Eidg. Technische Hochschule, Zurich, Berichte der Fachgr, Computer Wissenschaften Nr. 6, November 1972.

M.E. Hopkins 1970:
'Computer aided software design'.
in Software Engineering Techniques, J.N. Buxton and B. Randell (eds) 1970, pps. 99-101.

M.E. Hopkins 1972:
'A case for the GOTO'.
Proceedings of the ACM Conference, Boston 1972 pp. 787-790.

A.M. Hormann 1970:
'Planning by man-machine synergism: a characterization of processes and environment'.
System Development Corporation, report SP-3484 March 1970.

T.E. Hull, W.H. Enright and A.E. Sedgwick 1972:
'The correctness of numerical algorithms'.
Proceedings of an ACM Conference on proving assertions about programs, Las Cruces, New Mexico 1972 pp. 66-73.

IBM 1969:
'IBM System/360 Operating System Assembler Language'.
Form C28-6514-6 IBM Corporation, White Plains, New York 1969.

ICL 1971:
'Modular Programming Techniques'.
ICL Dataskil, publication 5092, 1971.

P. Johansen 1967:
'Non-deterministic Programming'.
B.I.T. Vol. 7 (1967) pp. 289-304.

H.B. Keller 1968:
'Numerical methods for two point boundary value problems'.
Blaisdell, Waltham Mass., 1968.

J.C. King 1969:
'A Program Verifier'.
Ph.D. thesis, Carnegie-Mellon University 1969.

D.E. Knuth 1968:
'The Art of Programming: Volume 1'.
Addison-Wesley, 1968.

A. Koestler 1964:
'The Act of Creation'.
Hutchinson, London 1964.

H.G. Kolsky 1969:
'Problem Formulation using APL'.
IBM Systems Journal No. 3, Vol. 8 (1969) pp. 204-219.

H.C. Lauer 1972:
'Correctness in Operating Systems'.
Ph.D. thesis, Carnegie-Mellon University 1972.

B.M. Leavonworth 1972:
'Programming with(out) the GOTO'.
Proceedings of the ACM Conference, Boston 1972 pp. 782-786.

C.H. Lindsey and S.G. van der Meulen 1971:
'Informal introduction to ALGOL 68'.
North Holland Publishing Company, Amsterdam 1971.

R.L. London 1972:
'The current state of proving programs correct'.
Proceedings of the ACM Conference, Boston 1972 pp. 39-46.

P. Lucas, P.E. Lauer and H. Stigleitner 1968:
'Method and notation for the formal definition of programming languages'.
Technical report TR 25-087, IBM Laboratory Vienna, June 1968.

M.L. Mannheim 1966:
'Hierarchical structure: a model of design and planning processes'.
M.I.T. Report No. 7, M.I.T. Press, Cambridge, Mass., 1966.

J. McCarthy 1966:
'A formal description of a subset of Algol'.
in Formal Language Description Languages for Computer Programming,
T.B. Steel Jnr, (ed), North Holland Publishing Company,
Amsterdam 1966 pp. 1-12.

H. McDaniel 1970:
'Applications of Decision Tables'.
Brandon/System Press Inc. 1970.

W.M. McKeeman, J.J. Horning and D.B. Wortman 1970:
'A Compiler Generator'.
Prentice-Hall, Englewood Cliffs 1970.

H.D. Mills 1970:
'Syntax-directed documentation for PL360'.
Communications of the ACM No. 4, Vol. 13 (1970) pp. 216-222.

H.D. Mills 1971:
'Top down programming in large systems'.
in Debugging Techniques in Large Systems, R. Rustin (ed), Prentice-Hall,
Englewood Cliffs 1971 pp. 41-55.

J.G. Mitchell 1970:
'The design and construction of flexible and efficient
programming systems'.
Ph.D. thesis, Carnegie-Mellon University 1970.
(Excerpts given as lecture notes by A.J. Perlis at a summer school
held in Marktoberdorf, W. Germany 1971).

J.G. Mitchell, A.J. Perlis and H.R. van Zoeren 1968:
'LC^2: A language for conversational computing'.
in Interactive systems for experimental applied mathematics,
M. Klerer and J. Reinfelds (eds), Academic Press, New York 1968.

P. Naur 1966:
'Proof of algorithms by generalized snapshot'.
B.I.T. Vol. 6, (1966) pp. 310-316.

P. Naur 1969:
'Programming by Action Clusters'.
B.I.T. Vol. 9, (1969) pp. 250-258.

P. Naur 1972:
'An Experiment in Program Development'.
B.I.T. Vol. 12, (1972) pp. 347-365.

P. Naur and B. Randell 1969:
'Software Engineering'.
Report on a Conference sponsored by the NATO Science Committee,
Garmisch 1968, published 1969.

J. Palme 1972:
Letter to the editor, Computer Journal No. 1, Vol. 15, Feb. 1972,
pp. 4,36.

D.L. Parnas 1972:
'A technique for software module specification with examples'.
Communications of the ACM No. 5, Vol. 15 (1972) pp. 330-336.

D.L. Parnas and J.A. Darringer 1967:
'SODAS and a methodology for system design'.
AFIPS Fall Joint Computer Conference 1967 pp. 449-474.

D. Pearson 1973:
Articles describing the CADES system, Computer Weekly, July 26th,
August 2nd, August 9th 1973.

G. Polya 1945:
'How to solve it'.
Princeton University Press. 1945.

B. Randell 1971:
in Efficient production of Large Programs, Computation centre of the
Polish Academy of Sciences 1971 pp. 36-37.

J.R. Rice 1968:
'The goto statement reconsidered'.
Letter to the editor, Communications of the ACM, No. 8, Vol. 11 (1968)
p. 538.

A.J. Rose 1966:
'The use of APL for describing programs at many levels of detail'.
IBM Research Report RC 1700, Oct. 1966.

D.T. Ross 1967:
'The AED approach to generalized computer-aided design'.
Proceedings of the ACM National Meeting 1967, pp. 367-385.

D.T. Ross 1969:
'Introduction to Software Engineering with the AED-0 language'.
Report ESL-R-405 M.I.T. DSR project No. 71425, 1969.

J.E. Sammet 1969:
'Programming Languages: History and Fundamentals'.
Prentice-Hall Inc., Englewood Cliffs 1969.

E.H. Satterthwaite 1972:
'Debugging Tools for High Level Languages'.
Software: Practice and Experience No. 3, Vol. 2, (1972) pp. 197-217.

A.L. Scherr 1973:
'Developing and Testing a Large Programming System, OS/360 Time Sharing Option'.
in Program Test Methods, W.C. Hetzel (ed), Prentice-Hall Inc., Englewood Cliffs 1973 pp. 165-180.

D.T. Schmidt and T.F. Kavanagh 1970:
'The Use of Decision Tables'.
in Applications of Decision Tables, H. McDaniel (ed) Brandon/System Press Inc. 1970.

J.I. Schwartz 1970:
'Analyzing large-scale system development'.
in Software Engineering Techniques, J.N. Buxton and B. Randell (eds), 1970 pps. 122-137.

H.A. Simon 1969:
'The Sciences of the Artifical'.
M.I.T. Press, Cambridge, Mass., 1969.

T.B. Steel 1966:
'Formal Language Description Languages for Computer Programming'.
North Holland Publishing Company, Amsterdam 1966.

J.E. Stoy and C. Strachey 1972:
'OS6 - An experimental operating system for a small computer. Part 1: General principles and structure'.
Computer Journal No. 2, Vol. 15 (1972) pp. 117-124.

W. Teitelman 1970:
'Towards a programming laboratory'.
in Software Engineering Techniques, J.N. Buxton and B. Randell (eds), 1970 pp. 137-149.

A.M. Turing 1949:
'Checking a Large Routine'.
in Report on a Conference on High Speed Calculating Machines, University Mathematical Laboratory, Cambridge 1949, pp. 67-68.

R.J. Waldinger and R.C.T. Lee 1969:
'PROW - A step toward automatic program writing'.
Proceedings of the First International Joint Conference on Artifical Intelligence, Washington D.C. 1969.

B. Wegbreit 1971:
'The ECL programming system'.
AFIPS Fall Joint Computer Conference 1971 pp. 253-262.

G.M. Weinberg 1971:
'The Psychology of Computer Programming'.
Van Nostrand Reinhold, New York 1971.

L.L. Whyte 1969:
'Structural Hierarchies: A challenging class of physical and biological problems'.
in Hierarchical Structures, L.L. Whyte, A.G. Wilson and D. Wilson (eds), American Elsevier, New York 1969 pp. 3-16.

A. van Wijngaarden 1966:
'Recursive definition of syntax and semantics'.
in Formal Language Description Languages for Computer Programming, T.B. Steel Jnr, (ed), North Holland Publishing Company, Amsterdam 1966 pp. 13-24.

A. van Wijngaarden (ed), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster 1969:
'Report on the Algorithmic Language ALGOL 68' Numerische Mathematik Vol. 14 (1969) pp. 79-218.

M.V. Wilkes 1968:
'The outer and inner syntax of a programming language'.
Computer Journal No. 3, Vol. 11 (1968) pp. 260-263.

N. Wirth 1968:
'PL360, a programming language for the 360 computers'.
Journal of the ACM No. 1, Vol. 15 (1968) pp. 37-74.

N. Wirth 1971a:
'The programming language PASCAL'.
Acta Informatica Vol. 1 (1971) pp. 35-63.

N. Wirth 1971b:
'Program development by step-wise refinement'.
Communications of the ACM No. 4, Vol. 14 (1971) pp. 221-226.

N. Wirth and H. Weber 1966:
'EULER: A generalization of Algol and its formal definition: part 2'.
Communications of the ACM No. 2, Vol. 9 (1966) pp. 89-99.

M. Woodger 1971:
'On semantic levels in programming'.
Proceedings of the I.F.I.P. Congress, Ljubljana, 1971, pp. TA-3-79 to TA-3-83.

W.A. Wulf 1972:
'A case against the GOTO'.
Proceedings of the ACM Conference, Boston 1972 pp. 791-797.

W.A. Wulf, D.B. Russell and A.N. Habermann 1971:
'BLISS: A language for Systems Programming'.
Communications of the ACM No. 12, Vol. 14 (1971) pp. 780-790.

W. Wulf and M. Shaw 1973:
'Global Variable Considered Harmful'.
SIGPLAN Notices No. 2, Vol. 8 (1973) pp. 28-34.

F. Zurcher and B. Randell 1968:
'Iterative multi-level modelling - a methodology for computer system design'.
Proceedings of the I.F.I.P. Congress, Edinburgh 1968 pp. D138-D142.