Decentralised Control Flow

A Computational Model for Distributed Systems

David H. Mundy

Computing Laboratory
March 1988

University of Newcastle upon Tyne

ABSTRACT

This thesis presents two sets of principles for the organisation of distributed computing systems. Details of models of computation based on these principles are given, together with proposals for programming languages based on each model of computation.

The recursive control flow principles are based on the concept of recursive structuring. A recursive control flow computing system comprises a group of subordinate computing systems connected together by a communications medium. Each subordinate computing system may either be a computing system which consists of a processing unit, a memory component, and some input/output devices, or is itself a recursive control flow computing system. The memory components of all the subordinate computing systems within a recursive control flow computing system are arranged in a hierarchy. Using suitable addresses, any part of the hierarchy is accessible to any sequence of instructions which may be executed by the processing unit of a subordinate computing system. This global accessibility gives rise to serious difficulties in the understanding of the meaning of programs written in a programming language based on the recursive control flow model of computation. Reasoning about a particular program in isolation is difficult because of the potential interference between the execution different programs cannot be ignored.

The alternative principles, decentralised control flow, restrict the global accessibility of the memory components of the subordinate computing systems. The concept of objects forms the basis of the principles. Information may flow along unnamed channels between instances of these objects, this being the only way in which one instance of an object may communicate with some other instance of an object. Reasoning about a particular program written in a programming language based on the decentralised control flow model of computation is easier since it is guaranteed that there will be no interference between the execution of different programs.

CONTENTS

ACKNOWLEDGMENTS

The starting point for the work reported in this thesis is that of Treleaven and Hopkins [Treleaven and Hopkins, 1981], and this thesis forms a contribution to the continuing research work of the Computer Architecture Group within the Computing Laboratory at the University of Newcastle upon Tyne. I owe much to the various members of that research group. In particular, thanks are due to Phil Treleaven, Richard Hopkins, and Isabel Gouveia Lima with whom I worked closely. The work has benefitted from technical discussions with different members of the Computing Laboratory - David Brownbridge, Martin McLauchlan, and Jon Garnsworthy, to mention but three.

Thanks are also due to those people who, directly or indirectly, helped me to produce this thesis. Professor Brian Randell and Professor Peter Lee both read drafts of this thesis and wrote helpful criticisms. Lesley Mundy suggested simplifications to my complex sentence construction, and also found most of the typographical errors.

Finally, and by no means least, I am indebted to Professor Brian Randell for the encouragement to pursue this work in a lively research environment, and to both the Science and Engineering Research Council of Great Britain and International Computers Limited for their financial support as a graduate student and as a research associate.

1 INTRODUCTION

Computing systems have been in widespread use for the past thirty years but, despite much popular misconception to the contrary, a radical machine capable of thought has been neither designed nor built. The first computing systems were seen, by their users at least, as nothing more than a further development of electronic calculators and indeed were first used for this purpose. Another early use of computing systems was to tabulate the behaviour of ballistic missiles for the armed services. Much of the interest in designing and building computing systems stemmed from the desire to remove the tedium and error involved in the production of mathematical tables. Resources were spent on the design of algorithms for the solution of existing problems and the subsequent encoding of these algorithms as sequences of instructions to be executed by the early computers. The history of the development of these early computing systems has been amply traced by Randell [Randell, 1973] and will not be considered in greater detail here.

It was not long before it was appreciated that complex systems other than those of the mathematician or the scientist could be modelled by sequences of instructions and executed by the processing unit of some computing system. Between the late 1950's and early 1960's many financial and accounting programs were written. There was also a rapid development of programming languages aimed at data processing applications [Willey, d'Agapeyeff, Tribe, Gibbens, and

Clark, 1961]. However, in that period, no effort was given over to research in the separate areas of architectures for computing system, models of computation, and programming languages.

To this day the design principles underlying the architecture of the first computing system and the first model of computation dominate computing science. These principles are often referred to as the von Neumann style [Gouveia Lima, Hopkins, Marshall, Mundy, and Treleaven, 1983], thereby associating both the architecture and the model with the pioneering work of von Neumann and his control flow computing systems [Goldstine, 1972]. These computing systems were constructed from a memory component, some input/output devices, and a processing unit. A sequence of instructions was stored in the memory component, along with the information processed by that sequence of instructions. The processing unit fetched each instruction of the sequence in turn from the memory component, along with any information required, caused the instruction to be executed, and stored any information generated by the execution of the instruction back into the memory component. Each instruction performed some 'simple' operation on the global state of the computing system. Examples of the instructions characteristic of the processing units found in the von Neumann style architecture would include those used to perform arithmetic operations on two numeric quantities and those used to transfer information between the processing unit and the memory component or input/output devices. The order in which the instructions were executed was determined by their order in the

sequence. The input/output devices could be used to store information for longer periods of time, in particular, during the time when the computing system was not in use.

At the outset, each computing system existed in isolation from all other computing systems. Therefore, during the execution of a sequence of instructions, there was neither co-operation nor communication between different computing systems. The information stored within one computing system could not be accessed by another computing system, except by the physical movement between the computing systems of some portable storage device, such as a magnetic disc or tape. The use of networking initially through standard telephone equipment together with modems, and then with various forms of wide or local area networks, enabled connections to remote computing systems to be established. Information could then be transmitted between computing systems. However, this capability did not necessarily permit the user to write two sequences of instructions such that one sequence of instructions executed on one computing system co-operated with the execution of the other sequence of instructions on a different computing system. Typically, the co-operation between the different computing systems occured at a low level, within the control program which controls the behaviour of the separate computing systems, and could not be exploited by an individual user.

The von Neumann model of computation is based upon the design of the von Neumann style architecture of computing systems and is basically a description of the behaviour of such computing systems. The essential concepts of the von Neumann model of computation are the sequence of instructions which are executed individually in order, and the memory component in which information may be stored. The close relationship between the design of the von Neumann style architecture and the von Neumann model of computation is reflected in the apparent ease with which programming languages based on the von Neumann model of computation are implemented on a von Neumann style architecture. For example, both the programming languages Lisp and Fortran reflect details of the IBM computing systems for which they were first designed. However, the von Neumann model of computation has been refined over the years to incorporate several ideas which have become fundamental to the discipline of computing science. An example of this refinement process may be traced from the introduction of high level programming languages such as Algol 60 [Naur, 1963], through the development of structured programming techniques [Dahl, Dijkstra, and Hoare, 1972] to the concept of abstract data types [Liskov, Snyder, Atkinson, and Schaffert, 1977]. These developments have been fed back into the design of von Neumann style computing systems. An example of this process is the ICL 2900 range of computing systems [Buckle, 1978] which was designed explicitly to support efficiently high level programming languages based on revisions of the von Neumann model of computation.

Thirty years have elapsed since the development of the programming language Fortran, the first example of a programming language based on the von Neumann model of computation which is still in widespread use. Despite this general acceptance of the von Neumann model of computation, both on the part of the manufacturers of computing systems and the users, many researchers within the Computing Science community are re-assessing the usefulness of the von Neumann model of computation. The uses to which von Neumann architecture computing systems have been put have increased dramatically since the introduction of mini- and micro-computers in the 1970's. The majority of programs for these systems have been written in programming languages based on the von Neumann model of computation. By the mid 1970's many Computing Scientists recognised that a crisis point in the development of software had been reached, in that the existing programming languages and techniques for the construction of large complex programs were not sufficient for the production of high quality correct software. One of the drawbacks to programming languages based on the von Neumann model of computation is the memory component. The individual cells of the memory component may be thought of as variables which may be assigned different values as execution of a program proceeds. With this interpretation of the memory component it is difficult to reason formally about the behaviour of programs written in a programming language based on the von Neumann model of computation [Turner, 1982; Wadge and Ashcroft, 1983]. The meaning of an expression in a program can be changed through assignment to the variables which appear in that expression.

The other drawback is that many of the programming languages based on the von Neumann model of computation are defined only informally. In fact a formal definition of some of these programming languages would be large, difficult to understand, and, in all probability, of little practical use [Backus, 1978]. It is these drawbacks which have motivated two contrasting areas of research.

One group of researchers has concentrated on introducing refinements to the von Neumann model of computation. Some have produced von Neumann style programming languages which have been defined formally [Wulf, London, and Shaw, 1976]. Typically, these programming languages support abstraction mechanisms which make program construction, by stepwise refinement, an easier task. Others have shown how proofs about programs written in von Neumann style programming languages may be constructed [Hoare, 1973; Gries, 1981].

A second group proposes that the solution to the software crisis will come, not through adaptations of the von Neumann model of computation, but rather through a novel model of computation. Typically, the novel model of computation will exhibit the mathematical property of referential transparency. This is the property whereby an expression refers to, or "denotes", a value, and the same expression always denotes the same value within the same scope [Turner, 1982]. This property allows proofs about programs to be written directly.

This second group of researchers is often at a disadvantage as, to establish their case firmly, it is important that the implementations of the programming languages based on these novel models of computation are readily available for use on existing architectures of computing systems, of which the von Neumann architecture of computing system is by far the most dominant. It is often the case that the novel models of computation cannot be implemented efficiently, with regard to time and space, on the von Neumann architecture. As a result, these novel models of computation have not received widespread acceptance because of the extra costs. In view of this, development of a novel model of computation has often led to research into novel architectures of computing systems which will efficiently support the particular model.

On a different front, developments in recent years in two separate technologies have caused a third group of researchers to investigate novel architectures. By the use of communications technology it is possible for several computing systems to be connected together by a network. The computing systems may be widely distributed but connected by a wide area network, or may be locally distributed and connected by a local area network. Communication between the different computing systems takes place across the network. Moreover, the technology of fabricating silicon chips has been advanced to the point where it is now possible to design and manufacture chips containing several processing units. Communication may occur between the different processing units on a single chip, so

that such chips exhibit the characteristics of very local distribution.

These technological developments permit computing systems to be connected together to form structured computing systems composed of subordinate computing systems. It seems probable that the standard computing systems of the not too distant future will be composed of several subordinate computing systems connected together. There are two interesting aspects to these computing systems. Firstly, the instructions of a single program may be stored within several different subordinate computing systems. Thus the execution of the instructions of the program may be performed concurrently by the processing units of the different computing systems. Secondly, the information stored within the memory component of one subordinate computing system can be made accessible to other subordinate computing systems. The execution of an instruction on one subordinate computing system may, therefore, change the information stored within a different subordinate computing system. Information will be able to flow freely between the subordinate computing systems.

To exploit this capability to the full the transfer of information between subordinate computing systems must be reflected in the models of computation on which the programming languages used to construct programs for such distributed computing systems are based. Since the transfer of information may occur concurrently, the issues which are

associated with concurrency must be considered in the design of models of computation for the exploitation of these computing systems.

An obvious first choice for the basis of the new model of computation for the distributed computing systems is the von Neumann model of computation. The benefits brought about by the introduction of structured programming techniques may have lessened the acute problems facing the designers of large software systems based on the von Neumann model of computation. The introduction of concurrency into the von Neumann model of computation such that it can be exploited by the programmer would increase the problems facing these designers. It has been argued that the von Neumann model of computation will be insufficient to allow for the correct description of computations for execution on concurrent computing systems [Chamberlin, 1971]. For this reason many researchers in computing science have turned away from the von Neumann model of computation and are investigating novel models of computation. Indeed, based as it is on an architecture which admits neither concurrency of execution nor communication of information with other computing systems, the von Neumann model of computation does not appear to be a suitable basis for a model of computation for distributed computing systems.

Some of the work of those researchers who are investigating architectures of computing system and models of computation other

than those based on the von Neumann style is surveyed in chapters two and three. Two novel designs of architectures of computing system and models of computation are described in chapter four. It is the aim of both designs to be suitable for the construction of general purpose distributed computing systems. Such computing systems might be constructed from contemporary von Neumann style computing systems, or from silicon chips each comprising several computing systems connected together on a single board.

The first of these designs, recursive control flow, originates in research undertaken at the University of Newcastle upon Tyne which investigated the use of the technology of chip fabrication to produce highly parallel computing systems [Treleaven and Hopkins, 1981]. The formal semantics for the model of computation of this design are presented. Analysis of the semantics lead to the conclusion that the recursive control flow model of computation is not necessarily the most suitable model of computation for general purpose distributed computing systems.

The second design, decentralised control, is a development of recursive control flow. It is a new design which is claimed to be suitable for the construction of general purpose distributed computing systems. The model of computation for this design is desribed and the formal semantics presented.

Since the architecture which underlies each of the proposed models of computation may well permit concurrent execution of programs, some of the issues concerned with concurrency are considered in chapter five. In particular, the notion of the consistency of the information represented within a distributed computing system is discussed. It is shown how inconsistencies may arise with the concurrent execution of instructions. Much of the material in this chapter is a survey of research in distributed data base management systems. A new strategy is outlined which ensures that the consistency of the information in a distributed computing system is maintained despite the underlying concurrency.

In chapter six, proposals for two programming languages are presented, one for each of the novel models of computation. The aim of each programming language is to describe the state of a complete computing system and not to perpetuate the traditional distinction between the programming language and the control program which controls the behaviour of a computing system. Both models of computation attempt to unify the different concepts of storage found on computing systems, since a distinction is often made between the information which is represented within a program and that which is repesented on the storage media attached to the computing system. The distinction is between information which lasts for the lifetime of a program, and that which has a lifetime exceeding that of the program. The programming language Basix originated in the work reported in [Gouveia Lima, et al., 1983]. The formal semantics for

the programming language have been constructed and are presented. An implementation of this programming language has been constructed, and several demonstration programs executed. This has allowed some experience of the programming language to be gained. Proposals for the design of a second programming language are also given, along with the formal semantics.

2 ARCHITECTURES OF COMPUTING SYSTEM

The von Neumann architecture of computing system mirrors, to a great extent, the design proposed by Babbage in the nineteenth century for his "analytic engine". This mechanical device had, amongst other things, a "mill" in which the processing of numeric quantities was performed and a "memory" where the results of the processing could be stored for use in later processing. The whole device was controlled by a sequence of pre-punched cards which specified the operations to be performed by the mill. A card would first be "read" into the device. The specified operation would then be performed by the mill retrieving quantities from the memory and returning results to be stored. Then the next card in the sequence would be read and the whole process repeated until the complete sequence of cards had been read. At this point the program specified by the sequence of operations would have been completed and the result of the program could be read from the memory. Iteration and conditional branching were provided by a sequencing mechanism which was also controlled by the pre-punched cards.

The architecture of the first stored program computing systems, such as the EDSAC, was based upon a processing unit connected to a memory component. The whole system behaved in a manner similar to Babbage's analytic engine. A sequence of operations was specified on a pre-punched paper-tape which was read into the memory component by a built-in assembler and loader. Instructions were fetched

sequentially from the memory component and executed by the processing unit. Results were stored within the memory component. Conditional branching instructions were supported which made it possible to construct program loops.

These first generation computing systems were both expensive to build and difficult to use. The capacity of the memory component was, by today's standards, extremely limited. The design of the processing unit was itself quite simple but the technology available made construction quite difficult. It was not considered feasible to have more than one processing unit in a computing system.

Since those pioneering days, with the advent of the transistor and then the integrated circuit, computing systems have become cheaper to build. Through the use of readily available mini- and micro-computing systems, computer technology has been put to uses which were undreamt of in the 1950's and the 1960's. However, the underlying architecture of the majority of these computing systems reflects the design principles of the late 1940's and early 1950's. These include, amongst other things, sequential execution of the instructions and a sharp distinction between the processing unit and the memory component.

Advances in the technology from which computing systems are constructed have allowed changes to be introduced. Some changes are simply improvements to the existing von Neumann architecture. For

example, the cost of memory devices has decreased whilst, at the same time, their capacity has increased. The capacity of the memory component on contemporary computing systems is vastly increased by comparison with that of systems of the 1970's. Similarly, more sophisticated circuitry within the processing unit permits a wider range of more complex instructions to be supported.

Other changes which exploitation of the advances in the technology permits is the removal of the distinction between the active processing unit and the passive memory component which is found in the von Neumann architecture. A possible new architecture of computing system could be based on a network of homogenous components. Each component is capable both of executing a program and storing information. Information could be transferred between the individual components.

Again, other changes, whilst maintaining the distinction between the function of the processing unit and that of the memory component could be used to produce a new architecture of computing system. Just as the memory component of a von Neumann computing system may be made up of several memory devices, so the processing unit may be replaced by several processing units. The new architecture is based around a memory component connected to a processing component which consists of many processing units.

In the following sections, the tentative outlines for the architectures of computing systems are described in more detail.

2.1 VON NEUMANN COMPUTING SYSTEMS

A von Neumann computing system consists of a single processing unit connected to a memory component. The program to be executed by the processing unit is stored within the memory component as a sequence of instructions. Execution of the program proceeds as each instruction in turn is fetched from the memory component and executed. The operands for the instructions will be stored within the memory component or in the internal registers of the processing unit. Information is passed between the instructions via the memory component or the internal registers. The order in which the instructions are executed is, in general, the order in which they are stored within the memory component. However, a mechanism is provided which allows execution to proceed to a different part of the sequence of instructions.

Speeding up the overall rate at which a sequence of instructions is executed can be achieved through the use of pipelining. As each instruction is being executed by the processing unit, the next instruction in the sequence can be retrieved and some preliminary decoding of that instruction and its operands performed. Once the first instruction has been executed, the second instruction, which has been partially decoded, can be executed. Whilst this second

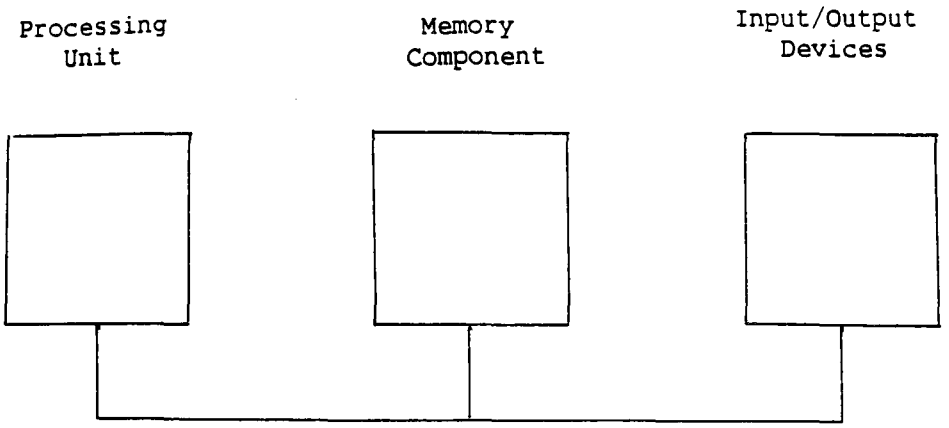

Figure 2.1 A Typical von Neumann Computing System

instruction is itself being executed, the third instruction in the sequence can be retrieved and some preliminary decoding on this instruction performed. The instructions of the program are still executed sequentially. However, the time taken to execute one instruction is overlapped with the time taken to perform some of the decoding of the next instruction. Such pipelining schemes have been used to advantage to build fast processing units for powerful computing systems.

The overall strategy of overlapping the execution of one instruction with the preliminary decoding of the next instruction can be extended. It would be possible to overlap the execution of one instruction with the preliminary decoding of several of the next instructions in the sequence. However, there are some drawbacks to the use of pipelining. First and foremost, the scheme cannot be used when conditional transfers of control are made within the program. The outcome of a conditional transfer of control instruction depends

upon a Boolean value calculated earlier in the instruction sequence. Since pipelining overlaps the execution of one instruction with the preliminary decoding of others, a transfer of control out of the current sequence of instructions will result in the decoding being performed unnecessarily for the instructions sequentially following the conditional transfer of control instruction. Furthermore, the first few instructions in the sequence to which control has been transferred will not have been decoded in advance.

Another way by which the throughput of a processing unit may be increased is by designing the processing unit to support a wider range of more complex instructions. This improves the performance of the processing unit by reducing the number of instructions in the program, and hence the number of instructions which must fetched and executed. Additionally, the designer of the processing unit may be able to take advantage of the internal organisation of the processing unit in execution these more complex instructions. However, many processing units provide instructions or addressing modes which the majority of high level programming languages cannot sensibly use. For example, in one study, measurements of the object code generated by one compiler for the IBM 360 range of computing systems have shown that just 10 instructions out of a possible 139 non-privileged instructions accounted for 80% of all instructions executed, 16 for 90%, 21 for 95%, and 30 for 99% [Alexander, 1975].

Support of more complex instructions often requires extra control paths within the processing unit. The existence of these control paths may actually slow down the execution of the more commonly used instructions. In some instances, the use of the extra complexity introduced into a processing unit may actually be less efficient than the use of the simpler instruction. For example, the operation implemented by the instruction "INDEX" on the VAX range of computing systems has been found to be less efficient than the same operation implemented explicitly using the simpler instructions of the processing unit such as "ADD", "MULTIPLY", "COMPARE", and "JUMP LESS UNSIGNED" [Patterson and Ditzel, 1980].

Furthermore, it is not always easy for the compiler writer to make use of these complex facilities. For example, the index addressing mode supported by the processing units of the VAX-11 range of computing systems may be used as the basis from which complex addressing modes to be constructed. Typically, the index addressing mode is used to specify a register which contains the index into some data structure, whilst the second addressing mode could be a displacement, relative to the program counter, which specifies the base address of that data structure. The address of the element which forms the operand to some instruction is the sum of the base address and the product of the index and the number of bytes in each element of the data structure. Thus this combination of addressing modes allows an array access in a high level programming language to be specified as a single operand to an instruction. However, use of

the index addressing mode with other addressing modes is not necessarily of general use to the compiler writer. For example, the index addressing mode together with the autoincrement indirect addressing mode can be used to specify an operand referenced through two levels of indirection, with the additional side effect that one level of indirection is changed as a result of execution of the instruction containing this compound addressing mode. It is not immediately apparent how a compiler writer can make use of this combination of addressing modes.

Such processing units may be more complex than is necessary. This complexity can be reduced in two ways. Firstly, the hardware structures required of the processing unit in order to support the chosen high level programming languages can be identified. Since many processing units are simply developments of those for which the programming language Fortran was first designed, much of the existing design is sufficient. The changes made to the processing unit may include instructions such as those to calculate the address of an array element from a dope vector. For example, the instruction "INDEX" found on DEC's VAX-11 range of computing systems may be used to calculate the address of an array element whilst at the same time checking that the index lies within the bounds specified for the array.

However, new structures, such as stacks, could be useful to support the programming language Algol 60 more effectively. The

Burroughs B5000 range of computing systems is an example of a design aimed at supporting the programming language Algol 60 efficiently. The instructions supported by the processing unit reflected the need to evaluate arithmetic expressions. All expressions were evaluated on a stack using simple "syllabic" instructions. Such instructions could cause one of the following operations to be performed:

- the value of an operand to be loaded onto the top of the stack;
- the address of an operand to be loaded onto the top of the stack;
- the top two elements of the stack to be removed, an arithmetic operation performed using them, and the result of that operation loaded onto the top of the stack.

These enhancements to the design of the processing unit of von Neumann computing systems, whilst aiming to improve the implementation of high level programming languages based on the von Neumann model of computation, amy actually not be beneficial. The realisation of the design in the available technology may be difficult because of the complex control and data paths required. An alternative approach has been to simplify the design of the processing unit. Computing systems built using such processing units are known as "RISC" (Reduced Instruction Set Computers); in general a smaller number of simpler instructions and fewer addressing modes are supported directly by the processing unit [Patterson and Sequin, 1981; Hennessey, Jouppi, Baskett, and Gill, 1983]. The facilities which are provided are aimed at being of more general use to the compiler writer than the assembly language programmer. This

simplification of the processing unit also has important side effects: the design of the processing unit is typically easier to test than that for a more complex processing unit, it requires fewer logic gates for its implementation, and should have fewer design errors.

One detailed simulation of such a simplified processing unit has shown conclusively that it outperforms a more complex processing unit, such as that found in the VAX-11 range of computing systems, in terms of speed of execution [Patterson and Sequin, 1981]. It is not obvious that simply using the advances in technology to produce more complex processing units for von Neumann computing systems is necessarily the best way to exploit the advances.

Execution of the instructions is always performed sequentially. Execution of two or more programs by a single processing unit can be achieved by either of the following two strategies.

The first strategy is straightforward. All the instructions of one program are executed before execution of the instructions of a subsequent program is started. The sequential behaviour of the processing unit is immediately apparent.

The second strategy involves the use of a control program to control the execution of the different programs. The control program arranges for groups of instructions from each program to be executed

in turn for a given time period or until some external event occurs. Some information about the state of the computing system must be saved after the execution of the group of instructions of one program. This information must be used to restore the state of the computing system when the next group of instructions from the same program is executed.

Depending upon the amount of information about the state of the computing system which has to be saved and restored, the work done over and above that of executing the different programs could be considerable. For example, consider the overheads involved in saving and restoring the state associated with a processing unit of the M68000 family. Each processing unit has eight general purpose data registers, eight address registers, and a status register which contains information about the state of the processing unit as each instruction is executed. At the very least, it will be necessary to save and restore the information represented by the status register. Additionally, if information is transmitted between the different groups of instructions of the same program through any of the registers of the processing unit, the state of those registers must also be saved and restored for each group of instructions. Failure to save and restore this information would result in the execution of one program interfering with the execution of some other program.

This second strategy gives the illusion that the processing unit can execute the programs concurrently. It is the basis of

multi-access time-sharing control programs. which allow several users to use a single processing unit 'simultaneously'. Programming languages which support concurrency have been developed both to make the construction of these time-sharing control programs easier and also to enable the programmer to exploit the concurrency provided by these control programs.

2.2 PARALLEL COMPUTING SYSTEMS

A parallel von Neumann computing system comprises what is logically a single memory component and two or more processing units. Each program is stored as a sequence of instructions and each processing unit executes the instructions sequentially. Since the memory component is directly accessible to all the processing units, the problem of interference may arise. Clearly the same memory location cannot be accessed simultaneously by two or more processing units if the respective accesses conflict. For example, one processing unit cannot store information to the same memory location from which another processing unit is fetching information. This interference, at the memory location level, can be resolved by the memory component itself.

If the programs executed by the processing units are completely independent then there can be no interference between the programs since no program has any memory location in common with any other program. However, if the programs have been written as co-operating

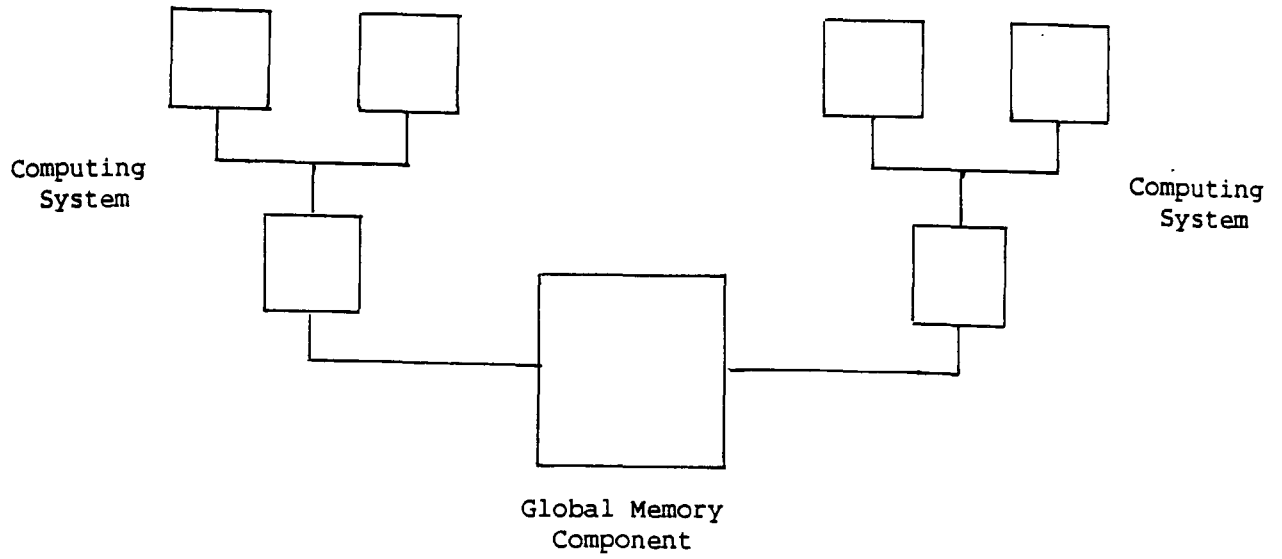

Figure 2.2 A Parallel von Neumann Computing System

tasks, the information to be shared between the programs must be stored within the memory component. All accesses to a shared memory location issued during the concurrent execution of programs will be serialised by the memory component. This may lead to the loss of information. For example, consider the following two sequences of instructions, both of which have the memory location "BAL" in common:

```
MOVE.W  BAL,D0        MOVE.W  BAL,D0
ADD.W   #100,D0       ADD.W   #100,D0
MOVE.W  D0,BAL        MOVE.W  D0,BAL
```

Execution of each sequence of instructions should increment the value held within the shared memory location "BAL" by 100. Thus, a correct ordering of the accesses made during the concurrent execution of the sequences of instructions will result in the value held within the shared memory location "BAL" being incremented by 200. However, there are some orderings of the accesses which will result in the value being incremented by 100; these orderings are incorrect. Some additional mechanism is required to force the correct ordering to be taken.

One such mechanism is based on the concept of a semaphore as found on conventional railway systems [Dijkstra, 1968]. Associated with each group of memory locations which are shared between different programs is a semaphore. The semaphore permits one program to gain control of the group of memory locations whilst the remaining programs are excluded. A simple semaphore may be implemented as a memory location on which the following operations may be performed:

```
PROCEDURE P(VAR S : Semaphore);
  BEGIN
  REPEAT
  UNTIL S = FALSE
  S := TRUE
  END { P };

PROCEDURE V(VAR S : Semaphore);
  BEGIN
  S := FALSE
  END { V };
```

When the memory location representing the semaphore contains the value 'TRUE', this indicates that the information shared between the programs is being accessed by one of the programs. No other program may access the information whilst the semaphore is set. When the memory location contains the value 'FALSE', this indicates that any program which is waiting to access the information may now do so. The semaphore must then be set to indicate that the information is currently being accessed. The responsibility for maintaining the semaphore and obeying the rules outlined above is the task of the individual program.

For the correct behaviour, the operation 'P' must be atomic. That is, it must not be possible for the operation 'P' to be started for

one program, and for this invocation to be interrupted during which time the operation 'P' is invoked for another program using the same semaphore. To avoid such interference each processing unit must support an atomic instruction to implement these semaphores. For example, the M68000 family of processing units has the 'TAS' (test and set) instruction for this purpose.

2.3 DISTRIBUTED COMPUTING SYSTEMS

The architecture of a distributed computing system consists of several subordinate computing systems connected together by some communications medium. Each subordinate computing system is autonomous and contains the facilities necessary both for processing information and for storing information.

A distributed computing system may be loosely coupled or tightly coupled. A loosely coupled distributed computing system is one in which the connection topology of the subordinate computing system is dynamic. Subordinate computing systems may be introduced into, and removed from, an existing distributed computing system. By contrast, a tightly coupled distributed computing system is one in which the connection topology is considered to be static, and it is not possible to introduce subordinate computing systems into, or remove subordinate computing systems from, an existing distributed computing system.

Some architectures of distributed computing system are based on a particular connection topology. The topology is chosen to allow any subordinate computing system to communicate with another through the minimum number of other subordinate computing systems. At one extreme this involves a ring configuration in which each subordinate computing system is connected to two other subordinate computing systems and has a worst case path of N - 1, where N is the number of subordinate computing systems. At the other extreme a cross-bar configuration occurs in which each subordinate computing system is directly connected to every other subordinate computing system and has a worst case path of 1.

Variations on this theme abound. For example the r-n-cube networks are conceptually arranged as cylinders with n rows of subordinate computing systems, each row containing rn subordinate computing systems [Burton and Sleep, 1981]. For any pair of positive integers n and r, an r-n-cube contains nrn subordinate computing systems. Each computing system is connected to 2r other computing systems. The worst path from one computing system to any other computing system is 3n/2. For example, a 4-8-cube has 524,288 subordinate computing systems, each of which is connected to 8 other subordinate computing systems. The worst path from one subordinate computing system to any other subordinate computing system is 12.

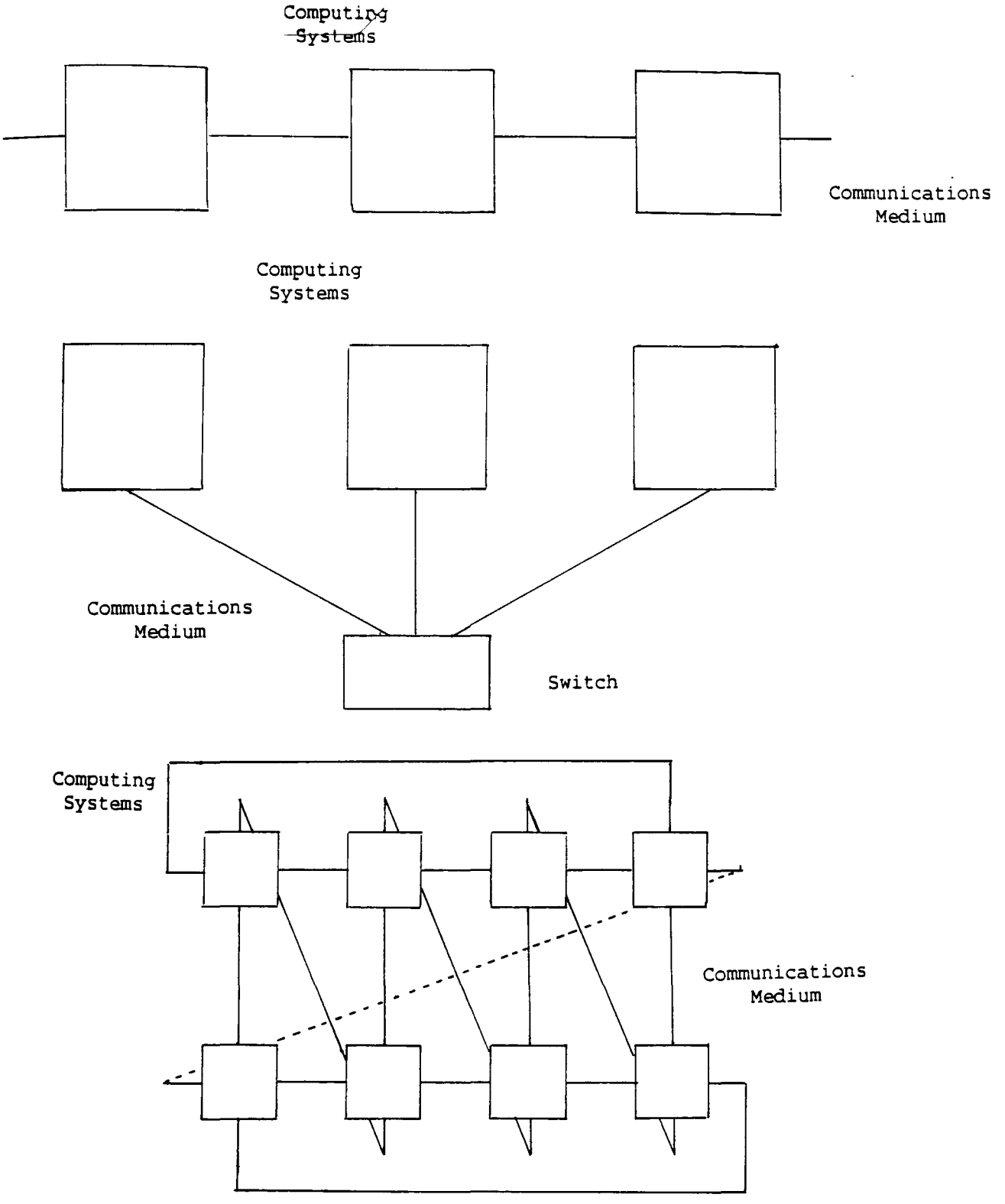

Figure 2.3 Various Distributed Computing Systems

2.4 DATA FLOW COMPUTING SYSTEMS

Data flow computing systems are an example of an architecture of computing system which has been designed as the result of research into novel models of computation [Chamberlin, 1971]. This particular architecture represents an attempt to provide efficient support for implementations of programming languages such as Val [Ackerman, 1978] and Id [Arvind, Gostelow, and Plouffe, 1978]. These programming languages are referentially transparent and support the "declarative" style of programming.

As outlined earlier, a program for a von Neumann computing system is represented as a sequence of instructions. Execution of the program proceeds as each instruction in the sequence is executed. For example the expression $ax^2 + bx + c$ could be evaluated by the following instructions for a von Neumann processing unit:

```
MOVE.W  x, D0
MOVE.W  x, D1
MULS    D1,D0
MOVE.W  a, D1
MULS    D1,D0
MOVE.W  x, D1
MOVE.W  b, D2
MULS    D2,D1
ADD.W   D1,D0
ADD.W   c, D0
```

The order in which these instructions are executed by the processing unit will be that in which they are written. Information is transmitted between the instructions by reference to globally accessible memory locations; in the example above the general data registers of the processing unit have been used for that purpose.

In complete contrast, the instructions of a program written for a data flow computing system are executed when the operands of the instruction are available. The same expression could be evaluated by the following instructions for a data flow computing system:

```
1.  MUL  x, x, 2/1
2.  MUL  ?, a, 4/1
3.  MUL  b, x, 4/2
4.  ADD  ?, ?, 5/1
5.  ADD  ?, c. -/
```

The values of the operands a, b, c, and x will be 'placed' into the instructions as they become available. Once all the operands for a particular instruction are available that instruction may itself be executed. As a result of the execution of the instruction further information may become available for use as operands to other instructions. In the example given above, the notation '2/1' in an operand position in the first instruction indicates that the information gained from execution of the that instruction should be placed in the first operand position of the second instruction. The question mark in the first operand position of the second instruction indicates that the operand will become available as a result of the execution of some other instruction. The instructions of a program may be executed concurrently. In the example above, it would be possible to perform the execution of the first and second instructions concurrently with that of the third instruction. Since there are no side-effects permitted in the data flow model of computation, this concurrency needs no synchronisation of the memory component.

The information which becomes available as a result of the execution of an instruction is a value which may be copied into the operands of other instructions. Whenever an operation is performed on an operand which is a composite value, for example an array or a record, the original value is left unchanged and a copy representing the alteration is created. This copy is then propagated to other instructions. This is a drawback to the data flow architecture of computing system. Programs for manipulating large data structures result in several copies of that data structure being represented within the memory component.

The architecture of a data flow computing system is typically based on a simple ring structure, with four different functional components connected together on this ring:

- one or more memory components;
- one or more processing units;
- a routing network;
- some input/output devices.

The memory component is used to store the instructions of the program. Since all information is represented within the operands of instructions, it is not possible to update a particular element of a data structure and allow that change to be visible to all other instructions. Instructions are fetched from the memory component when all of its operands are available. There may be several processing units in the ring each capable of executing a single instruction at a time. This allows the execution of a program to

proceed concurrently. Since there is no globally updateable memory component, there is no need for synchronisation between the different processing units. The results obtained by the execution of the instructions are passed to a routing network. The routing network copies the results of the executed instructions into the specified operands of other instructions. This may then allow these latter instructions to be fetched from the memory component and executed by the processing units. Finally there is an interface between the ring and the input/output devices. This allows the data flow computing system to be attached to conventional devices or to be a subordinate part of some larger computing system.

2.5 REDUCTION COMPUTING SYSTEMS

Several functional or applicative programming languages have been proposed as candidates for the solution of the software crisis. In their favour is the supposed ease with which a program may be written. In particular, it has been argued that these programming languages enable the early production of prototypes of software systems for appraisal by the users [Henderson and Minkowitz, 1986; Turner, 1985]. However, implementations of these programming languages on existing von Neumann architecture computing systems have not been altogether successful. A common complaint has been that the programs written in these programming languages are simply not space or time efficient when they are executed. Programs written in a functional programming language often take longer to execute and

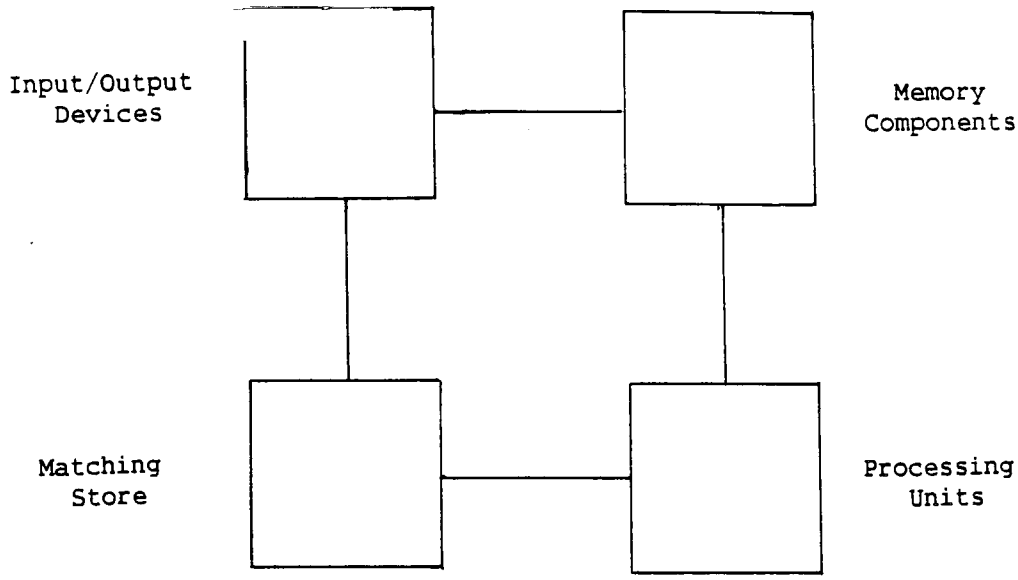

Figure 2.4 A Data Flow Computing System

require more space than the equivalent programs written in a von Neumann style programming language. Despite these serious drawbacks, functional programming languages are still considered to be useful. This has directed research into the design of computing systems which will support an implementation of a functional programming language as efficiently as an implementation of a von Neumann style programming language can be supported on a von Neumann computing system.

The program to calculate the value of the expression "$ax^2 + bx + c$" can be represented as a graph. Each terminal node represents a basic operand and each non-terminal node represents an operator. The program is executed by evaluating the nodes and collapsing the graph to a single value which denotes the value of the expression. This process is known as reduction. The instructions shown below are a linear representation of the graph:

1. ADD 2 c
2. ADD 3 5

3. MUL a 4
4. MUL x x
5. MUL b x

The integers in the operand position of an instruction are a reference to the sub-graph of instructions which must be reduced in order to provide the actual value of the operand. For example, the instruction "MUL a 4" indicates that whilst one of the operands for the multiplication operation is to be taken from the memory location associated with the label "a", the other will be obtained by evaluation of the instruction with the label "4". Once an instruction has been executed it can be reduced to the value obtained by the execution of that instruction. Any further reference to that instruction yields the value and does not cause the instruction to be executed again.

An architecture for a reduction computing system can be based on a ring structure similar to that of a data flow computing system. The instructions which make up the program graph are stored in the memory component. An instruction can be executed when all of its operands are available. An operand is available if it is a simple value such as an integer or a reference to a data structure. An operand which is a reference to some other instruction is, however, unavailable. Since the operands are accessed by reference, copies of a large data structure are not propagated between instructions.

Early designs for reduction machines consisted of a single processing unit connected to a single memory component [Berkling,

1975; Clarke, Gladstone, MacLean, and Norman, 1980]. The former is a design for a string reduction computing system, the latter is a design for a graph reduction computing system.

A more recent design, known as "ALICE" (Applicative Language Idealised Computing Engine), has been built around a distributed computing system [Darlington and Reeve, 1981]. This computing system is comprised of a group of memory components and a group of processing units connected together by a switching network. The latter allows any processing unit to access any memory component. A processing unit fetches from a memory component an instruction which can be executed. This instruction may require other instructions to be executed before it can itself be executed. The performance of this exploratory implementation of a reduction computing system is far from ideal. Recent figures state that reductions can be performed at the rate of one thousand per second [Townsend, 1987]. When it is considered that the implementation of this reduction computing system utilises over one hundred Transputers, it can be appreciated that the overheads of the reduction architecture are not insignificant.

The ALICE project has been developed further as the "Flagship" project [Watson, Sargeant, Watson, and Woods, 1987]. Once again, the basis of the project is a distributed computing system, but each memory component is associated with a particular processing unit. The memory component - processing unit pairs are connected together

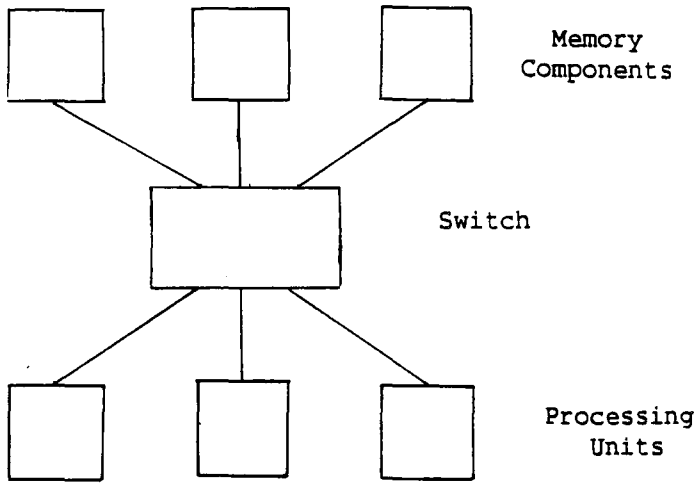

Figure 2.5 A Reduction Computing System

by a switching network which allows any processing unit to access any memory component. However, restraining the processing unit to access the local memory component is preferable since it reduces the amount of information transmitted across the switching network. This thereby increases the overall performance of the computing system. This restraint can be imposed by ensuring that the information required by a processing unit is to be found in the local memory component with which it is associated.

2.6 ARRAY PROCESSORS

An array processor is a specialised computing system suitable for executing programs which require the manipulation of arrays. Tasks such as weather forecasting and image processing depend heavily upon array processing. Many existing von Neumann computing systems cannot provide the raw computing power to support these tasks adequately. The problem lies in that execution of an instruction by a von Neumann

processing unit can only affect, at most, a few operands. Array addition must be expressed as a sequence of instructions which are executed in turn. For example, the following program performs addition of two arrays with 64 elements:

```
FOR i := 1 TO 64
  DO C[i] := A[i] + B[i]
```

The group of instructions to perform the addition of one element must be executed 64 times. Furthermore, there is an overhead involved in maintaining the loop control variable.

In contrast, the example of array addition could be performed by the single ADD instruction on an array processor. The distinction between the von Neumann processing unit and the array processor lies in the ability of the latter to execute a single instruction which affects several groups of operands. There is a single centralised control unit which is responsible for controlling the parallelism.

An array processor is composed of a group of processing elements each with its own local memory component. The individual processing elements can execute instructions which manipulate the information stored within the local memory component. Also, the processing elements are all connected to a single control unit which forms an interface between the host computing system and the processing elements. The control unit propagates individual instructions to the processing elements and also routes information between the host computing system and the processing elements. The processing

elements are themselves connected together by a network which allows information to flow between them. For example, this network may connect the processing elements into a square array in which each processing element is connected to four neighbours. The execution of a sequence of instructions by each processing unit is synchronised by the control unit. The interconnection of the processing elements allows information to be moved directly between adjacent processing elements. For example, calculations such as the following may be represented by a single instruction:

```
FOR i := 1 TO 63
  DO A[i] := A[i + 1]
```

A single "mask" bit in each processing element indicates whether that particular processing element is disabled or enabled. Instructions are only executed by those processing elements which are enabled. Programs such as the following can be represented as a single instruction by setting the "mask" bit in each processing element as appropriate:

```
FOR i := 1 TO 64 DO
  IF C[i]
    THEN A[i] := B[i]
```

Array processors can be difficult to program. The vectors and matrices of a program must be sub-divided into groups of data which can be mapped onto the individual processing elements, and allocated to those processing elements so as to take advantage of the particular interconnection topology of the elements. If there are

too few processing elements in the array processor then parts of the vector or matrix must be exchanged between the processing elements and the host computing system. It may prove difficult to minimise the flow of information between the processing elements.

2.7 CONCLUSION

It has been assumed for the purpose of this thesis that the general purpose computing systems of the future will consist of several subordinate computing systems interconnected by some communications medium. Information may flow freely through the medium between the subordinate computing systems. It will be desirable to be able to introduce subordinate computing systems into an existing distributed computing system and to remove subordinate computing systems from an existing distributed computing system. To be useful, these operations should cause the minimum disruption to the existing distributed computing system.

Some design principles are required to enable distributed computing systems with the characteristics outlined above to be constructed. Several architectures of computing system have been surveyed in this chapter, and the general design principles of each outlined. No one architecture provides a satisfactory basis on which the distributed computing systems of the future may be constructed.

The separation of the processing units and the memory components into distinct groups, as is found in the parallel von Neumann as well as in some data flow and reduction computing systems, makes it difficult to introduce or remove subordinate computing systems. This arises because the memory components are globally accessible to all the processing units. One refinement of the reduction architecture of computing system, in which each memory component is paired with a processing unit, does not relieve this particular problem since, as before, the memory components are still globally accessible to all of the processing units.

The use of special topologies, as is found in array processors and closely coupled distributed computing systems, also makes it difficult to construct general purpose distributed computing systems with the characteristics outlined above. Typically, the interconnection topology of the architecture is static and relies on the existence of neighbours. The programmer is encouraged to take advantage of this dependence when writing programs for these architectures.

Since none of the architectures surveyed in this chapter fit the requirements of the future, two new designs of architecture are proposed. These are outlined in chapter four.

3 MODELS OF COMPUTATION

The growth of interest in models of computation, as discussed earlier, has been prompted by the software crisis of the 1970's and the advances made in the technology of chip fabrication. The former has led to the development of models of computation with rigorous mathematical definitions. The latter has resulted in the design of novel architectures of computing system which will support different models of computation. Many of these models of computation have also been defined rigorously.

A rigorous definition of a programming language allows the construction of proofs about the behaviour of programs written in that language. These proofs, along with the original specification from which the program was written, make it possible to demonstrate that the program fulfills the objectives set [Ambler, Good, Browne, Burger, Cohen, Hoch, and Wells, 1977]. Furthermore, a rigorously defined programming language can ease the refinement process involved in the development of a program from an abstract specification. At each step in the refinement process it is possible to show that satisfactory progress has been made.

Computing systems may be compared with simple electronic calculators. To perform a calculation on a calculator the calculation must be broken down into a sequence of steps. Each step is a basic operation which can be performed on the calculator.

Knowledge of the nature of the calculation and the calculator are needed before the calculation can be performed. Furthermore, some method by which the calculation can be broken down into the sequence of steps must be known. For example, the value of the expression '$ax^2 + bx + c$' could be calculated by the following steps:

```
CLEAR
ENTER    a
ENTER    x
MULTIPLY
ENTER    b
ADD
ENTER    x
MULTIPLY
ENTER    c
ADD
DISPLAY
```

Should it be necessary to perform the calculation on a different type of calculator, the sequence of steps would probably have to be rewritten to take account of the operations which could be performed on the second calculator. For example, the same calculation could be rewritten as the following steps:

```
CLEAR
ENTER    a
MULTIPLY
ENTER    x
ADD
ENTER    b
MULTIPLY
ENTER    x
ADD
ENTER    c
DISPLAY
```

All that was involved in the transition from the first example to the second was translation of the calculation from postfix notation to infix notation. The nature of calculators is such that the

operations which can be performed are based on arithmetic operations. Consequently the operations are generally well understood and transferring a calculation from one calculator to a different calculator is not that complex.

The specification of the behaviour of the calculator can be expressed in terms of the net effect each operation has when it is performed; this is the model of computation. The model of computation specifies what can be done using the calculator. This specification may be given in natural language or more formally. For example, a formal specification of the previous example is given below using VDM [Jones, 1986]:

```
CLEAR
ext wr Mem1, Mem2 : [Real]
post Mem1 = nil and Mem2 = nil

ENTER (R : Real)
ext wr Mem1, Mem2 : [Real]
pre Mem2 = nil
post Mem1 = R and Mem2 = Mem1'

ADD
ext wr Mem1, Mem2 : [Real]
pre Mem1 ≠ nil and Mem2 ≠ nil
post Mem1 = Mem1' + Mem2' and Mem2 = nil

MULTIPLY
ext wr Mem1, Mem2 : [Real]
pre Mem1 ≠ nil and Mem2 ≠ nil
post Mem1 = Mem1' * Mem2' and Mem2 = nil

DISPLAY R : REAL
ext wr Mem1, Mem2 : [Real]
pre Mem1 ≠ nil
post R = Mem1' and Mem1 = Mem2' and Mem2 = nil
```

Using this specification it can be shown that the sequence of instructions for the first calculator will yield the value of the expression given. Furthermore this specification outlines the limitations of this particular calculator. It can be seen that there are only two memory elements associated with the calculator and that these elements are used as a simple push-down stack. Consequently, only two operands may be represented within the calculator at any one time.

The programming language for the calculator is based upon the model of computation for that calculator. It defines the "sentences" which may be written and provides a way of understanding the meaning of those sentences. Again this definition can be given either formally or informally. An example of a formal definition is given below:

```
Instruction ::= CLEAR
              |   ENTER real
              |   ADD
              |   MULTIPLY
              |   DISPLAY

Program     ::= Instruction
              |   Instruction Program

Meaning["CLEAR"] = CLEAR
Meaning["ENTER real"] = ENTER (real)
Meaning["ADD"] = ADD
Meaning["MULTIPLY"] = MULTIPLY
Meaning[Program] = Meaning[Instruction]; Meaning[Program]
```

The definition consists of two parts. The first part defines the sentences of the language which are syntactically valid. The second

part defines the subset of the sentences which are semantically valid. For example, the sentence "CLEAR ADD" is syntactically valid, yet it is not semantically valid because the pre-condition of the operation "ADD" does not hold.

The processing unit of a computing system executes instructions in much the same way as operations are performed on a calculator. In a similar way the model of computation specifies what can be done using the processing unit. However, the variety of instructions which can be executed by a processing unit are far greater than the set of operations which can be performed by a calculator. Furthermore, the differences between the instruction set of one manufacturer's processing unit and that of some other manufacturer can be vast. A model of computation could be based upon the instruction set of a particular processing unit but there are serious drawbacks to this approach. Firstly, understanding the complete model of computation would be beyond the capability of many users. Secondly, transferring programs from one processing unit to another with a different instruction set requires the program to be rewritten. Thirdly, mapping a high-level description of a problem into a sequence of instructions for the processing unit is tedious and error-prone.

An example of a specification of the behaviour of a single instruction, MOVE.L, is given below. This instruction, which is supported by the M68000 family of processing units, moves a 32 bit quantity from a given source to a given destination. Various

condition flags are set depending upon the value transferred. Different combinations of the addressing modes may also be used with this instruction; only the register direct mode is specified below.

```
MOVE.L  DS, DD : { 0 .. 7 }
ext wr DR : map { 0 .. 7 } to LongWord
ext wr CC : map { X, N, Z, V, C } to Bit
post DR = DR' | { DD -> DR'(DS) } and
     CC = CC' | { V -> 0,
                  C -> 0,
                  N -> if DR'(DS) < 0 then 1 else 0,
                  Z -> if DR'(DS) = 0 then 1 else 0 }
```

Even for this trivial instruction with a relatively simple combination of addressing modes, the specification is far from transparent. The amount of state information which must be considered is large; in this example it constitutes eight data registers and the five condition bits. A specification of all 74 instructions of the M68000 processing unit with the different combinations of the ten addressing modes would have to take into account the complete state of the computing system of which that processing unit was a part. The state would include not only the internal registers of the processing unit (eight data registers, eight address registers, the program counter, the status register) but also the memory map of the computing system. For an idea of the size that the specification of a processing unit can reach, that for the IBM 370 requires 446 pages [IBM, 1987].

A model of computation based on the behaviour of machine instructions is not necessarily very useful. Programs are built from

sequences of instructions and groups of these instructions perform useful tasks. Consequently, a model of computation may describe the behaviour of these larger groups of instructions but not outline the behaviour of the individual instructions from which they are built. Indeed, the model of computation may describe the behaviour of programs in terms of a computing system which does not exist, but which has been defined in some formal mathematical sense. This approach allows the description of the computations to be raised from the level of the transfer of information between registers of the processing unit towards mathematical logic. Ultimately these high level programs must be mapped into the instructions of some existing architecture of computing system, if they are actually to be used. This implementation maps the abstract details of the model of computation into the concrete details of the architecture of the computing system on which the abstract programs are to be executed. This abstract approach is outlined in [Landin, 1964; Landin, 1965], where a model of computation based on Church's Lambda notation is presented. In [Landin, 1964] the SECD "machine" is described; this has been used as a basis for the programming language Lispkit [Henderson, 1978].

From the above it is apparent that there can be a total separation between the concrete details of a particular computing system and the abstract details of some model of computation. Consequently, a programming language need not mirror the underlying details of any particular computing system. Clearly there are advantages and

disadvantages to be considered in the choice between an operational model of computation and one that reflects more abstract mathematical concepts. The operational model of computation allows the user to write programs with the knowledge of the intermediate states through which the computing system will go in order to execute the instructions of those programs. This, it could be argued, permits the maximum performance to be extracted from the computing system. Alternatively, the mathematically orientated model of computation allows programs to be written "transparently". The ease of understanding of such programs exceeds that of currently existing programs which are expressed in operational models of computation. Reasoning about the behaviour of the programs is also facilitated.

Once this abstract approach has been adopted it is debatable if there is one model of computation which is better than any other. Historically the von Neumann model of computation has dominated Computing Science. This may be purely for pragmatic reasons; it can be implemented efficiently on the dominant architecture of computing systems. However, there are now three or four alternative models of computation challenging its position.

As outlined in chapter one, the impetus to investigate these new models of computation has come from two directions. The software crisis of the 1970's has directed research into more formal models of computation. Similarly, advances in the technology of chip fabrication have brought about an interest in new architectures of

computing system. The aim of this chapter is to review the different models of computation and to investigate how concurrency is supported within them.

3.1 THE VON NEUMANN MODEL OF COMPUTATION

The first programming languages used, those of the early 1950's, were based directly on the architectures of the available computing systems. The instructions executed by the processing units of these computing systems were reflected in the description of the programming languages used. The model of computation was, in effect, a description of the behaviour of the underlying computing system. To write a program the user needed a detailed knowledge of the architecture of the computing system on which the program was to be executed. This was a hinderance since once a program had been written for one particular computing system, it could not easily be used for other computing systems.

The move away from the architecture of the underlying computing system as the basis for the model of computation started with the development of primitive high level programming languages, known in Britain as "autocodes" [Brooker, 1958]. Rather than write a program in terms of the architecture of the particular computing system upon which the program was to be executed, generalised arithmetic expressions involving integers were used. These expressions were mechanically translated into sequences of instructions for the

particular computing system on which the program was to be executed. Each computing system had its own particular autocode but there were sufficient similarities between them to make translation from one to another relatively easy [Burnett-Hall, Dresel, and Samet, 1964].

This led to the development of the programming language Fortran which is of particular interest [Samet, 1969; Knuth and Pardo, 1976]. Machine independence was not initially a primary goal, but it was deemed important to have a notation which was mathematically concise and which did not resemble the instructions of any particular processing unit. Furthermore, it was proposed that the processing units of future computing systems should be designed to support an instruction set which would make implementation of this programming language relatively easy. The model of computation for the programming language Fortran exhibits many features of the computing system on which it was first implemented, the IBM 704; examples are:

- linear program structures;
- linear data structures;
- operations on simple data elements (integers and reals);
- sequential execution of programs.

The work of an international committee, which resulted in the definition of the programming language Algol-60 [Naur, 1963], was without reference to any particular computing system then in existence. It was, however, dependent upon the von Neumann architecture of computing system. Indeed, it has been shown [Landin,

1965a; Landin, 1965b] how some of the semantics of Algol-60 may be expressed in the Lambda notation of Church and how the SECD "machine" may be used as an abstract basis for understanding the meaning of computations written in the programming language Algol-60. The definition of the model of computation for Algol-60 gave rise to different implementations of the language on several computing systems. It was possible, in theory, to transfer programs written in Algol-60 from one computing system to some other computing system.

The trend of designing models of computation independently from any specific computing system had now begun. Often the design of a model of computation encouraged research into computing system architectures which would efficiently support that model of computation.

The basis of the von Neumann model of computation is an architecture of computing system comprising a single processing unit with sequential control of operation and a set of resources, typically memory cells and peripheral devices. A program is written as a sequence of instructions, the execution of these instructions being basically sequential, based on the flow of control from one instruction to the next in the sequence. At any time during the execution of a program there is a unique point in the sequence of instructions which identifies the instruction currently being executed. A mechanism is also provided within the von Neumann model of computation to enable non-linear programs to be represented. This

mechanism can be thought of as the explicit movement of the unique point from one instruction to some other instruction. The flow of control resembles a single thread running through the program. The thread links together the instructions in the order of their execution. Since there is a single thread of control, the instructions may not be executed concurrently.

As the flow of control passes from one instruction to the next instruction transitions are made in the state of the computing system. To understand the behaviour of an execution of a program it is necessary to construct the trace of the state transitions.

The details of the von Neumann model of computation are evident from the design of the early computing systems. The architecture of such computing systems as the IBM 704 and the more recent ICL 2900 range demonstrate facets of the model at the level of the hardware; sequential execution of machine instructions, a "program counter" to represent the unique point of control, a memory component with a linear address space, and basic operations on words and bytes. Since the von Neumann architecture of computing system is also known as the "control flow" architecture, the von Neumann model of computation is often referred to as the "control flow" model of computation.

3.2 PARALLEL CONTROL FLOW

The control programs for the first computing systems processed the individual steps of a task sequentially. Each program was executed to completion before the execution of the next program was started. As computing systems became more powerful it was both feasible and desirable to share the resources of these computing systems amongst several users simultaneously. Time-sharing control programs gave the illusion that several programs could be executed concurrently by the processing unit of a von Neumann computing system. Some control programs even allowed parts of the same program to be executed concurrently. To take advantage of this parallelism the parallel control flow model of computation was developed as a generalisation of the existing von Neumann model of computation.

In the parallel control flow model of computation the flow of control is not restricted to a single thread. Different parts of a single program may be executed concurrently. Mechanisms exist for controlling the concurrency explicitly. To enable information to be passed between the different parts of the program a globally accessible memory component is provided. Since the parts of the program may be executed concurrently the problem of interference must be addressed. Interference occurs when a data structure which is held within the globally accessible memory component is subjected to conflicting operations made on behalf of two or more programs which are executed concurrently. For example, the following two sequences

of instructions make use of the variable "X" which is stored within the globally accessible memory component:

```
a := X;          b := X;
X := f(a)        X := g(b)
```

Execution of the first instruction of each sequence of instructions will cause the value represented by the variable "X" to be fetched from the globally accessible memory component into the local data space of each sequence. Execution of the second instruction of each sequence will cause the variable "X" to have some other value assigned to it. The memory component cannot respond to concurrent "read" and "write" requests. Consequently, the requests made during the concurrent execution of the two sequences of instructions given above will be serialised. Certain orderings of these requests will result in the apparent loss of some of the requests. Various strategies have been devised to ensure that the concurrent execution of the different sequences do not interfere with one another. This problem is considered in greater detail in chapter five.

Programming languages in the parallel von Neumann style have allowed the concurrency to be expressed at different levels. At one extreme the individual statements of a program may be executed concurrently; at the other extreme individual procedures and functions may be executed concurrently. In both cases the concurrency is under the control of the programmer.

3.3 OBJECT ORIENTED

The object oriented model of computation is probably best known through the programming language Smalltalk [Ingalls, 1978]. The model of computation is a generalisation of the von Neumann model of computation. Programs are constructed from instances of a set of globally accessible objects and each object implements some specific function which is of general use to the programming community. For example, an object might implement a symbol table [London, Shaw, and Wulf, 1978]. The description of this object will not only describe the variables required to represent the symbol table, but will also describe the routines which are necessary to manipulate those variables.

An instance of an object may be created dynamically as a program is executed; this instance is distinguishable from all other instances of any object. The instance has a local data space which contains the variables used to represent the symbol table. This local data space may only be manipulated by the routines which are described in the textual description of the object.

Information may be transmitted between different instances of objects. In this way, one instance of an object may use the facilities provided by another instance of an object. Two different approaches to the transmission of information between instances of objects have been taken by designers of object oriented programming

languages.

One mechanism is based on procedure calls [Ingalls, 1978; Liskov, Moss, Schaffert, Scheifler, and Snyder, 1978]. An instance of an object makes a request of some other instance of an object by invoking one of the routines provided by the second instance. Control is passed from the instance of the object making the request to the instance of the object which will satisfy this request. The routine of the second instance is then executed, and control is finally returned to the first instance which made the original request. The instances of objects which comprise a program are not executed concurrently.

The alternative mechanism is based on messages [Hewitt and Baker, 1977]. An instance of an object sends a message bearing some request to another instance of an object. The second instance then processes that request, and may return a message bearing some reply to the first instance which made the original request. The two instances are executed concurrently. The strategy based on messages can be used to build programs where the relationship between an instance of an object making a request of some other instance of an object cannot be represented simply as that of client and server. For example, there is no requirement that the instance of the object which receives a request should respond directly to the instance which made that request. The request could be forwarded to some other instance of an object, and a response generated from this third instance.

The origins of the object oriented model of computation may be traced through the development of the programming language Simula [Dahl, et al., 1972] to the work on abstract data types [Liskov, et al., 1976].

3.4 LOGIC

Natural language as a model of computation would make the construction of programs an easier task than it is at present using von Neumann style programming languages. The interests of the user would be expressed directly in the program rather than in the details of the underlying computing system on which the program was to be executed. Even though it is not possible to take the sentences of a natural language and to translate them into the corresponding instructions for a computing system, it has been suggested [Kowalski, 1974] that the predicate calculus mirrors rational human thought. Consequently, a model of computation based on the predicate calculus would be naturally orientated towards the user. This is the approach taken by the Japanese Fifth Generation Computing Project [Uchida, 1982].

The Japanese project proposes that the computing systems of the 1990's will be much more high level than contemporary computing systems. Their long-term goal is to design and produce an architecture of computing system to support efficiently the logic model of computation.

One of the results of research by the Artificial Intelligence community has been the construction of theorem provers for statements written in the first order predicate calculus. These theorem provers are, in fact, implementations of the logical model of computation. A program written in the logic model of computation consists of a sequence of statements expressed in the predicate calculus. Some of these statements specify the facts and rules about the problem, whilst other statements represent various theorems which are to be proved with respect to the facts and rules. Execution of a program is the process of proving or refuting these statements.

A program written in the logic model of computation consists of a set of propositions and a set of queries expressed in a subset of the first order predicate calculus known as Horn clauses. The set of propositions represents the knowledge of the world modelled by the program. This knowledge can be categorised into facts and rules.

A fact describes some property of one or more individual objects in the external world. For example, the following two statements represent information about the properties "mortal" and "father", and the objects "socrates", "john", and "bill":

```
mortal(socrates) <-
father(john, bill) <-
```

The first statement may be interpreted as "Socrates is a mortal", whilst the second statement may be interpreted as "John is the father of Bill".

A rule describes some relationship between various groups of objects in the external world. For example, the following two statements represent information about the rules "likes" and "grandfather":

```
likes(john, X) <- likes(X, john)

grandfather(X, Y) <- father(X, Z), father(Z, Y)
```

The first statement may be interpreted as "John likes everyone who likes him", whilst the second statement may be interpreted as "Somebody is the grandfather of somebody else if there is someone who has the former as their father and who is himself the father of the latter".

The goal statement

```
<- father(john, bill)
```

is a query which is satisfied if the statement "father(john, bill)" has been asserted as a fact.

The goal statement

```
<- grandfather(john, X)
```

is a query which is satisfied if at least two statements of the form "father(john, Y)" and "father(Y, X)" have been asserted as facts, where the logic variable "Y" may be replaced by the same object in both statements. The logic variables "X" and "Y" will be instantiated to the set of objects for which such pairs of statements exist.

The logic variables introduced into the clauses and statements of a logic programming language differ from the variables within a von Neumann programming language. In a von Neumann programming language a variable is given a value by one instruction and this can be changed at will by other instructions. In contrast, a variable in a logic program is instantiated to a set of values which satisfies all clauses in the statement. Variables of a statement are considered local to all clauses of that statement.

Execution of a program written in a logic programming language is an attempt to prove or refute the goal statements of the program with respect to the propositions. The goal statement will be a set of clauses, each of which must be proved if the goal statement is itself to be proved. The clauses will make reference to the propositions of the program. Variables appearing within a clause must be instantiated to the set of values for the corresponding proposition.

Proving or refuting a goal statement can be a complex process. This process is known as resolution [Robinson, 1965]. Variables appearing in more than one clause of a goal statement are shared between the clauses. Once a variable has been instantiated to a value, that instantiation is visible to all other clauses using that variable. Should a particular instantiation of the variables refute the goal statement, then further instantiations may be made. This process involves backtracking through the propositions of the program. Quine [Quine, 1974] also gives a rule for the resolution of

statements in logic; he highlights how statements containing variables may be resolved. Quine's rule only allows boolean variables within the statements; to permit boolean predicates to appear within the statements the rule must be extended slightly. Rather than putting first '**True**' and then '**False**' for some chosen variable, the predicate must be replaced by '**True**' and '**False**'. Replacing a predicate by '**True**' involves determining those arguments for which the predicate will return '**True**'. Similarly, replacing a predicate by '**False**' involves determining those arguments for which the predicate will return '**False**'. It is in the process of determining these arguments that backtracking occurs.

From the rule for the resolution of statements in the first order predicate calculus, it is possible to determine how parallelism may be exploited. Firstly, all parallelism is under the control of the theorem prover and cannot be exploited by the user. Within that there are two forms of parallelism which may be exploited; "OR parallelism" and "AND parallelism". "OR parallelism" may be exploited when a proposition is given as a set of statements. Each statement can be resolved concurrently. For example, "OR parallelism" may be exploited in the resolution of the following clause:

```
likes(john, X)
```

given the proposition

```
likes(john, X) <- likes(mary, X)
likes(john, X) <- hates(fred, X)
```

Each proposition will produce a set of values for the logic variable "X" which satisfy the clause. The union of these two sets produces the overall result for the goal statement.

"AND parallelism" may be exploited when there is more than one clause in a goal statement. Each clause of the goal statement is resolved concurrently. It is necessary to synchronise the instantiation of any variables which are common to the clauses. "AND parallelism" may be exploited in the resolution of the following goal statement:

```
likes(john, X)
```

using the proposition

```
likes(john, X) <- likes(mary, X) , hates(fred, X)
```

Each clause produces a set of values. The result of the goal clause is the disjoint union of these sets.

The logic model of computation is good at representing structural relationships between objects. It has been used to construct so-called "expert systems" which represent specialist knowledge about particular subjects. However, it is poor at representing mathematical relationships. For example, the ubiquitous "Factorial" function is shown below:

```
fac(0, 1)
fac(1, 1)
fac(N, R) <- N > 1, sub(N, 1, X), fac(X, Y), mul(N, Y, R)
```

The arithmetic operators are represented as relations between numbers.

A programming language based on the logic model of computation is, strictly speaking, referentially transparent. The set of propositions which are used for resolving queries should be static. However, it is often useful to be able to add extra propositions as a program is executed, or to remove or amend existing propositions. For this reason extra-logical statements are often added to logic programming languages. The property of referential transparency is lost when such statements are introduced into the programming language.

3.5 REDUCTION

The reduction model of computation was first formulated in the late 1950's and early 1960's during the development of the programming language LISP. LISP is a language for expressing algorithmic thoughts and is a formalism for reasoning about recursion equations as a model of computation [Sussmann, 1982]. The core of the programming language LISP, commonly known as "pure LISP", is an example of the reduction model of computation. Pure LISP is a mathematical programming language which has a formal and complete description; no such claims are made for the extension of the language, LISP 1.5, which bears a significant resemblance to a programming language based on the von Neumann model of computation [McCarthy, Abrahams, Edwards, Hart, and Levin, 1962]. Within the reduction model of computation there is no concept of a globally accessible memory component which may be altered during the execution

of a program. In a reduction programming language, there are no assignment statements which could give rise to side-effects; the reduction model of computation exhibits the property of referential transparency as outlined in chapter one.

A program written in the reduction model of computation is a mathematical function. The overall structure of a complex program is a hierarchy of function applications and each constituent function within the hierarchy may be regarded as a program in its own right. Since the reduction model of computation permits no side-effects, information is transmitted between different parts of a program through the argument passing mechanism and the result returning mechanism of function applications.

The hierarchy of function applications which occur during the execution of a program form a tree. The root of the tree represents the outermost function application whilst the leaves of the tree represent the individual variables and constants of the program. Executing the program is equivalent to 'walking through' the tree. When a fragment of the tree has been evaluated, that part of the tree may be replaced by the value. This is known as reduction. It is safe to perform reduction because of the referential transparency property of the reduction model of computation.

There are two distinct routes which this walk through the tree may take - applicative order and normal order. Applicative order

reduction is equivalent to a walk through the tree from the leaves to the root. All of the expressions forming the arguments of a function application are evaluated before the function itself is applied to the arguments. If the value of an argument is not used during the application of the function, then the work performed evaluating that argument is wasted.

Normal order reduction is equivalent to a walk through the tree from the root to the leaves. The expressions forming the arguments to a function application are only evaluated when the application of the function requires their value. Clearly this could lead to arguments being evaluated more than once, which is unnecessary since the model of computation does not permit side-effects. The strategy of only evaluating at most once those arguments whose value is required is known as "lazy evaluation" [Henderson and Morris, 1976; Friedman and Wise, 1979].

The use of lazy evaluation allows programs which manipulate infinite data structures to be written. Only the part of the infinite data structure which is actually required for computation will be constructed. For example, the following definition "integers", written in the programming language SASL [Turner, 1976], will form a list of all the positive integers:

```
def integers = 1 : add1 integers
def add1 x = (hd x) + 1 : add1 (tl x)
```

If lazy evaluation was not available, it would not be possible to

write these particular definitions. Neither of the two definitions, "integers" and "add1", would terminate in an implementation of a reduction programming language which did not support lazy evaluation. However, with lazy evaluation, the value of "integers" is a list; the head of this list is the value '1', whilst the tail of this list contains a value known as a "closure". This "closure" value may be used to evaluate successive elements of the list as they are required.

In passing it is important to recognise that applicative order reduction and normal order reduction have distinctly different mathematical properties. Applicative order reduction is strict, or "bottom preserving", since errors occuring during the evaluation of any argument will be detected during the evaluation of the function application. Normal order reduction is not strict, as any error in the evaluation of an argument will be detected only if the evaluation of the function application requires the value of that argument.

FP [Backus, 1978; Williams, 1982] is a strict reduction programming language based on a set of combining forms rather than the lambda calculus. Lispkit [Henderson, 1978] is a non-strict language based on the lambda calculus. SASL [Turner, 1976], KRC [Turner, 1982], and Miranda [Turner, 1984] are non-strict languages based on combinators.

In an applicative order reduction the arguments to functions may be evaluated concurrently. Since there are no side-effects there is no need to synchronise the evaluation. In a normal order reduction the scope for concurrency is reduced. An argument is evaluated only if the value it denotes is required. The arguments to the basic operations such as the arithmetic operators will always need to be evaluated, but those to user defined functions need not be. The concurrent evaluation of arguments will be limited to those of the basic operations.

3.6 DATA FLOW

A data flow program is based on the flow of data between the individual instructions of the program. Like the reduction model of computation, the data flow model of computation is referentially transparent.

Early designs of programming languages for the data flow model of computation were single assignment programming languages [Ackermann and Dennis, 1978], [Arvind, et al., 1978]. A variable could be assigned to only once and the value remained associated with that variable throughout the execution of the program. However, these single assignment programming languages resembled existing von Neumann programming languages in all other respects. The flow of control through the program was represented by the order in which the instructions were written. Thus, unlike the reduction programming

languages, there was an iterative control statement. To avoid re-assigning to variables within this statement, it was necessary to distinguish between the different values for each iteration. The name of a variable could be used to access the current value associated, or the value associated on the previous iteration.

A more recent development of the data flow model of computation has been based on the idea of streams of values [Kahn and MacQueen, 1977]. A function produces a stream of values from a given stream of values; the function can be thought of as a filter. Other functions are then 'plumbed' onto the input and the output of the function. In this way a program can be constructed. The programming language Lucid [Ashcroft and Wadge, 1977; Wadge and Ashcroft, 1983] allows the programmer to write programs which use streams of values. For example, the statement

```
n = 1 fby n + 1
```

defines the variable "n" to be the sequence of positive integers. Each function takes streams of values as arguments, and may return as a stream of values as a result. Programs written in this programming language consist of a number of statements which define functions and variables. The order of these statements is strictly immaterial since there is no concept of control flow within the programming language.

3.7 CONCLUSION

One of the disadvantages of some of the novel models of computation is that they are too high level and, therefore, it is difficult for the user to influence the actual behaviour of the underlying computing system as a program is executed. In many cases the programmer should not be concerned about the low level details of the computing system. However, it must be recognised that programming is an exercise in good engineering. A well designed program will be one that, among other things, makes reasonable demands upon the resources available on the computing system on which it is executed.

For example, it is easy to write the following program to implement a sort algorithm in the KRC reduction programming language [Turner, 1982]:

```
sort []      = []
sort (a : x) = insert a (sort x)

insert a []      = [a]
insert a (b : x) = a : b : x, a <= b
                   b : insert a x
```

The program is quite transparent; it implements the insertion sort algorithm. Many implementations of reduction programming languages will cause the list which is being sorted to be reconstructed each time the function 'insert' is applied. The space requirement of this particular program would be proportional to the square of the number of items in the list.

The same program can be written in a von Neumann programming language. In the example below the redundancy in space has been overcome by the judicious use of assignment statements:

```
TYPE Table = ARRAY [ 1 .. 100 ] OF INTEGER;

PROCEDURE Sort(VAR T : Table);
  VAR I : INTEGER;

  PROCEDURE Insert(VAR T : Table;
                   Lwb, Upb : INTEGER);
    VAR S : (Scanning, Found, Exhausted);
        I, J, X : INTEGER;
    BEGIN
    S := Scanning;
    I := Lwb;
    REPEAT
      IF I > Upb
        THEN S := Exhausted
      ELSE IF T[I] > T[Upb]
             THEN S := Found
      ELSE I := I + 1
    UNTIL S<>Scanning;
    IF S = Found THEN
      BEGIN
      X := T[Upb];
      FOR J := Upb DOWNTO I + 1
        DO T[J] := T[J - 1];
      T[I] := X
      END
    END { Insert };

  BEGIN
  FOR I := 2 TO 100
    DO Insert(T, 1, I)
  END { Sort };
```

No extra space is required by this program except that needed to represent the auxiliary variables and the return addresses for the procedure calls.

The first program written in the reduction programming language is the easier to understand; the declarative style of programming, of

which it is an example, outlines the effect which is required rather than a procedure by which it may be acquired. However, given the current state of implementations for reduction programming languages, it is not sufficient to stop at that point. Too much detail is hidden by the clarity of expression. The program is a good abstraction of the problem, but it leaves many of the issues unresolved. With reference to this example of a sorting algorithm, Knuth notes that the manufacturers of computing systems have estimated that over one quarter of the execution time used on their computing systems is spent in sorting; indeed, there are some installations where this activity accounts for more than one half of the total execution time [Knuth, 1973].

Until acceptable implementations of the novel models of computation are available, programming in the von Neumann style programming languages will persist. Since the "better" novel models of computation are referentially transparent, it is possible to perform transformations on the programs and yet retain the meaning of the program. This can be used, albeit with limited success, to transform an 'inefficient' program written in a novel programming language into a more efficient program written in a von Neumann programming language. Some of the existing techniques remove certain forms of recursion and replace it by iteration [Burstall and Darlington, 1977]. Another technique, known as "memoisation", is used to reduce the number of times an expression is evaluated [Hughes, 1985]. A survey of the different techniques is given by

Darlington [Darlington, 1987]. However, many of these techniques are insufficient to derive the equivalent programs automatically. Thus, at present, the main practical advantage of the novel models of computation is their clarity of expression and their referential transparency. They may be used to write specifications of programs. These specifications can be refined manually to produce equivalent programs written in the von Neumann style. At each step of the refinement process proofs can be constructed about the correctness of the progress made. Furthermore, at any point in the process an implementation of the novel model of computation may be used to execute the program. This allows an early prototype of the program to be demonstrated [Henderson and Minkowitz, 1986; Turner, 1985].

The referentially transparent programming languages also seem inappropriate for the description of systems which undergo discernible changes as time progresses. The side effects which a program has on the real world may be tangible and important. For example, a software system which controls the behaviour of an industrial process may be able to change the state of the mechanisms which physically control that process, and be able to sense changes in the physical process through those same mechanisms. In a referentially transparent programming language, such changes may be modelled by a data structure which represents the state of the real world. This data structure must be passed as a parameter into every operation and must be returned as a result of every operation. An example of this is given in the database program in [Henderson,

Jones, and Jones, 1983]. Furthermore, a program which has some effect on the real world may have to provide certain stimuli in a specific order. An evaluation mechanism, such as data flow or reduction, where instructions are executed in a non-deterministic order, makes it difficult to construct programs to meet this requirement. Some additional constraints are required to force the evaluation into the desired order.

In general, it has not been shown that these novel models of computation are necessarily the best basis on which to build general purpose programming languages. The usefulness of these programming languages has been demonstrated in certain selected problem domains. For example, the programming language Prolog has been used to construct so-called "expert systems". However, there is little evidence that these programming languages are suitable for significant problems.

Concurrency may be exploited at three distinct levels in the different models of computation. At the lowest level, the processing unit may allow the individual micro-instructions to be executed concurrently. Whether or not this is the case should have no effect on a particular model of computation.

At the next level the operands to operations in the model of computation may be evaluated concurrently. The concurrency at this level cannot be directly controlled by the programmer. For some

models of computation this form of concurrency is inappropriate. For example, the von Neumann model of computation permits operands to have side-effects. The order in which the operands are evaluated may affect the overall behaviour of the program. In models of computation which have the property of referential transparency the operands may be evaluated concurrently without affecting the behaviour of the program.

The degree of concurrency which can be exploited at this second level is probably quite limited. The average number of operands in an expression is not high. This sort of concurrency is best suited to an architecture of computing system in which the processing units fetch instructions from a pool of available instructions. An architecture in which one processing unit explicitly requests some other processing unit to evaluate some of the operands concurrently may well have a large overhead in communication.

The highest level of concurrency is that at which individual statements or routines may be executed concurrently. The concurrency at this level may more reasonably be controlled directly by the user. Allowing individual statements to be executed concurrently may lead to inefficient programs. The cost of communicating a request to execute a statement to another processing unit, together with the overheads involved in the processing unit requesting the information accessed in that statement, may far outweigh the cost in terms of sequential execution of the program.

However, using the routine as the item of concurrency, concurrent execution can be attractive. If the routines of a program have been designed to represent distinct entities in the world modelled by the program, concurrent execution of these routines is a natural outcome.

This is a similar approach to that found within the object oriented model of computation. In particular, the encapsulation of the variables required to represent a particular data structure, together with the routines which are necessary to manipulate these variables into a single object allows the behaviour of the object to be described cleanly in isolation from all other objects. This may be done since the variables of an instance of the object may only be manipulated by the routines described in the textual description of the object. This encapsulation has two additional properties. The representation of structural entities in the real world as objects in the program which models that world is a useful abstraction technique [Kerr, 1987]. Furthermore, the independence of the different instances of objects within a program may permit concurrent execution of those instances. It is known in advance that no variables are shared between the different instances of objects. Consequently, it is not possible for the concurrent execution of the different instances to result in the variables within one instance of an object being in an inconsistent state. However, it is possible that the variables of two or more instances may be inconsistent with respect to each other. The issues of consistency are discussed in chapter five.

4 TWO ALTERNATIVE DESIGNS

Different designs for architectures of computing system and models of computation were outlined in chapters two and three respectively. In those chapeters it was suggested that these designs would not be appropriate to encompass the developments of the future. The computing systems of the not too distant future will consist of a number of heterogenous computing systems connected together by a communications medium. Two alternative designs for an architecture of computing system and the associated model of computation are presented in this chapter.

The first design, recursive control flow, was produced by Treleaven and Hopkins at the University of Newcastle upon Tyne and is described in detail in [Hopkins, 1984]. A formal specification of the behaviour of the design has been constructed and is presented in this thesis. The computing systems built using the recursive control flow principles are recursively structured. Each element of the structure is either a primitive computing element comprising a processing unit, a memory component, and a communications capability, or it may be another structured computing system. The memory component of each computing element is globally accessible.

The second design, decentralised control flow, is presented for the first time in this thesis. Analysis of the recursive control flow model of computation has led to a simplification of the design.

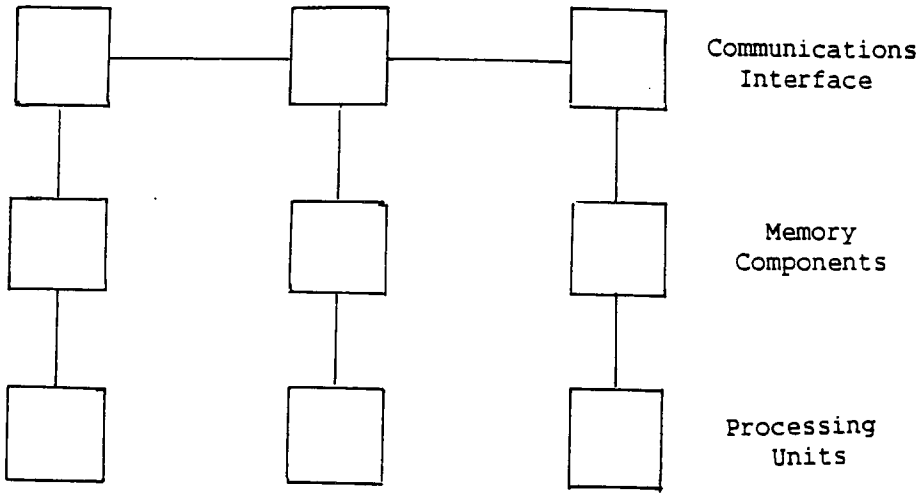

Figure 4.1 A Recursive Control Flow Computing System

The existence of a globally accessible memory component can influence the ease with which good quality software can be produced. Reflection of the globally accessible memory component in the model of computation on which programming languages are based is liable to encourage the programmer to exploit the global accessibility to obtain "efficient" programs. In the decentralised control flow architecture, the memory component of each computing element may be accessed only by the programs which are executed by the local processing unit. This has the distinct advantage that a computing system can be decomposed into the separate parts which together form the whole system. Each part can be considered in isolation from all the other parts. However, the computing systems built from the decentralised control flow principles are also recursively structured and comprise the same elements as those found in the recursive control flow computing systems.

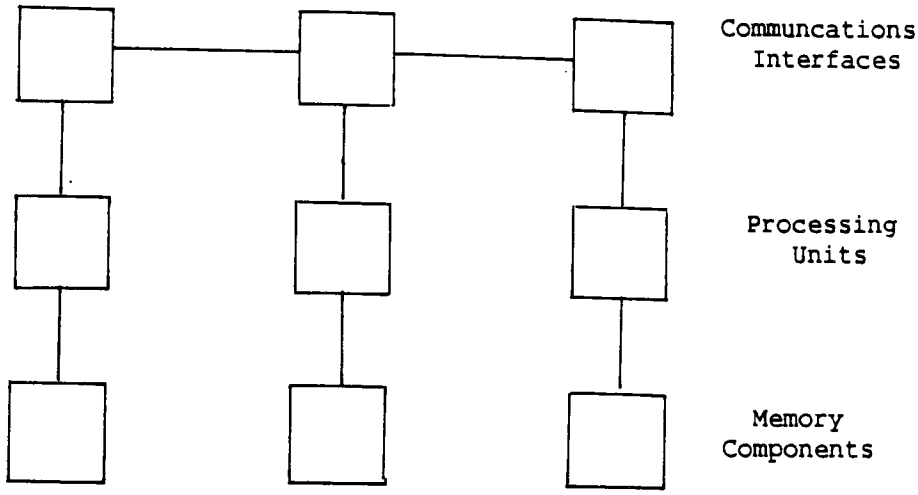

Figure 4.2 A Decentralised Control Flow Computing System

The concept which lies behind the two designs is that of a group of connected subordinate computing systems providing various "services" to the different sequences of instructions which are executed by the individual processing units of the computing system. The nature of the service provided by a subordinate computing system may range from the specific to the general purpose. For example, a specialist computing system such as an array processor could be connected as a subordinate computing system. Alternatively, a subordinate computing system could simply provide a general purpose computing service to support the execution of any program. The concept of service is also reflected in the two models of computation proposed for these architectures. A program which is executed on one subordinate computing system may request a service to be performed by a program which is executed on some other subordinate computing system. On receipt of a request a program may cause the resources attached to the subordinate computing system on which it is executed to undergo a change in state.

For a computing system to be a member of one of the architectures proposed in this chapter, it must adhere to a rigorously defined interface. At the lowest level of communication between a subordinate computing system and the communications medium a specified interface, such as VMEBus [Clement, 1987] or X.25 [Tanenbaum, 1981], must be agreed upon. This will allow information to be transferred between the different subordinate computing systems attached to the communications medium. However, some higher level protocol is also required to permit the transfer of information between the objects supported in the programming languages implemented on the different architectures.

The concept of service underlying the two architectures is not new. It has been in existence at least since the introduction of the IBM 360 series of computing systems. In these systems a special purpose processing unit controlled access to the input/output devices. Commands were received from the central processing unit which were then executed by the special purpose processing unit. The result of executing these commands could cause information to be transferred between the memory component and the input/output devices. Additionally an indication that some state had been reached could be signalled to the central processing unit by the transmission of a message, usually in the form of an interrupt, by the special purpose processing unit. Clearly, in such computing systems, the memory component is globally accessible to both the central processing unit and the special purpose processing unit which

controls the input/output devices. The principles proposed for the two architectures represent an attempt to generalise this existing concept by the introduction of objects between which information may flow.

The motivation for investigating new designs for architectures of computing system and models of computation arose from the prominence given to custom designed silicon chips through the publication of Mead and Conway's book [Mead and Conway, 1980]. It has been suggested that the technology of chip fabrication could be better exploited by the use of designs with regular structures. Memory devices are prime candidates since they are constructed from regular arrays of small devices. Typically, processing units are designed using irregular structures and it could be difficult to exploit the technological advances with such designs. However, the possibility of designing a processing unit with a regular structure has been considered [Treleaven, 1982]. Such a processing unit, together with some memory devices, could be used as the basis of a computing system. The current state of the technology of chip fabrication might only allow the individual components of this computing system to be constructed from several chips. However, as the integration levels rise, it might become possible to construct a single chip which constitutes a complete computing system; it may even become possible to construct a chip containing several computing systems. A board or cabinet might contain several of these computing systems connected together to form a larger computing system.

To take advantage of these possibilities, the principles used for the construction of the computing systems must permit replication [Glushkov, Ignatyev, Myasnikov, and Torgashev, 1974; Wilner, 1980]. Replication will allow subordinate computing systems to be connected together to form a single larger computing system. Use of recursive principles also ensures that a requirement of the technology is met. As integration levels rise, the interconnection paths between subordinate computing systems will shorten. At each level of recursion, a group of computing systems is logically connected to one another. At the lowest level, distinct groups of computing systems are connected together into a single larger computing system. It is possible to arrange these groups physically so that the individual computing systems within each group are physically close thereby ensuring short interconnection paths. The longest interconnection paths will be found at the highest level of recursion.

These principles are not restricted solely to the design of computing system using the technology of chip fabrication. The principles may also be used as a general structuring tool from which computing systems may be constructed from other subordinate computing systems [Randell, 1983]. In particular, the principles of the design are appropriate for the construction of distributed computing systems where the subordinate computing systems might be of the conventional von Neumann style, and the interconnections between the subordinate computing systems might be a local area network.

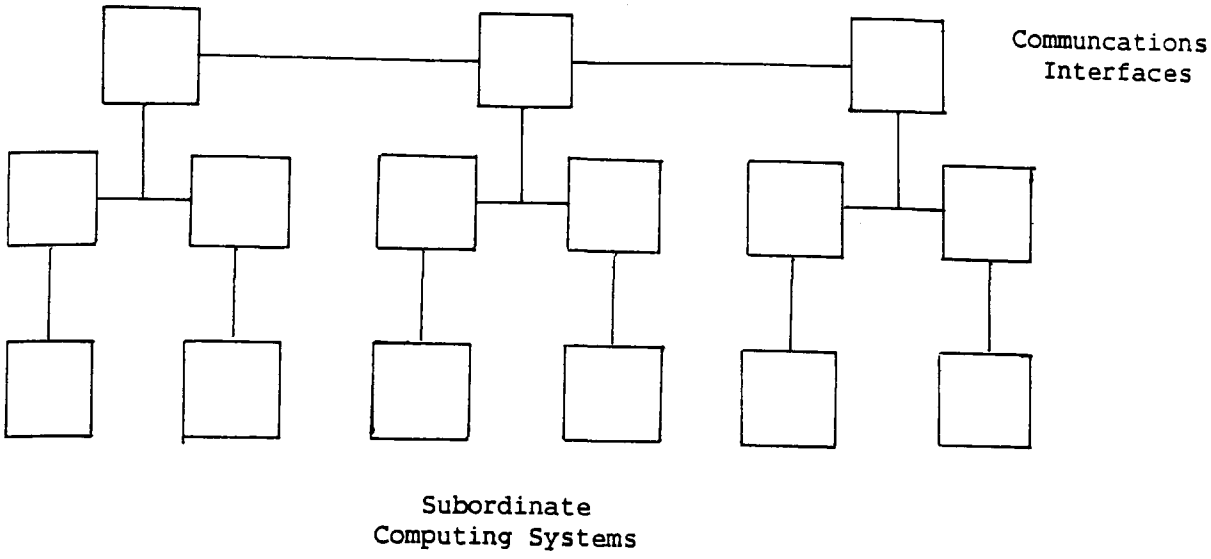

Figure 4.3 A Replicated Computing System

4.1 RECURSIVE CONTROL FLOW

The design of the recursive control flow architecture has been motivated by the possibility of constructing a general purpose computing system from replicated computing elements [Treleaven and Hopkins, 1982]. The computing elements are interconnected to form a larger computing system. Each computing element comprises a processing unit, a memory component, and a communications capability. The communications capability allows different computing systems to be attached to one another so that a hierarchically structured computing system may be constructed. Programs may be executed concurrently by the different processing units of the computing system and information may be transferred between the programs through the use of the globally accessible memory component. The instructions executed by the processing unit of a computing element are based on a synthesis of the concepts underlying the von Neumann, the data flow, and the reduction architectures. This synthesis is

reflected in the recursive control flow model of computation [Treleaven and Hopkins, 1981].

4.1.1 Information Structure

The memory cells of the memory component are hierarchically organised. Each memory cell represents a delimited string, this being a value of arbitrary length which may contain other delimited strings. At the lowest level, the memory cell contains a bit pattern which represents some basic value such as an integer or a character. The memory component of each computing element is itself a member of the total memory structure and each memory cell appears within a particular context in the overall memory structure. The manner in which individual cells within the memory structure are addressed reflects the hierarchic organisation.

An address is a sequence of selectors which identifies a path from the context in which the address appears to the context in which the memory cell addressed appears. For example, the delimited string shown below consists of four subordinate delimited strings. The outermost delimited string is associated with the identifier "A". The four subordinate delimited strings are associated with, from left to right, the identifiers "w", "x", "y", and "z".

```
A: (w: 1  x: (2  4)  y: 3  z: (4  q: 6))
```

The delimited string associated with the identifier "w" consists simply of the integer "1", whilst that associated with the identifier

"x" is itself a delimited string containing the two integers "2" and "4". The delimited string associated with the identifier "x" may be accessed using the address "A/x" or the address "A/1". Similarly, the integer "2" which appears within that delimited string may be accessed from the context "A/x" by the address "/0".

4.1.2 Program Representation

Programs are also stored within the hierarchically organised memory structure. An instruction is specified by an operator which may be followed by a number of operands. A sequence of instructions is delimited by parentheses. The operator of an instruction may be specified in one of three ways:

- an encoding of one of the primitive operations implemented by the processing unit;
- the address of an object which contains a sequence of instructions;
- a sequence of instructions.

In a similar way the operands of an instruction may be specified in one of three ways:

- a literal data item;
- an address of an object which contains a data item;
- a sequence of instructions which, when executed, will yield a data item.

Execution of a sequence of instructions proceeds as follows. At any point during the execution of a sequence of instructions the processing unit is associated with a particular object within the sequence. This is referred to as the locus of control and resembles the "program counter" register found in von Neumann style processing units. As execution of the sequence proceeds, the locus of control is moved from one instruction to the next in the sequence.

For example, the expression "axx + bx + c" may be evaluated by the execution of the following delimited string:

```
(+ (* (+ (* a x) b) x) c)
```

This is perhaps the simplest sequence of instructions which may be written to evaluate the expression. A more complex sequence of instructions is given below, and will form the basis of the discussion of the execution mechanism which follows:

```
(+:= (* a x) b t
 *:= t x ../10
 + c ())
```

4.1.3 Program Execution

The manner in which each instruction is executed depends upon the specification of the operator. An operator specified by an encoding of a primitive operation is executed directly by the processing unit. If the operator has operands, subordinate processing units are associated with the memory cells containing these operands and the values represented by the operands are transmitted from the subordinate processing units to the superior processing unit. When

sufficient values have been received by the superior processing unit, the operation is executed. The subordinate processing units are no longer required and can terminate their activity. The locus of control of the superior processing unit is then moved on to the next instruction in the sequence.

For example, using the sequence of instructions given above, the locus of control is initially placed at the first element of the delimited string. The first element is, in this example, the procedural operator "+:=" which adds its first two operands together, placing the result of this addition in the delimited string addressed by its third operand. In the sequence of instructions given above, this third operand is specified as the address of an element outside the immediate context of the sequence of instructions. Subordinate processing units are activated to evaluate the three operands. On receipt of the values represented by the first two operands, the sum is formed, and this value is then transmitted to the subordinate processing unit associated with the third operand. The locus of control of the superior processing unit is then moved on over the three operands, to the element containing the operator "*:=".

In passing it is worth noting that the instruction with the procedural operator "*:=" has a special form of address as its third operand. The address "../10" specifies the eleventh element of the context in which the address is written. In this instance, the address "../10" specifies the last element of the sequence of

instructions; this element contains the "unknown" value which is represented as "()".

The subordinate processing units are responsible for retrieving the operands for an operator and transmitting the values to the superior processing unit. Some of the operations are classed as functional and execution of such an operation causes a value to be returned to the superior processing unit. Other operations are classed as procedural. Execution of one of these operations causes a value to be stored in a specified memory component. The value is transmitted by the superior processing unit to the subordinate processing unit which has been associated with the result operand. This subordinate processing unit is responsible for storing the value in the memory component.

Thus, for example, in the instruction "+:= (* a x) b t", the subordinate processing unit associated with the operand specified by the address "b", causes the value stored at that address to be transmitted to the superior processing unit. Correspondingly, the subordinate processing unit associated with the operand specified by the address "t" waits until a value is received from the superior processing unit; this value is then stored at that address.

Operators which are specified as addresses cause a subordinate processing unit to be associated with the object addressed. This object should itself be a sequence of instructions. The sequence of

instructions is executed by the subordinate processing unit, the superior processing unit remaining idle during this time. When the locus of control of the subordinate processing unit reaches the end of the sequence of instructions the subordinate processing unit terminates its activity and the superior processing unit becomes active again. A value may be transmitted from the subordinate processing unit to the superior processing unit; this allows a value to be returned as the result of executing some nested sequence of instructions. The locus of control of the superior processing unit is then moved on to the next instruction in the sequence. Arguments may be passed from the context surrounding the memory cell with which the superior processing unit is associated to the subordinate processing unit. These arguments are then used by the subordinate processing unit during the execution of the nested sequence of instructions. A special context register, which refers to the locus of control of the superior processing unit, is initialised in the subordinate processing unit. Operations executed by the subordinate processing unit may then access any arguments by addresses relative to this special context register. The arguments are only evaluated when they are accessed, thereby giving "call by name" semantics.

An operator may also be specified recursively as a sequence of instructions. The execution of the sequence of instructions is performed in precisely the same manner as outlined in the previous paragraph.

The subordinate processing units used to evaluate the operands of an operator execute in one of two possible modes. In one mode the subordinate processing unit is passive whilst waiting to receive messages from the superior processing unit. These messages may cause the subordinate processing unit to perform any one of the following actions:

- 'move' to a different memory cell;
- store some value at the memory cell with which it is associated;
- copy the value at the memory cell with which it is associated into a message to be sent to the superior processing unit;
- execute the contents of the memory cell with which it is associated as a sequence of instructions.

In the alternative mode the subordinate processing unit is active and executes the sequence of instructions contained in the memory cell with which it is associated.

As outlined above, operands are not fetched from memory cells by the processing unit in the classic von Neumann sense. Rather, the values they represent are transmitted to the processing unit by subordinate processing units associated with the memory cell. Similarly, a value is stored in a memory cell by a subordinate processing unit whose locus of control is positioned at that particular memory cell. Retrieving an operand which is specified as a number causes that number to be transmitted by the subordinate processing unit to the superior processing unit. An operand specified as an address causes the locus of control of the

subordinate processing unit to be moved to the memory cell addressed. If the value stored in the memory cell is to be retrieved, it is transmitted by the subordinate processing unit to the superior processing unit. If a value is to be stored in the memory cell, that value is received from the superior processing unit by the subordinate processing unit which then places it in the addressed memory cell. The subordinate processing unit then indicates to the superior processing unit that the value has been successfully stored.

An operand may also be specified recursively as a sequence of instructions. The locus of control is moved to the first instruction in the sequence and then executed. When the sequence of instructions has been executed, a value may be transmitted by the subordinate processing unit to the superior processing unit. Finally, an operand may be specified as the unknown value. The processing unit associated with a memory cell containing the unknown value must wait until some other processing unit has replaced the contents of the memory cell with some other value. The original processing unit may then resume its activity.

The control flow principles are supported by the sequential execution of instructions and the globally accessible memory component. The data flow principles are supported through the use of the unknown value as an operand to an instruction. The reduction principles are supported through the use of nested sequences of instructions and the delayed evaluation of arguments.

In the example of the sequence of instructions given above, all three varieties of principles are demonstrated. The first two instructions show the use of control flow. The first operand of the the first instruction shows the use of reduction, whilst the unknown value which appears in the second operand of the final instruction shows how data flow may be simulated.

4.1.4 Architecture

The description of the execution of a sequence of instructions given above implies that a processing unit could be dynamically associated with any memory component. Clearly this cannot be the case since there is a physical static association between a processing unit and a memory component. The static organisation proposed for the computing elements of a recursive control flow computing system is the tree structure [Hopkins, 1984]. A computing element may consist recursively of other computing elements or may be primitive, in which case it comprises a processing unit, a memory component, and a communications capability.

The dynamic association of a processing unit with any memory component is achieved in the following manner; the activity of a processing unit on a memory component may be transmitted to the processing unit statically associated with that memory component. To enable the migration of activities between processing units, each computing element must support a standard interface. This interface

allows the following commands to be transmitted between two processing units:

- change the locus of control of an activity;
- execute the instructions at the current locus of control;
- copy the contents of the current locus of control to another processing unit;
- replace the contents of the current locus of control by a value;
- terminate the activity.

These commands may be issued at the lowest level, that of the micro-instructions used to implement the instructions of the recursive control flow model of computation [Katz, 1984]. Therefore, all communication between a superior processing unit and a subordinate processing unit takes place beneath the execution of the instructions of the model of computation. The only communication between processing units which can be controlled explicitly by the programmer occurs through the use of the globally accessible memory component or by some value being returned as the result of executing a sequence of instructions.

4.1.5 Model of Computation

A very low level model of computation for the recursive control flow architecture has been outlined [Treleaven and Hopkins, 1981]. The programming language BASIX, described in chapter six, is based on this model of computation. The semantics for this model of computation are given below.

The operators described in the specification are PLUS, IF, GOTO, FORK, and JOIN. The PLUS operator is taken to typify the usual arithmetic and comparison operations. The IF and GOTO operators allow the flow of control within a sequence of instructions to be altered explicitly. The FORK and JOIN operators control the concurrency exploited by a program.

The type Object in the specification represents the memory component of a recursive control flow computing system. It may be thought of as a delimited string. The individual elements of a delimited string, the memory cells, are represented by the type Component. This type contains not only the primitive types such as Number and Address, but also Object. This recursive type allows the hierarchical memory component to be represented.

The type State represents the state of the whole recursive control flow computing system at any point in time. The components Current and Root of this type represent, respectively, the locus of control of the sequence of instructions currently being executed and the locus of control of the special context register. Thus, the component Root allows arguments to be accessed during the execution of a sequence of instructions.

```
Object    = seq of Component

Component = Number U Address U OpCode U Object

Address   = seq of { NEXT, PRIOR, IN, OUT, ROOT }

OpCode    = { PLUS, *PLUS, MINUS, *MINUS, .., GOTO, FORK, .. }

Path      = seq of N1

State     :: Memory  : Object
             Current : Path
             Root    : Path

EvalArg(c : Component, s : State) v : [Component], s' : [State] =
  if c in Number
  then v, s' = c, s
  else if c in Address
       then let p = MakePath(Current(s), Root(s), c)
            in  v, s' = Fetch(p, Memory(s)), s
  else if c in Object
       then v, s' = EvalList(1, c, s)
  else v, s' = nil, nil

EvalList(n : N1, o : Object, s : State) v : [Component], s' : [State] =
  if not (n in dom o)
  then v, s' = nil, s
  else let c = o(n)
       in  if c in Number
           then if not (n + 1 in dom o)
                then v, s' = c, s
                else v, s' = EvalList(n + 1, o, s)
           else if c in Address
                then let p = MakePath(Current(s) ~ [n], Root(s), c)
                     in  let s0 = mu(s, Current -> p,
                                        Root -> Current(s) ~ [n])
                             c = Fetch(p(1 .. len p - 1), Memory(s))
                         in let v1, s1 = EvalList(p(len p), c, s0)
                            in  let s2 = mu(s, Memory -> Memory(s1))
                                in  if not (n + 1 in dom o)
                                    then v, s' = v1, s2
                                    else v, s' = EvalList(n + 1, o, s2)
           else if c in Object
                then let s0 = mu(s, Current -> Current(s) ~ [n],
                                    Root -> Current(s) ~ [n])
                     in  let v1, s1 = EvalList(1, c, s0)
                         in  let s2 = mu(s, Memory -> Memory(s1))
                             in  if not (n + 1 in dom o)
                                 then v, s' = v1, s2
                                 else v, s' = EvalList(n + 1, o, s2)
```

```
          else if c = PLUS
               then v, s' = ExecPLUS(n, o, s)
          else if c = *PLUS
               then v, s' = Exec*PLUS(n, o, s)
            ..
          else if c = GOTO
               then v, s' = ExecGOTO(n, o, s)
          else if c = FORK
               then v, s' = ExecFORK(n, o, s)
          else ..

ExecPLUS(n : N1, c : Component, s : State) v : [Component], s' : State =
  let s0 = mu(s, Current -> Current(s) ~ [n + 1])
  in  let v1, s1 = EvalArg(o(n + 1), s0)
      in  let s2 = mu(s1, Current -> Current(s) ~ [n + 2])
          in  let v2, s3 = EvalArg(o(n + 2), s2)
              in  let s4 = mu(s, Memory -> Memory(s3))
                  in  if not (n + 3 in dom o)
                      then v, s' = (v1 + v2), s4
                      else v, s' = EvalList(n + 3, 0, s4)

Exec*PLUS(n : N1, c : Component, s : State) v : [Component], s' : State =
  let s0 = mu(s, Current -> Current(s) ~ [n + 1])
  in  let v1, s1 = EvalArg(o(n + 1), s0)
      in  let s2 = mu(s1, Current -> Current(s) ~ [n + 2])
          in  let v2, s3 = EvalArg(o(n + 2), s2)
              in  let s4 = Assign(n + 3, o, s3, (v1 + v2))
                  in  let s5 = mu(s, Memory -> Memory(s4))
                      in  if not (n + 4 in dom o)
                          then v, s' = nil, s5
                          else v, s' = EvalList(n + 4, o, s5)

ExecGOTO(n : N1, c : Component, s : State) v : [Component], s' : State =
  let p = MakePath(Current(s) ~ [n + 1], Root(s), o(n + 1))
  in  let v1, s1 = EvalList(0, Fetch(p, Memory(s)), mu(s, Current -> p))
      in  v, s' = EvalList(n + 2, o, mu(s, Memory -> Memory(s1)))

ExecFORK(n : N1, c : Component, s : State) v : [Component], s' : State =
  let p = MakePath(Current(s) ~ [n + 1], Root(s), o(n + 1))
  in  let v1, s1 = EvalList(0, Fetch(p, Memory(s)), mu(s, Current -> p))
      in  v, s' = EvalList(n + 2, o, mu(s, Memory -> Memory(s1)))

MakePath(cc, rc : Path, a : Address) p : Path =
  if len a = 0
  then p = cc
  else let cc1 = if a(1) = NEXT
                 then Succ(cc)
                 else if a(1) = PRIOR
                      then Pred(cc)
                 else if a(1) = IN
                      then cc ~ [0]
                 else if a(1) = OUT
                      then cc(1 .. len cc - 1)
                 else if a(1) = ROOT
                      then rc
                 else cc
       in  p = MakePath(cc1, rc, tl a)
```

```
Succ(p : Path) p' : Path =
  p' = p(1 .. len p - 1) ~ [p(len p) + 1]

Pred(p : Path) p' : Path =
  p' = p(1 .. len p - 1) ~ [p(len p) - 1]

Fetch(p : Path, c : Component) c' : Component =
  if len p = 0
  then c' = c
  else c' = Fetch(tl p, c(p(1)))

Assign(n : N1, o : Object, s : State, v : Component) s' : State =
  let c = o(n)
  in  if c in Number
      then s' = mu(s, Memory -> Replace(Memory(s), Current(s) ~ [n], v)
      else if c in address
           then let p = MakePath(Current(s), Root(s), c)
                in s' = mu(s, Memory -> Replace(Memory(s), p, v))
      else s' = s

Replace(c : Component, p : Path, v : Component) c' : Component =
  if len p = 0
  then c' = v
  else c' = c(1 .. p(1) - 1) ~
              [Replace(c(p(1)), tl p, v)] ~
                c(p(1) + 1 .. len c)
```

The semantics of the recursive control flow model of computation given are deficient in one important respect. It has not been possible to show the concurrency which is supported by the architecture and which can be expressed in the model of computation. For example, evaluation of the operands of operators such as PLUS may be performed concurrently. Any changes in the memory component of the overall computing system made during the evaluation of one operand must be visible during the evaluation of the other operand. The formal specification of the recursive control flow model of computation given above implies that the evaluation of the operands is performed serially. Similarly, it has not been possible to specify the behaviour of the JOIN operator. This operator causes a subordinate processing unit to terminate its activity. The superior processing unit which caused that subordinate processing unit to be activated is notified that one of its subordinate processing units has indeed terminated its activity. If the superior processing unit was waiting for the termination of all of its subordinate processing units it would now be possible for it to continue its own execution. This low level description needs the flow of information and control between the different processing units which are involved in the execution of a program to be modelled.

4.1.6 Concurrency in the Model of Computation

It would be possible to adapt the existing specification by the inclusion of more detail. The FORK and JOIN operators may be modelled by specifying the state of the individual processing units. The complete state of the computing system may be specified as follows:

```
Activity :: Current  : Path
            Root     : Path
            Children : seq of ActivityId
            Parent   : ActivityId
            Status   : { Active, InActive }

System   :: Activites : map ActivityId to Activity
            Memory    : Object
```

Each processing unit has its own state which indicates the status of that processing unit. It may either be active, which implies that the processing unit can execute instructions, or it may be inactive which implies that the processing unit is waiting for the subordinate processing units which it activated to terminate. Operationally one processing unit may be selected from the group of processing units in the system and a single instruction or sequence of instructions executed. The FORK and JOIN operators may now be specified more precisely. Execution of the FORK operator causes a new processing unit to be added to the group of processing units in the system. Execution of the program of this new processing unit proceeds concurrently with that of the existing processing units of the system. This concurrent behaviour is simply modelled by the serial interleaving of the execution of the instructions of the different

processing units of the group. Execution of the JOIN operator causes the processing unit to be removed from the group of processing units in the system. The processing unit which caused this subordinate processing unit to be activated is notified that the JOIN operator has been executed by one of its subordinate processing units.

The specification at this level shows how execution of the processing units may be modelled by serialisation of the execution of the individual instructions of the processing units. Different orderings of the individual instructions of a group of processing units may result in the computing system reaching different states.

For example, consider the concurrent execution of the following two sequences of instructions:

```
(:= t1 b                (:= t2 b
 := b  (+ t1 100))       := b  (+ t2 200))
```

There are six different orderings of the execution of these instructions. Only two of these orderings ensure that the value stored at the delimited string associated with the identifier "b" is incremented by 300. Of the remaining four orderings, two result in the value being incremented by 100, whilst the other two result in the value being incremented by 200.

However, even this more detailed specification does not reflect the true nature of the recursive control flow architecture and its model of computation. Modelling the actual behaviour requires

further detail, this time at the level of the micro-instructions of the recursive control flow architecture. Since the individual operands of an instruction may be evaluated concurrently, the flow of information between the subordinate processing unit and the superior processing unit must be modelled. It is not sufficient to model the instructions of the recursive control flow architecture atomically. It is necessary to model the micro-instructions which are used to implement the instructions of the recursive control flow architecture. Again the description of the computing system consists of a group of processing units. Each processing unit may execute a micro-program which implements the recursive control flow instructions.

For example, the PLUS operator will cause the two operands to be evaluated concurrently by two subordinate processing units. Execution of the PLUS operator thus takes place in several distinct stages. Firstly, the subordinate processing units are activated and each subordinate processing unit evaluates its operand thereby sending a message to the superior processing unit to indicate the value of that operand. Then, on the basis of these messages, the superior processing unit can calculate the value and may then send a message to some superior processing unit indicating the result of that calculation. All these processing units may execute instructions from their micro-programs concurrently.

The model required to specify the behaviour of the recursive control flow architecture at this low level of detail is somewhat similar to that required at the higher level. However, it is now the individual micro-instructions which are being modelled. The steps by which an operator is implemented have become visible. Again, different orderings of the execution of the micro-instructions of the group of processing units may lead to the computing system reaching different states.

For example, consider the following two instructions:

(+:= b 100 b) (+:= b 200 b)

If these two instructions are executed concurrently, then subordinate processing units will be activiated which executed the microprogram sequences concurrently. The execution of each instruction given above requires three subordinate processing units. Two of these processing units transmit the values of the operands of the instruction to the superior processing unit, whilst the third processing unit receives the value to be assigned to the delimited string associated with the identifier "b". Again, there are six orderings for the transmission of values between the subordinate processing units associated with the operands specified by the address "b" and the superior processing units associated with each instruction. Two of these orderings will result in thew value stored at the delimited string associated with the identifier "b" being incremented by 300, whilst, of the remaining four orderings, two will cause that value to be incremented by 100, and two will cause it to

be incremented by 200.

To understand the behaviour of the recursive control flow model of computation completely, this detailed specification must be constructed. The reason for the complexity of the specification lies mainly in the existence of the globally accessible memory components. The model of computation supports both concurrency and a globally accessible memory component and therefore the order in which instructions and operands are evaluated can have an effect on the state of the memory component. The communication between processing units which occurs at the level of the micro-instruction is visible at the level of the execution of the individual instructions. The model of computation seems unnecessarily complex; in practice, restrictions would need to be introduced into the programming languages used to write software for the recursive control flow computing systems in order to restrict the model of computation. Typically such restrictions would prohibit the combined use of concurrency and the globally accessible memory component. This would simplify the specification by hiding the communication between the processing units which occurs at the level of the micro-instruction.

4.2 DECENTRALISED CONTROL FLOW

The complexity of the recursive control flow model of computation does not reflect the apparent simplicity of the underlying principles. The simplicity lies in the recognition that computing systems can be constructed recursively from heterogenous autonomous computing elements. The complexity arises from the organisation of these elements into a hierarchical structure in which the memory components of the individual computing elements are globally accessible.

The decentralised control flow architecture and its associated model of computation are a refinement of the recursive control flow principles. The important difference is the absence of a globally accessible memory component and a restriction on the nature of side-effects within operand execution. The work on abstract data types and the object oriented model of computation has directed this refinement process.

The decentralised control flow architecture assumes a network of subordinate computing systems, each of which is autonomous. Each subordinate computing system has a memory component which may only be accessed by the programs which are executed on the processing unit of that subordinate computing system. This coupling of the memory component and the processing unit into a single entity reflects both an abstract data type and the object oriented model of computation.

An abstract data type consists of a description of the variables required to represent some data structure and a description of the routines which are necessary to manipulate those variables. An instance of an abstract data type consists of a local data space for the representation of the variables, together with the code for the routines. The variables of an instance may only be manipulated by the routines of that instance. As suggested in chapter three, the central concept of the object oriented model of computation, that of the object, has been taken from the work on abstract data types. Consequently, the decentralised control flow principles could be used to support an implementation of an object oriented programming language. Taking this approach gives a mechanism for constraining the concurrency; this is outlined in the remainder of this chapter and is developed in more detail in chapter five.

4.2.1 Information Structure

A general purpose distributed computing system consists of a number of subordinate computing systems connected together by some communications medium. The scale of distribution will not affect the behaviour of the distributed computing system except for certain details such as the time taken to transmit information across the communications medium. Each subordinate computing system has resources attached to it which may be accessed only by those programs which are executed by the processing unit of the subordinate computing system. There is no memory component which is accessible

to all the subordinate computing systems. Information may be transmitted from one subordinate computing system to another subordinate computing system by the sending and receipt of messages using the communications medium.

There is no reason why the resources attached to a subordinate computing system should be limited to a memory component and input/output devices. A communications medium could itself be an attached resource. The use of this resource would be controlled by a program which is executed by the processing unit of the subordinate computing system. This permits the construction of hierarchically structured distributed computing systems.

4.2.2 Program Representation and Execution

Each subordinate computing system is capable of executing a sequence of instructions. These instructions may make references to the resources which are attached to the subordinate computing system on which it is executed. A sequence of instructions may be totally independent of all other sequences. The subordinate computing system on which such a sequence of instructions is executed can have no effect on any other subordinate computing systems nor can it be effected by any other subordinate computing system. Alternatively, a group of sequences of instructions may be designed to interact. However, there is no globally accessible memory component which may be used for the transfer of information between the execution of

sequences of instructions on different subordinate computing systems. The communications medium which connects the subordinate computing systems together into a distributed computing system is used to transfer information between different subordinate computing systems. Rather, the information to be transferred between subordinate computing systems is placed in a message which is transmitted from one subordinate computing system to another.

The protocol used for the transfer of information between subordinate computing systems is based on the semantics of the procedure call as found in programming languages such as Pascal. Thus the transfer of information is viewed as a two-way process. The source of the information creates a message containing the information to be transmitted to the destination. At the destination, the information is processed and some response is then transmitted back to the source.

The message transmitted from the source subordinate computing system to the destination computing system will indicate what sort of processing is required. Thus the messages transmitted across the communications medium between the source subordinate computing system and the destination subordinate computing system consist of the following two pieces of information:

- a field to identify what processing is required;
- a field or group of fields which contain the information to be processed.

Additionally, the communications subsystem will require a field to identify the destination subordinate computing system, and a field is also required to identify the source subordinate computing system so that a response may be made to the message. This response consists solely of the information to be transferred from the destination subordinate computing system to the source subordinate computing system. A tag field is required to distinguish between the two types of messages which may be transmitted between subordinate computing systems. The messages are categorised as requests and responses. The following declaration of a variant record in the programming language Pascal outlines the format of a message:

```
TYPE Message = RECORD
                 Source      : SystemID;
                 Destination : SystemID;
                 CASE Tag : (Request, Response) OF
                   Request: (Action : ActionID;
                             RequestValues : ListOfValues);
                   Response: (ResponseValues : ListOfValues)
               END
```

Associated with each subordinate computing system is a table of entries which allows the field identifying the variety of processing to be mapped into a particular sequence of instructions. On receipt of a message bearing a request, the specified action is looked up in the table. If an entry is found, the sequence of instructions indicated by the entry is executed and a response transmitted. However, if no entry is found which contains the specified action, a standard response is transmitted indicating a failure to find the required action.

In passing it should be noted that the effects of a globally accessible memory component can be obtained by ensuring that every subordinate computing system within the distributed computing system has the actions 'read' and 'write'. Indeed, it would appear that a recursive control flow computing system is subsumed by a decentralised control flow computing system. The actions of the subordinate computing systems of the former may be implemented explicitly in the latter.

The procedure call semantics enforce the transmission of a response from the destination subordinate computing system to the source subordinate computing system. Furthermore, the source computing system cannot continue execution of the sequence of instructions which originated the request until the response has been received. Consequently, the procedure call semantics restrict the degree of parallelism which may be exploited. This is in sharp contrast to those distributed computing systems in which the transfer of information occurs by the sending and receipt of messages. In such computing systems there is a high degree of parallelism.

The details of the operations actually supported by a particular subordinate computing system are irrelevant to the other subordinate computing systems in the distributed computing system since one subordinate computing system does not have access to another subordinate computing system except through the transfer mechanism outlined above. Consequently, the actual set of operations

implemented by a particular subordinate computing system can be special purpose or general purpose. Furthermore, since the resources attached to a particular subordinate computing system cannot be accessed except through the agency of the sequence of instructions executed by that subordinate computing system, the specific details of these resources can be hidden from all other subordinate computing systems in the decentralised control flow computing system.

Since the semantics of the transfer mechanism between two subordinate computing systems are based on those of the procedure call, there may only be a single thread of control within a program. Consequently, it is not possible for different parts of a program to be executed in parallel. This is an obvious disadvantage of the organisation proposed here. To increase the degree of parallelism available, the semantics of the transfer mechanism must be redesigned. The execution of the sequence of instructions by the source subordinate computing system may proceed rather than be delayed until a response has been received from the destination subordinate computing system. Thus there are two distinct threads of control active simultaneously.

4.2.3 Model of Computation

The formal semantics of the decentralised control flow model of computation are given below:

```
System = map SystemId to SubordinateSystem

SubordinateSystem :: Data        : map Name to Value
                     Operations  : map Name to seq of Statement
                     Code        : seq of Statement
                     PC          : N
                     ConnectedTo : set of SystemId

Meaning(S : System, I : SystemId, A : seq of Value)
        S' : System, R : map Name to Value
  let Instr = Code(S(I))(PC(S(I)))
  in case Instr of
       Return: S' = S + { I -> mu(S(I), Code -> [], PC -> 0) }
               R  = Results(Instr)

       Call:   let S1, R1 =
                    Meaning(S +
                              { Dest(Instr) ->
                                 mu(S(Dest(Instr)),
                                     Code ->
                                      Operations(S(Dest(instr))
                                       (Op(Instr)),
                                      PC -> 1) },
                            Dest(Instr),
                            Arguments(Instr))
               in S', R =
                   Meaning(S1 +
                            { I ->
                               mu(S1(I),
                                   Data -> Data(S1(I)) + R1,
                                   PC -> PC(S1(I)) + 1) },
                           I,
                           [])

       Assign: S', R = Meaning(S +
                                { I ->
                                   mu(S(I),
                                       Data ->
                                        Data(S(I)) +
                                         { Name(Instr) ->
                                            Expression(Instr) },
                                       PC -> PC(S(I)) + 1) },
                               I,
```

[])

The formal semantics given above model the behaviour of a decentralised control flow computing system from the viewpoint of a single instance of an object which may send requests to instances of other objects. The semantics show clearly that the variables associated with an instance are local to that instance, and that only the routines local to an instance may change the variables of that instance.

4.2.4 Concurrency in the Model of Computation

The semantics outlined below show the effect of concurrent execution of the different instances of objects within a decentralised control flow computing system. It is assumed that an instance can only respond to a single request at a time. Consequently, the different requests to an instance are serialised.

```
System = map SystemId to SubordinateSystem

SubordinateSystem :: Data       : map Name to Value
                     Operations : map Name to seq of Statement
                     Code       : seq of Statement
                     PC         : N
                     Requests   : seq of Request
                     Results    : map SystemId to seq of Value
                     Status     : { Busy, Waiting }
                     ConnectTo  : set of SystemId

Request  :: Operation : Name
            Requester : SystemId
            Arguments : seq of Value

Meaning(S : System) S' : System
  let I = Choose(dom S)
  in if Status(S(I)) = Waiting
     then S' = S
     else if PC(S(I)) in dom Code(S(I))
```

```
        then S' = Execute(S, I)
        else if len Requests(S(I)) = 0
             then S' = S
             else S' = Execute(S + { I ->
                       mu(S(I), Code ->
                       Operations(S(I))
                         (Operation(Requests(S(I))(1))),
                       PC -> 1,
                       Status -> Busy)}, I)

Execute(S : System, I : SystemId) S' : System
  let Instr = Code(S(I))(PC(S(I)))
  in case Instr of
       Return: S' = S +
                    { I ->
                       mu(S(I),
                           Code -> [],
                           PC -> 0,
                           Requests ->
                            tl Requests(S(I))),
                      Requester(Requests(S(I))) ->
                       mu(S(Requester(Requests(S(I)))),
                           Results -> Result(Requester(S(I))) +
                                      { I -> Results(Instr) },
                           Status -> Busy) }

       Call:   S' = S +
                    { I ->
                       mu(S(I),
                           PC -> PC(S(I)) + 1),
                      Dest(Instr) ->
                       mu(S(Dest(Instr)),
                           Requests ->
                            Requests(S(Dest(Instr))) ~
                              [ MakeRequest(Op(Instr),
                                            I,
                                            Arguments(Instr))])


       Wait:   if Name(Instr) in dom Results(S(I))
               then S' = S +
                         { I ->
                            mu(S(I),
                                PC -> PC(S(I)) + 1,
                                Status = waiting) }
               else S' = S

       Assign: S' = S +
                    { I ->
                       mu(S(I),
                           Data ->
                            Data(S(I)) +
```

{ Name(Instr) ->
Expression(Instr)})

It has been easier to specify the behaviour of the decentralised control flow model of computation than it is for that of the recursive control flow model of computation. Any part of the globally accessible memory component in the distributed computing system built on the recursive control flow principles may be altered by any of the sequences of instructions which happen to be executed by the subordinate computing systems. It is not possible to order these alterations. In contrast, because access to the memory components of a distributed computing systems constructed on the basis of the decentralised control flow model is strictly limited to the sequence of instructions executed by the processing unit of the subordinate computing system to which the particular memory component is attached, the various alterations which are made to the memory components can be ordered. The memory component of a particular subordinate computing system within a decentralised control flow computing system is isolated from all the processing units except that of the subordinate computing system to which it is attached. Consequently, the memory component is only altered by the sequence of instructions executed by that processing unit.

4.3 CONCLUDING REMARKS

Two different designs for the construction of distributed computing systems have been presented in this chapter. Both designs are based on the concept of recursive structuring. Any component within a distributed computing system may be atomic in that it cannot be decomposed into other components, or it is compound in which case it is composed of other subordinate components. The use of recursion allows the designs to be applicable to a wide range of distributed computing systems. At one extreme, a distributed computing system may be thought of as a subordinate component of some other distributed computing system. At the other extreme, that same distributed computing system can be thought of as a group of individual components connected together by some communications medium into a single computing system. The designs aim to present a distributed computing system both as a complete computing system and as a computing system in which components may be freely inserted and removed.

These two aims have been described, separately, in the literature. Producing a complete computing system from several distinct subordinate computing systems has been the objective of several groups of researchers. For example, the Newcastle Connection [Brownbridge, Marshall, and Randell, 1983] is a software system which allows the filestores of several computing systems to be viewed as single entity. A user of one computing system could access the

filestores of the other computing systems without realising that the information was actually associated with some other computing system. This is achieved by extending the names which are valid within each computing system's filestore to include entries for those of the other computing systems. In the case of computing systems executing the Unix operating system, this extension to the filestore is conceptually simple. Each filestore is arranged as a hierarchy; the Newcastle Connection presents an enlarged hierarchy to the user which encloses the filestores of the different computing systems.

At the hardware level, the design for a recursive machine [Wilner, 1980] is based on replicated subordinate computing systems which are organised into a single computing system. This design has been proposed as a possible technique to exploit the potentials of the technology of chip fabrication. However, the structuring principle proposed, recursion, lends itself to wider application.

The principles of both recursive control flow and decentralised control flow allow subordinate computing systems to be connected together into a single computing system. Each design permits sequences of instructions to be executed in parallel by the subordinate computing systems and for information to be transferred between the different subordinate computing systems. The essential difference between the two designs lies in the nature of the parallelism and the transfer of information.

In the recursive control flow design there is complete freedom in the exploitation of parellelism through the explicit use of the primitive operators "FORK" and "JOIN" and implicitly during the evaluation of operands. Additionally, the memory components of each subordinate computing system are globally accessible. Recursive control flow reflects quite clearly the designs of the Newcastle Connection and the recursive machine. The globally accessible memory components and the unrestricted parallelism do, however, have certain drawbacks.

The foremost difficulty is that of producing a formal description of the behaviour of a recursive control flow computing system. The description produced earlier in this chapter depends upon detailed knowledge of the information which is transferred between different subordinate computing systems as an instruction is executed, or an operand is evaluated. It has not been possible to describe the behaviour of the system in terms of the primitive operators alone; details of the mechanisms required to implement the execution of the instructions appears in the description.

Secondly, the formal description is difficult to understand since it has been impossible to hide the details of recursive control flow which should be irrelevant at this level of description. It will not be easy to use the description to reason about the behaviour of sequences of instructions since there is no abstraction away from the level of details of recursive control flow. Any reasoning about the

behaviour of a sequence of instructions must be considered in terms of these low level details.

In contrast, the description of the behaviour of a decentralised control flow computing system is easier to produce simply because the individual subordinate computing systems of such a computing system can be considered in isolation from one another. The interface between each subordinate computing system is clearly defined, and the behaviour of the whole distributed computing system is based on this interface.

5 ISSUES OF CONCURRENCY IN DISTRIBUTED COMPUTING SYSTEMS

The decentralised control flow model of computation uses the concept of an object as the most basic level of structuring. An instance of an object encapsulates both the data structures and the control structures found in programs written in von Neumann style programming languages. Requests may be sent to an instance of an object, interpreted by the instance and changes made to the data structures local to that instance. Such changes made to the data structures are performed by the control structures local to the instance. A change in the state of an instance of an object occurs when some change is made to the data structures of the instance. This change in the state is "visible", or "discernible", to other instances of objects if it affects the future behaviour of the instance.

A large or complex piece of software may require several different objects. The specification of the software describes the relationships between these different objects. After each and every change in the state of any instance of an object, it is important that these relationships hold. When the relationships hold, the group of instances of objects is said to be in a consistent state. Should the relationships be found not to hold, the group of instances of objects has reached an inconsistent state. For example, a group of objects which represents information about a banking system would probably have the additional restriction that the sum of money

modelled by instances of those objects must be invariant.

The majority of models of computation and programming languages are based on a strictly sequential flow of control in which the execution of a program occurs in isolation from all other programs. Any inconsistencies which arise in a program stem from a failure on the part of the programmer to produce a correct sequence of instructions. Typically, research into concurrency has been constrained to those issues arising within a single program. For example, the programming language Pascal has been extended so as to permit the concurrent execution of statements [Ben-Ari, 1982]. However, the resultant programs are considered in isolation from one another. Similarly the concurrency which may be exploited in control programs has been restricted to single computing systems. Both are inappropriate because they assume a centralised computing system as their basis.

Some of the issues relating to concurrency are discussed in [Liskov, 1981]. Whilst the concern of that paper is toward mechanisms to support robust software which will survive failures in the communications medium, the same mechanisms may be used to ensure that the group of objects in a computing system remain in a consistent state. Much of the work in this area borrows techniques and solutions from work already undertaken in the area of distributed databases.

The work on Actors reported by Hewitt and Baker [Hewitt and Baker, 1977] represents an attempt to introduce some formalism into a model of computation for distributed computing systems. Their actor theory is a formalisation of the object oriented model of computation based on message passing. However, this formalism is not taken sufficiently far to guarantee the coherency of state for distributed computing systems.

Schlageter [Schlageter, 1978] outlines some areas in which the issues of concurrency in database systems are more complex than those for sequential control programs:

- the enormous number of resources to be controlled;
- a process may work on a variable number of resources;
- the resources may be addressed associatively;
- the contents of the resources are connected by arbitrarily complex consistency constraints;
- the set of resources may vary with time.

These issues are, with the exception of associative addressing, also applicable to distributed computing systems.

In this chapter a method is introduced for ensuring that the group of objects used by different users is maintained in a consistent state. Execution of a program which interacts only once with only one globally accessible object will maintain, in a consistent state, the computing system of which that object is a part if the following two conditions hold:

- the state of the computing system immediately before the program interacts with the object;
- the state of the computing system immediately after the program interacts with the object.

Let S(t) represent the state of the computing system at some time instant t. If that computation C occurs at a time instant i, and takes k time instants before completion, then the following must hold if that occurrence of the computation C is to be regarded as safe:

- S(i) must be consistent;
- S(i+k) must be consistent.

It is assumed that the state S(i+k) has been reached by the occurrence of the computation C in the state S(i). Furthermore, repetition of the occurrence of the computation C in the state S(i) must yield the state S(i+k). Coherence of the state of the computing system relates to the topics of integrity and consistency in database theory.

5.1 INTEGRITY AND CONSISTENCY OF OBJECTS

Many of the issues of concurrency have already been resolved through research into distributed database management systems. In this section, those issues which are pertinent to distributed computing systems are introduced.

The valid states of an instance of an object or group of instances of objects in a computing system may be described by an invariant.

The invariant must hold before the execution of any sequence of instructions which changes the state of an instance or group of instances. If it can be guaranteed that execution of the sequence of instructions will cause a valid state to be reached then it can be guaranteed that the integrity of the computing system will be maintained. At some point during the execution of the sequence of instructions it may be the case that the invariant does not hold. It is important that such states are purely transient and are not made visible to other sequences of instructions which may be executed concurrently.

Many large or complex programs are constructed from a hierarchy of instances of objects. The effect of executing one of the routines of one of these instances may allow an instance to reach an inconsistent state with respect to the group of instances of objects as a whole. This is often the case in real programs. An object may be designed to perform a general task which partially transforms the initial state towards the desired final state. The object will not necessarily be orientated towards the specific problem domain of the whole program. However, the overall effect of the execution of a program must be to take a group of instances of such objects from one consistent state to some other consistent state. The inconsistent states which may arise during the execution of the program must not be visible to other programs. This restriction is necessary to ensure that some other program is not executed in an initially inconsistent state. Furthermore, the execution of the program must

be atomic. It must either be executed to completion in order that the instances of the objects reach a new consistent state, or it must not execute at all and the instances of the objects should remain in their initial consistent state. Partial execution is forbidden as it might lead to a group of instances of objects reaching an inconsistent state.

Guaranteeing that the execution of a sequence of instructions will cause a computing system to remain in a valid state can be divided into two distinct tasks. Potential failures may be detected by scanning the text of the program to check that the individual instructions do not violate certain static constraints. For example, it is possible, given suitable type information, to ensure that the values assigned to variables within the local data space of an instance are within a specific range. This form of checking is performed by most programming language compilers. Unfortunately, a large set of invalid states may still be reached from a program which has been statically checked. For example, two or more variables within the local data space of an instance of an object may be related by some invariant. It may not be possible to ensure that the relationship between such variables remains invariant simply by scanning the textual description of the object. More seriously, a group of instances of objects may reach an inconsistent state as a result of the concurrent interactions of requests sent to those instances. These inconsistent states arise as a result of the dynamic behaviour of the program. When programs are executed

concurrently and may interact with a group of globally accessible instances of objects some mechanism is required to ensure that the group of instances of objects within a computing system remain in a consistent state.

Consider the two following sequences of instructions:

```
BEGIN                          BEGIN
x := [R Get]                   y := [R Get]
[R Put f(x)]                   [R Put g(y)]
END                            END
```

The notation "x := [R Get]" means that the request "Get" is sent to the instance of an object identified by the local variable "R". The response to this request is placed within the local variable "x". The variable "R" will contain a reference to an instance of an object to which the request "Get" may be sent. It is assumed that the variable "R" will be assigned that reference when the sequence of instructions is executed. Similarly, the notation "[R Put f(x)]" means that the request "Put" is sent to the instance of an object identified by the variable "R". This request also contains the value obtained by evaluating the expression "f(x)". It is assumed that the local variable "R" in each sequence of instructions contains a reference to the same instance of an object.

Both sequences of instructions interact with the globally accessible instance of an object referenced through the local variable "R". The initial state of this instance before the execution of either sequence of instructions may be represented as

"r". In the following section the concurrent execution of the two sequences of instructions is considered.

It is assumed that the instance of the object referenced through the local variable "R" cannot respond to more than one request at any time. This restriction forces the requests sent to an instance of an object to be serialised. The requests made of the globally accessible instance may be arranged as six possible orderings, known as schedules [Eswaran, Gray, Lorie, and Traiger, 1976]. The six orderings may be placed into two categories. Each ordering considers the requests made of the globally accessible instance from the viewpoint of that instance. Consequently, the requests from one sequence of instructions may be interleaved with those from the other sequence of instructions.

5.1.1 Sequential Execution

i

```
x := [R Get]
[R Put f(x)]
y := [R Get]
[R Put g(y)]
```

Execution of the first sequence of instructions precedes the execution of the second sequence. The final state of the instance is dependent upon g(f(r)).

ii

```
y := [R Get]
[R Put g(y)]
x := [R Get]
[R Put f(x)]
```

Execution of the second sequence of instructions precedes the execution of the first sequence. The final state of the instance is dependent upon f(g(r)).

The execution of the two sequences of instructions has been serialised. One sequence is executed to completion before execution of the second sequence is started. This guarantees that the computing system stays in a consistent state; starting from a consistent state, each sequence of instructions leaves the computing system in a consistent state. Both these orderings are known as "serial schedules". Such schedules always leave the computing system in a consistent state.

5.1.2 Concurrent Execution

i

```
x := [R Get]
y := [R Get]
[R Put g(y)]
[R Put f(x)]
```

Execution of the second sequence of instructions is enclosed by the execution of the first sequence. The final state of the instance is dependent upon f(r). Any effect that the execution of the second sequence had on the instance has been lost.

ii

```
x := [R Get]
y := [R Get]
[R Put f(x)]
[R Put g(y)]
```

Execution of the second sequence of instructions overlaps the

execution of the first sequence. The final state of the instance is dependent upon g(r). Any effect that the execution of the first sequence of instructions had on the instance has been lost.

iii

```
y := [R Get]
x := [R Get]
[R Put f(x)]
[R Put g(y)]
```

Execution of the first sequence of instructions is enclosed by the execution of the second sequence. The final state of the instance is dependent upon g(r). Any effect that the execution of the first sequence of instructions had on the instance has been lost.

iv

```
y := [R Get]
x := [R Get]
[R Put g(y)]
[R Put f(x)]
```

Execution of the first sequence of instructions overlaps the execution of the second sequence. The final state of the instance is dependent upon f(r). Any effect that the execution of the second sequence of instructions had on the instance has been lost.

Concurrent execution of the two sequences has resulted in the loss of the effects of one of the executions of one of the sequences. This, in turn, has led to the computing system being in an inconsistent state. This has arisen because the two sequences of instructions interfere when they are executed concurrently. These orderings are known as "non-serial schedules".

A serial schedule will always cause the computing system to reach in a consistent state. Additionally, some non-serial schedules are said to be equivalent to serial schedules in as much as they also cause the computing system to reach a consistent state. However, the non-serial schedules given above are not equivalent to serial schedules, precisely because the two sequences of instructions interfere when they are executed and thus a consistent state is not reached.

5.2 INTERFERENCE AND INDEPENDENCE

To maintain the consistency of group of instances in a computing system, it is sufficient to ensure that the execution of sequences of instructions do not interfere. The necessary and sufficient conditions to guarantee that the state of computing system is consistent despite the concurrent execution of instructions are outlined by Schlageter [Schlageter, 1978]. In this section the concepts of interference and independence are explored with particular reference to distributed computing systems.

The requests sent to instances of objects during the execution of a sequence of instructions may be classified as 'update' requests and 'inspect' requests. An update request sent to an instance of an object causes the state of that instance to be transformed. In contrast, an inspect request has no effect on the state of the instance to which it is sent. Associated with every sequence of

instructions are two sets; the inspect set and the update set. The members of these sets are the names of those instances of objects to which requests are sent during the execution of the sequence of instructions. The members of the inspect set are the names of those instances which are recipients of inspect requests when the sequence of instructions is executed. The members of the update set are the names of those instances which are recipients of update requests when the sequence of instructions is executed. Since a request sent to one instance may result in a subsidiary request being sent from that instance to some other instance, and so on, an inspect request which propagates an update request is classified as an update request. To ensure that the two sets are disjoint, the names of any instance which is a recipient of both an inspect request and an update request is a member only of the update set.

Two sequences of instructions are independent if the update sets of both are disjoint, and the inspect set of one sequence does not contain names in the update set of the other sequence, and vice versa. If two sequences of instructions are not independent they are potentially interfering; concurrent execution of the two sequences may lead to the computing system being left in an inconsistent state. To ensure that the computing system is left in a consistent state it is sufficient to restrict the concurrent execution of sequences of instructions to those which are independent. Such sequences of instructions will always yield serial schedules. In the next section different methods of determining the inspect and update sets are

outlined.

Static analysis of the textual description of a sequence of instructions, perhaps performed by the programming language compiler, can be used to determine the update and inspect sets of that particular sequence of instructions. The sets will contain the names of all those instances of objects which could possibly be recipients of requests during the execution of the sequence of instructions. For example, a sequence of instructions which sends inspect requests to an instance of an object of type "T" would have the names of all those instances of that object in its inspect set. However, in all probability, only a small subset of those instances might actually receive requests when the sequence of instructions is executed. Clearly, using static analysis as a basis for determining whether any two sequences of instructions may be executed concurrently is unnecessarily restrictive. In the worst case, the inspect and update sets of each sequence of instructions may contain the name of every instance in the computing system. The sequences of instructions are potentially interfering and it will not be possible to execute them concurrently. It is more probable that each sequence of instructions will actually send requests to a small number of instances when it is executed; if the two sequences send requests to different groups of instances then it may be possible to execute them concurrently. Consequently, static analysis of the text of a program is a poor choice since it may eliminate much potential concurrency.

The inspect and update sets may also be determined dynamically as a sequence of instructions is executed. Initially the sets are both empty; as requests are sent to different instances of objects the sets are enlarged. At any time, the sets contain only the names of those instances which requests have been recipients of requests. This ensures that the degree of concurrency which can be exploited is maximised. However, as the inspect and update sets are enlarged during the execution of the sequence of instructions, a request may eventually be sent to an instance of an object which has already received a request during the execution of some other sequence of instructions. Consequently, the two sequences of instructions are now interfering with one another. To ensure that the state of the computing system remains consistent, this interference must be detected, and it may be necessary to take some appropriate action which will return the computing system to a consistent state.

Two different methods have been proposed in the literature to ensure that the execution of two or more sequences of instructions do not interfere. Both methods are described below.

5.2.1 Locks

To exclude other executions of sequences of instructions from interfering, a "lock" may be placed on an instance of an object during the execution of some sequence of instructions. Before an inspect or update request may be sent to an instance, it must first

be locked. This may be achieved by sending a lock request to the instance. Any requests sent to the locked instance during the execution of sequences of instructions other than that which requested the lock are invalid. This guarantees that the execution of two or more sequences of instructions cannot intefere with respect to a globally accessible instance of an object. Once a sequence of instructions has successfully made requests to the instance of an object, an unlock request should be sent to the instance. This then allows other sequences of instructions to send requests to that instance. The details of locks for distributed databases are presented in [Eswaran, et al., 1976]. The use of locks within a decentralised control flow computing system is outlined below.

Using the example of the two sequences of instructions given earlier, these could be rewritten to include explicit lock and unlock requests of the object addressed through the variable "R":

```
BEGIN                          BEGIN
[R Lock]                       [R Lock]
x := [R Get]                   y := [R Get]
[R Put f(x)]                   [R Put g(y)]
[R Unlock]                     [R Unlock]
END                            END
```

The action of locking an object excludes all other programs from sending requests to that object. The resulting schedules obtained from executing the two sequences of instructions are serial. Hence the computing system remains in a consistent state.

The position of the lock and unlock requests is crucial. One of the following strategies must be adopted if the consistency of the computing system is to be maintained:

- before the execution of a sequence of instructions is started, lock requests are sent to all instances of objects which are to receive requests during the execution of this sequence of instructions; an "unlock" request may be sent to any instance of an object provided that no further requests are to be sent to that instance;
- "lock" requests are sent in a predetermined order to the instances of objects which are to receive requests during the execution of the sequence of instructions; an "unlock request may be sent to an instance of an object provided that no further requests are to be sent to that instance;
- "lock" requests are sent in any order to the instances of objects which are to receive requests during the execution of the sequence of instructions; an "unlock" request may be sent to an instance of an object provided that no subsequent "lock" requests are sent to any instance of any object;

These strategies have different repercussions on the dynamic behaviour of the execution of a sequence of instructions.

The first strategy, which is also the easiest to implement, requires all the instances of objects to be locked in a single action before execution of the sequence of instructions is started. Once all the instances are locked, execution of the sequence of

instructions may begin. It corresponds to the static analysis described earlier in which the inspect and update sets of a particular sequence of instructions are determined in advance of the execution of that sequence. Locking the instances in advance may result in a large number of instances being locked unnecessarily. This arises because it is not always possible to determine in advance which particular instances are actually required. Consequently, this strategy may tend to restrict the degree of concurrency which can be exploited.

The second strategy, which is also relatively simple to implement, is based on a total ordering of all the instances of objects within the computing system. The instances of objects which are to receive requests during the execution of a sequence of instructions must be locked according to this ordering. Again, as with the first strategy, it may not be possible to determine in advance those instances of objects which will actually be required. Consequently, this strategy will also tend to limit the degree of concurrency which can be exploited.

The third strategy, which is also the most complex of the three, enables the number of instances of objects which have to be locked to be minimised. It corresponds to the dynamic analysis of the inspect and update sets of a particular sequence of instructions. However, to ensure that consistency is maintained, certain overheads are involved. Firstly, a lock request sent during the execution of one

sequence of instructions to an instance of an object which has already received a lock request on behalf of some other sequence of instructions causes the execution of the first sequence of instructions to be "rolled back". "Roll back" involves sending "unlock" requests to all instances of objects locked on behalf of the sequence of instructions, and also restoring those instances to the state which they had before they received the "lock" request during the execution of the sequence of instructions. The effect of "roll back" is to undo the work which has been achieved during the execution of the sequence of instructions. Secondly, an unlock request cannot be sent to an instance of an object until all the instances of the objects required during the execution of the sequence of instructions have been successfully obtained. This means that an instance of an object may be locked for the duration of the execution of the sequence of instructions, thereby enforcing a serial schedule.

There are drawbacks to the use of locks. First and foremost the degree of concurrency may be limited. As a program is executed the number of objects locked on its behalf grows during the first phase until no more objects are locked. During this time there will a decreasing number of programs which are independent of this program. This will depend upon the granularity of locking. As the objects are unlocked the degree of concurrency may increase. Secondly, construction of programs from other programs will lead to a hierarchy of lock-unlock requests. Programs lower in the hierarchy which

unlock should have the changes visible to the higher level programs but not to other programs outside the hierarchy.

As described here, the locks are exclusive. That is, once a lock request has been received by an instance of an object from a sequence of instructions, that instance is inaccessible to all other sequences of instructions. This exclusivity ensures that a serial schedule is obtained. However, it also restricts the degree of concurrency which can be exploited. It is possible for the exclusiveness of the locks to be relaxed so that several sequences of instructions may share an instance of an object. Consistency of the instances can still be guaranteed by application of certain constraints on the sharing permitted.

5.2.2 Timestamps

Associated with each subordinate computing system is a clock which generates globally unique timestamps. Each sequence of instructions is assigned a unique timestamp by the clock local to the processing unit executing the sequence. Every request made of an object is accompanied by the timestamp of the sequence making that request. The task of the object addressed is to satisfy the requests made of it in strict timestamp order. To do so will serialise the executions of the different sequences.

Each object will have to maintain a record of the last timestamp received and acted upon. Whenever a new request is received the timestamp of the request must be checked against that recorded by the object. If the timestamp presented with the request is less than that stored, the requests received with timestamps exceeding the current request become invalid, and the sequences of instructions which issued those requests must be rolled back and restarted. Conversely, if the timestamp presented exceeds that stored, the request is satisfied and the presented timestamp with that request is stored.

Applied strictly, the timestamp mechanism enforces a serial schedule. However, as with locks, there are certain drawbacks. It is possible for requests to be received in other than the timestamp order, and for these requests still to maintain the consistency of the instance of objects within the computing system.

5.3 OBJECT HISTORIES

In this section the concept of an object history is introduced, and various properties of object histories are discussed. An object history gives details of those computations which have occurred and which objects they have accessed. Each object has an object history associated with it. The history is an ordered sequence containing the details of the requests made of an object by different sequences of instructions.

Associated with each object is a table giving details of the requests made of that object. The ordering of the table reflects the order in which the requests were received by the object. Each entry in the table consists of the following components:

type: the type of request made (inspect or update);

requester: the identity of the sequence of instructions making the request;

state: the state of the object at the time the request was received.

The table may be used to construct a graph which represents the requests made by sequences of instructions to any given instance. Taking the tables of all the objects in a computing system allows a graph to be built which reflects the behaviour of all the sequences executed across the distributed computing system as a whole. Some constraints must be placed on the ordering of the entries in the tables. This is to ensure that inconsistent states are not reached; in particular, cycles may not exist in the graph. A cycle could be found in the graph if either of the following situations arise:

- an instance receives two requests from one sequence interleaved by a request from some other sequence;
- two or more instances each receive two requests from two or more sequences but in a different ordering.

The first situation can be avoided by ensuring that a cycle is not created in the history of an instance. The second situation is harder to avoid since it involves potentially constructing the graph for the whole computing system. This will require not only the histories of those instances to which a sequence has addressed

requests, but also the history of any instance to which other sequences have addressed requests if those same sequences have also addressed requests to the instances addressed by the original sequence.

Consider the situation where receipt of an update request by an instance X from the execution of the sequence of instructions P is denoted in the object history, HX, of the instance X by the value

```
MakeRequest(Update, P, S)   where S represents the state
                            of the instance X at the time
                            the update request was received.
```

Similary, receipt of an inspect request by an instance Y from the sequence of instructions Q is denoted in the object history, HY, of the instance Y by the value

```
MakeRequest(Inspect, Q, S)  where S represents the state
                            of the instance Y at the time
                            the update request was received.
```

The progression of time at an instance of an object is related to the sequence of inspect and update requests received by that instance.

Consider the two sequences of instructions P and Q given below:

```
P: BEGIN                    Q: BEGIN
   x := [R Get]                [R Put y]
   END                         END
```

Execution of the sequence of instructions P makes an inspect request to the instance addressed by the variable R, whilst execution of the sequence of instructions Q makes an update request of the instance addressed by the variable R. The object history for the instance addressed by the variable R will be one of the following:

HR = [MakeRequest(Inspect, P, S1), MakeRequest(Update, Q, S1)]

HR = [MakeRequest(Update, Q, S1), MakeRequest(Inspect, P, S2)]

The first sequence of entries denotes that the inspect request issued by the execution of the sequence of instructions P was received before the update request issued by execution of the sequence of instructions The second sequence of entries denotes that the update request issued by the execution of the sequence of instructions Q was received before the inpsect request issued by execution of the sequence of instructions P.

If a request issued as a result of the execution of some sequence of instructions P is received before a request issued during the execution of some other sequence of instructions Q by an instance X, this is denoted by P <X Q.

Consider the execution of the two sequences of instructions P and Q given below:

```
P: BEGIN                    Q: BEGIN
   x := [R Get]                y := [R Get]
   [R Put f(x)]                [R Put g(y)]
   END                         END
```

These two sequences of instructions are identical to those given in an earlier section. It was noted there that some orderings of the requests made during the execution of the two sequences of instructions led to the lost update problem. The object histories may be used to determine when an inconsistent state has been reached.

The execution of the two sequences of instructions P and Q are represented by the following object histories for the instance R.

- The execution of the sequence of instructions P precedes the execution of the sequence of instructions Q. The object history for the object R is:

```
HR = [MakeRequest(Inspect, P, S1),
      MakeRequest(Update, P, S1),
      MakeRequest(Inspect, Q, S2),
      MakeRequest(Update, Q, S2)]
```

  hence P <R Q.

- The execution of the sequence of instructions Q precedes the execution of the sequence of instructions P. The object history for the object R is:

```
HR = [MakeRequest(Inspect, Q, S1),
      MakeRequest(Update, Q, S1),
      MakeRequest(Inspect, P, S2),
      MakeRequest(Update, P, S2)]
```

  hence Q <R P.

- The execution of the two sequence of instructions P and Q are interleaved in some manner. The overall effect of the execution is that it appears that the execution of the sequence of instructions P had not occurred. There are two possible object histories for the object R which represent the two possible orderings of the inspect requests made by the two sequences of instructions:

  a.

```
HR = [MakeRequest(Inspect, P, S1),
      MakeRequest(Inspect, Q, S1),
      MakeRequest(Update, P, S1),
      MakeRequest(Update, Q, S2)]
```

The inspect request of the sequence of instructions P precedes the inspect request of the sequence of instructions Q. Consequently, the execution of the sequence of instructions Q overlaps the execution of the sequence of instructions P.

b.

```
HR = [MakeRequest(Inspect, Q, S1),
      MakeRequest(Inspect, P, S1),
      MakeRequest(Update, P, S1),
      MakeRequest(Update, Q, S2)]
```

The inspect request of the sequence of instructions Q precedes the inspect request of the sequence of instructions P. Conseqently, the execution of the sequence of instructions P is enclosed by the execution of the sequence of instructions Q.

- The execution of the two sequences of instructions P and Q are interleaved in some manner. The overall effect of the execution is that it appears that the execution of the sequence of instructions Q had not occurred. There are two possible object histories for the object R which represent the possible orderings of the inspect requests made by the sequences of instructions.

a.

```
HR = [MakeRequest(Inspect, Q, S1),
      MakeRequest(Inspect, P, S1),
      MakeRequest(Update, Q, S1),
      MakeRequest(Update, P, S2)]
```

The inspect request of the sequence of instructions Q precedes the inspect request of the sequence of instructions P. Consequently, the execution of the sequence of

instructions P overlaps the execution of the sequence of instructions Q.

b.

```
HR = [MakeRequest(Inspect, P, S1),
      MakeRequest(Inspect, Q, S1),
      MakeRequest(Update, Q, S1),
      MakeRequest(Update, P, S2)]
```

The inspect request of the sequence of instructions P precedes the inspect request of the sequence of instructions Q. Consequently, the execution of the sequence of instructions Q is enclosed by the execution of the sequence of instructions P.

In the latter two cases of the executions of the two sequences of instructions P and Q, the inconsistency of the instance of the object addressed by the variable R may be determined because the history of the instance cannot be ordered into the form P <R Q or Q <R P. Thus the existence of the lost update may be determined. In passing it is worth noting that a lost update is only problematic when the execution of the sequence of instructions which has been lost changes the state of some other instance such that this latter change is visible. Thus an inconsistency is introduced into the group of instances in a computing system.

The ordering of a group of inspect requests is strictly irrelevant since such requests do not alter the state of the instance addressed. Consequently, the two following pairs of object histories are

equivalent:

```
[MakeRequest(Inspect, P, S1),      [MakeRequest(Inspect, Q, S1),
 MakeRequest(Inspect, Q, S1),       MakeRequest(Inspect, P, S1),
 MakeRequest(Update, P, S1),        MakeRequest(Update, P, S1),
 MakeRequest(Update, Q, S2)]        MakeRequest(Update, Q, S2)]

[MakeRequest(Inspect, P, S1),      [MakeRequest(Inspect, Q, S1),
 MakeRequest(Inspect, Q, S1),       MakeRequest(Inspect, P, S1),
 MakeRequest(Update, Q, S1),        MakeRequest(Update, Q, S1),
 MakeRequest(Update, P, S2)]        MakeRequest(Update, P, S2)]
```

The example above dealt with the simple case of two sequences of instructions which updated a single shared instance. This is now generalised to the case of two sequences of instructions which update two shared instances.

Consider the two sequence of instructions P and Q given below:

```
P: BEGIN                        Q: BEGIN
   a := [R1 Get]                   x := [R1 Get]
   b := [R2 Get]                   y := [R2 Get]
   [R1 Put f(a, b)]                [R1 Put p(x, y)]
   [R2 Put g(a, b)]                [R2 Put q(x, y)]
   END                             END
```

Once again, these two sequences of instructions were considered in an earlier section. The execution of the sequence of instructions P makes inspect requests on the instances addressed by the variable R1 and R2 and makes update requests on both those same instances. The execution of the sequence of instructions Q makes inspect requests on the instances addressed by the variable R1 and R2 and makes update requests on both those same instances.

The various orderings of the executions of the two sequences of instructions are now considered from the viewpoint of the object histories of the instances addressed by the variables R1 and R2. There are four possible objects histories for each instance:

- P < Q

```
[ MakeRequest(Inspect, P, S1),
  MakeRequest(Update, P, S1),
  MakeRequest(Inspect, Q, S2)
  MakeRequest(Update, Q, S2) ]
```

- Q < P

```
[ MakeRequest(Inspect, Q, S1),
  MakeRequest(Update, Q, S1),
  MakeRequest(Inspect, P, S2)
  MakeRequest(Update, P, S2) ]
```

- P encloses Q

```
[ MakeRequest(Inspect, P, S1),
  MakeRequest(Inspect, Q, S1),
  MakeRequest(Update, Q, S2)
  MakeRequest(Update, P, S2) ]
```

- Q encloses P

```
[ MakeRequest(Inspect, Q, S1),
  MakeRequest(Inspect, P, S1),
  MakeRequest(Update, P, S2)
  MakeRequest(Update, Q, S2) ]
```

Only the first and second object histories are valid. The third and fourth object histories must always lead to an inconsistent state. However, since requests are being made to two independent instances, it is necessary to consider the ordering relation of both object histories. If the ordering relation of the two object histories is different, the group of instances in the computing system will reach an inconsistent state.

To generalise further, consider the following three sequences of instructions:

```
P: BEGIN          Q: BEGIN          R: BEGIN
   x := [A Get]      x := [B Get]      x := [C Get]
   [A Put f(x)]      [B Put f(x)]      [C Put f(x)]
   y := [B Get]      y := [C Get]      y := [A Get]
   [B Put g(x, y)]   [C Put g(x, y)]   [A Put g(x, y)]
   END               END               END
```

Once again, there are four possible object histories for each of the instances addressed by the variables A, B, and C. However, only two of the possible object histories yield an ordering on the sequences of instructions in the form p <R q. These orderings are, for the instances addressed by the variables A, B, and C respectively:

- P < R or R < P
- P < Q or Q < P
- Q < R or R < Q

The combination of these orderings may or may not result in the objects A, B, and C being in a consistent state. Of the eight possible combinations, six are valid:

- P < R and P < Q and Q < R => P < Q < R
- P < R and P < Q and R < Q => P < R < Q
- P < R and Q < P and Q < R => Q < P < R
- R < P and P < Q and R < Q => R < P < Q
- R < P and Q < P and Q < R => Q < R < P
- R < P and Q < P and R < Q => R < Q < P

The other two combinations are both invalid:

- P < R and Q < P and R < Q
- R < P and P < Q and Q < R

Both of these latter combinations imply that execution of one sequence of instructions has interfered with that of another sequence of instructions. For example, the combination P < R and Q < P and R < Q signifies that the requests mights have been made of the objects in the following order:

```
xP := [A Get]
[A Put f(xP)]

xR := [C Get]
[C Put f(xR)]
yR := [A Get]
[A Put g(xR, yR)]

xQ := [B Get]
[B Put f(xQ)]
yQ := [C Get]
[C Put g(xQ, yQ)]

yP := [B Get]
[B Put g(xP, yP)]
```

The inconsistency here may arise because the value used in the Put request made of the object B during the execution of the sequence of instructions P depends upon the value preserved in the variable x which is local to that sequence of instructions. This local variable contains a value received from the object A which has since received a Put request during the execution of the sequence of instructions R.

The interference outlined in the previous two examples cannot be detected simply by examination of the object histories of the objects to which requests have been made during the execution of any one of the sequences of instructions. To detect this interference it is necessary to consider not only the object histories of all the objects to which requests have been made during the execution of a

sequence of instructions but also the object histories of those objects which other interfering sequences of instructions have addressed. From all these object histories it is possible to construct a graph representing the ordering of the group of executions of the sequences of instructions. If the graph contains any cycles, then an inconsistent state will be reached by the group of objects considered. On detection of the inconsistency, the sequence of instructions must be rolled back to a point at which no cycles exist in the graph. At this point the group of objects will be in a consistent state. In the example given above, all three sequences of instructions must be rolled back, thereby undoing any useful work done.

The disadvantage of this strategy lies in the amount of information which is required to determine if some sequence of requests has resulted in a group of objects remaining in a consistent state. Furthermore, the information in the object histories about the activities of a particular sequence of instructions may need to be preserved beyond the lifetime of that sequence of instructions. The information about the sequence of instructions may only be discarded when it is known that the sequence of instructions will not need to be rolled back.

Until now the nature of the instances to which requests are made has not been considered. It has been implicitly assumed that these instances do not themselves make requests of other instances. In the

decentralised control flow model of computation an instance may make requests of any other instance. A request made of one instance may result in a chain of requests emanating from that instance to other instances. This has an important consequence on the use of the object histories. Instances which provide services to other instances will probably have a long life-time. It may be necessary to retain all the information about the requests made by instances objects so that future inconsistencies may be resolved.

To constrain the amount of information whixh must be represented by the object histories, the concept of "transaction" is introduced. A transaction is any sequence of instructions with the following three properties:

i. execution of a transaction which starts with the computing system in a consistent state will always leave the computing system in a consistent state;

ii. during the execution of a transaction inconsistent states may arise; such state should be invisible to other transactions;

iii. a transaction is executed completely, or not at all.

The changes which have been made to the computing system as a result of a transaction become permanent on completion of the transaction. Transactions may be nested; an enclosed transaction and the enclosing transaction are not subject to the same concurrency control that exists between independent transactions. In the object histories, the identification of the sequence of instructions making

a request is replaced by the identification of the hierarchical transaction structure.

At least one instance of the special transaction object is required in the computing system. An instance of this object yields, on request, a unique transaction identifier and is responsible, on notification that a transaction has terminated, for ensuring that a consistent state has been reached.

5.4 CONCLUDING REMARKS

The approach to concurrency control, described as object histories, takes much of the burden away from the user. It is the user's responsibility to identify the different transactions within a sequence of instructions, but the underlying run time system is solely responsible for ensuring that the computing system remains in a consistent state.

Locks maintain consistency at the expense of parallelism. Time-stamps require the existence of a global clock to restrict the requests received by the instances of objects in a computing system to a serial schedule. The aim behind object histories is to maximise the parallelism which can be exploited in a distributed computing system. The obvious disadvantages to this approach are the need to perfrom roll-back on detection of an inconsistent state and the amount of information which must be passed between transactions and

instances of objects to maintain the object histories.

6 PROGRAMMING LANGUAGES FOR THE TWO NEW DESIGNS

Two new designs of architecture of computing system have been outlined in chapter four, together with descriptions of the models of computation underlying those designs. In this chapter two different programming languages are described, one for each of the new designs. Neither programming language is presented as the definitive programming language for distributed computing systems but rather as a vehicle for further research.

Many models of computation do not permit a computing system to be described as a single entity. There is a sharp distinction between the activity of a program written in a particular programming language and the steps required to enable that program to be executed on a computing system. The majority of contemporary computing systems require a layer of specialised software known as the control program. The complexity of this software varies greatly between one computing system and another. However, the role of the control program is essentially the same; it provides an interface between the underlying components of the computing system and the programs which may be executed on that computing system. For example, most control programs organise the storage media which may be attached to the input/output devices into some form of structure.

The control program also supports a language of its own; the job control language [Flores, 1971]. This language has semantics which

usually differ from those of the programming languages used to write software. The job control language is used to control the behaviour of a program as it is executed. Consequently, a program or group of programs is executed through the agency of the control program.

Both the programming languages presented in this chapter represent an attempt to break down the distinction between the job control language and the programming language. Two areas of research have influenced the proposals described here to integrate the programming language and the job control language. The synthesis of a programming language and a job control language was first described in [Stoy and Strachey, 1972]. The programming language BCPL was used to implement a small control program for a minicomputing system. This programming language was also used as both the job control language in which users controlled the behaviour of the computing system, and the programming language in which users wrote applications programs. This enabled the programmers to use the facilities provided by the control program directly from within their programs. Furthermore, a user's program could invoke other programs as routines.

One of the rigid distinctions evident between a programming language and a job control language is the different treatment given to the storage capabilites of a computing system. Three distinct levels of storage may be found within most contemporary computing systems:

- internal registers of the processing unit;
- individual cells of the memory component;
- storage devices attached to the input/output devices.

The internal registers of the processing unit are used during the execution of a program to preserve values between the execution of a few instructions. These registers are implemented using very fast logic circuitry and, typically, the processing unit has only a small number of such registers. For example, the Motorola M68000 has one set of eight general purpose data registers and one set of eight special purpose address registers.

The memory component of a computing system is used to hold the instructions of programs and their data as they are executed by the processing unit. The memory components of contemporary computing systems range in capacity from thousands of cells to millions of cells. In many cases, the actual memory component of the computing system is insufficient to hold all the information representing the different programs currently being executed by the processing unit. One of the tasks of the control program may be to move the information held within the memory component back and forth from the input/output devices so that each program which is executed has sufficient cells of the memory component.

The input/output devices are used for longer term storage of programs and data. An individual storage medium such as a disc pack may hold some millions of characters. Typically the storage media

are removable, thus permitting an infinite amount of information to be accessed through the input/output devices.

The flow of information between the internal registers and the memory component is performed under the control of the programming language. This ensures that the distinction between the two is not visible to the users of the programming language. The flow of information between the memory component and the input/output devices is under the control of the control program at the request of the programming language. To access a particular item held on an input/output device, the programmer must explicitly request that data to be transferred. The distinction between the input/output devices and the memory component is visible; an item stored on the input/output devices may only be used within a program when it has been transferred from the input/output devices to the memory component. A further distinction is often made between the information held within the memory component and that held within the input/output devices. The former represents the code and data of programs which may be active and in the process of being executed. The latter represents the data and perhaps code of programs which are inactive and cannot be executed. However, since the control program is responsible for the movement of information back and forth between the input/output devices and the memory component, this distinction may be less distinct.

A program is used to model some world of interest to the user. If the program is executed under the control of a batch processing control program, input values are given to the program before execution is started and output values become available once the program has been executed. All the input values must be available before the program can be executed. The program may be thought of as a function from input values to output values. It is often desirable to use the output values of one program as the input values of some other program. In a batch processing environment the output values from one program must be saved on some long-term storage medium so that they can be used subsequently as input values to the other program. Typically the values are stored on a magnetic storage medium such as a disc or tape; such information may be stored for an indefinite period of time. The interval between the completion of execution of the first program and the start of execution of the second program can be of any length. It would not be possible for the information stored on the magnetic storage medium to be 'lost'.

The output values stored on the magnetic storage medium may be used as input values to several programs. There is no need to execute the program which generated the original output values more than once as the storage medium is used to preserve the output values for later use. The concept of preserving values for later use is common to many models of computation. Within a program a variable may be assigned a value which has been obtained by evaluating an arbitrarily complex expression. It would be possible to re-evaluate

the expression each time the value it represented was required. However, it is sufficient to retrieve the value assigned to the variable, rather than re-evaluate the expression.

In the referentially transparent models of computation a variable can never have some other value assigned to it. Therefore, to preserve a succession of output values a new variable is required for each of the values produced. Whilst this requirement makes symbolic manipulation of the program text feasible, it may lead to wasted storage space. Every variable to which no further reference will be made occupies unuseable storage space. In many implementations of referentially transparent programming languages a garbage collector is provided to reclaim such storage space. Other techniques exist to prevent the unneccessary creation of the wasted storage space in the first place. In the von Neumann model of computation the succession of output values can be represented by the re-assignment of values to existing variables. This allows the space occupied by a value which will not be accessed again to be re-used; this form of assignment represents an optimisation of the use of the memory component. It depends upon the ability of the user to achieve a correct partial ordering of the statements of the program, such that a variable is not re-used until all the statements dependent upon the initial value assigned to the variable have been executed.

A variable within a program allows some value which has been calculated to be preserved for use at a later occasion. These

variables disappear once the program in which they were created has been executed. In order to preserve values between executions of programs, variables must also exist outside of the program but within the control program. One of the tasks of the control program of a computing system is to organise the storage of information within the computing system for differing periods of time.

With the increasing sophistication of control program languages it is quite often possible to use them to write programs. The distinction between writing programs and writing job control scripts to control the behaviour of those programs has become blurred. Indeed, recent developments have made it even more difficult to discern the difference between the two activities. For example, the REXX language [Cowlishaw, 1984] has been designed specifically with both activities in mind. It can be used to write conventional programs which are compiled into executable machine code, or job control scripts may be written to control the execution of programs.

Programming languages such as Pascal and Algol 68 provide a rich set of data and control structures in which a problem can be represented. In contrast, many job control languages provide only rudimentary structures. The control structures are limited to conditional statements and explicit transfers of control. The data structures are limited to single memory cells and various organisations of file such as sequential, indexed sequential and random. Whilst there has been a trend towards the formal definition

of programming languages there has been no such development in job control languages. This is of no consequence for many job control languages as they are not sufficiently powerful to be used as programming languages. However, as the level of sophistication increases it is likely to become a problem. For example, the different UNIX shells support languages which provide many of the control structures found in conventional programming languages. Programs written in programming languages such as C may be called as routines from control programs. It is possible to write quite complex programs in these languages.

As noted earlier, the only data structuring capablities provided in many control program environments are those related to the storage of information on the input/output devices. The methods of organising the information are usually classified by the access patterns which each supports. For example, the records stored within a file organised sequentially may only be accessed sequentially. The information stored within a file has no intrinsic type associated with it. Depending upon the underlying structure of the file it may be a sequence of blocks or a sequence of lines. It is the task of the users of the information to ensure that the information is accessed in a sensible manner. The procedures to access the information operate on the raw information stored on the device.

A read request causes some physical unit of information to be transferred and it is the responsibility of the user to transform

this physical unit into a logical unit. For example, the read request may transfer a block of 512 characters from the input/output device to the memory component. If the information represented is a sequence of text lines, then it is the responsibility of the users to determine the beginning and end of the successive lines. This low level interpretation is no different from the simple requests which may be made by the processing unit of the memory component. Single words of information are transferred between the memory component and the processing unit. However, most programs require a higher level interpretation of the information held within the memory component, so that a sequence of adjacent memory cells in interpreted as an array or a record.

The conflict between the different mechanisms for the storage of information for differing periods of time is described in [Atkinson, Chisholm, and Cockshott, 1981]. A programming language usually provides a set of constructors for the representation of data structures within a program. Data structures which are to exist beyond the lifetime of a program must be transmitted to the control program for storage on some storage medium. Typically, the constructors for the representation of data structures within a program do not match those supported by the control program. Some programming languages have been extended to provide additional constructors which match those supported by the control program. The alternative approach, and that taken in the development of the programming language PS-Algol, supports the constructors of the

programming language on the storage media provided by the control program.

The seemingly artificial distinction between the programming language and the job control language stems partly from the distinction between programs which are active and data which is passive. Neither of the programming languages presented in this chapter supports this divided view. In both programming languages the distinction between short term storage in the memory component and long term storage on some storage medium is deliberately blurred.

Within many distributed computing systems, the basic unit which may be shared between the component computing systems is the file. A file contains a set of information with which little or no type information can be associated. To use the contents of the file the access patterns appropriate to that file must already be known as must the manner in which the individual items of the file are to be interpreted. It is quite likely that access routines and routines to interpret the information in the file will be duplicated both at the sending computing system and at the receiving computing system. Moreover, it is not possible for the owner of the information to control rigidly how the information is used.

The programming language for recursive control flow computing systems treats both short term storage and long term storage equivalently. This has been achieved by considering the filestore of

the computing system as an extension to the memory component of the computing system. A single hierarchy of storage is thus presented to the programmer. Names beneath a certain point in the hierarchy identify the cells within the memory component which exist only during the execution of a program. Names above that point identify the various files in the filestore which exist beyond the execution of individual programs.

In contrast, within the programming language for decentralised control flow computing systems objects are used to represent data structures both during the execution of a program and outside the execution of programs. Transfer of objects between short term storage and long storage is the responsibility of the underlying software interface between the components of the computing system and the program. Instances of objects which are passive may be held on long term storage. When the instance of an object becomes active, as when it receives a request from some other instance of an object, the instance is transferred to short term storage so that the control structures associated with the instance may be executed.

6.1 RELATED WORK

Several programming languages have been designed to address the issues outlined in this and earlier chapters, such as the expression of concurrency, the maintenance of consistency, and the persistence of data. In this section some of these programming languages are reviewed. A wider survey of the different programming languages designed for use in concurrent computing systems has been presented in [Stotts, 1982].

6.1.1 Pascal-m

The extensions to the programming language Pascal which have resulted in Pascal-m [Abramsky and Bornat, 1983] include the addition of mailboxes, processes, and modules. The programming language is designed for the construction of programs for networks of loosely coupled computing systems.

A mailbox represents a named channel through which messages may be transmitted between processes. When a mailbox is declared within a process or module, it is given a type by the programmer. This restricts the messages which may be transmitted through the mailbox to being values of the defined type. Consequently, run time type checking of the messages sent through mailboxes is unnecessary. However, strictly applied compile time type checking restricts the usefulness of the mailboxes as will be outlined later. Mailbox

identifiers may themselves be transmitted within a message; this allows the connection topology of a given set of processes to be changed dynamically. However, reference or pointer values may not be transmitted within messages. This ensures that the local variables of one process cannot be manipulated directly by some other process.

Each process has associated with it a thread of control which allows the execution of the processes within a module to proceed concurrently. It is not possible for the programmer to control this concurrency directly. Any number of processes may send messages through a given mailbox and, correspondingly, the messages sent through a mailbox may be received by any number of processes. A process sending a message through a mailbox is delayed until there is at least one process to receive a message from that mailbox. Indeed, the transmission of messages through mailboxes is synchronous.

Modules are introduced as a structuring tool to group together the definitions of a set of processes and to declare the mailboxes which are required for communication. The mailboxes may be associated with a particular process either statically, by being explicitly named within the textual description of the process, or dynamically, by being passed as a parameter to the process. A process is rather like a conventional Pascal program; whereas the latter communicates with its environment through the use of external files, the former does so through the use of mailboxes. However, processes which communicate with one another are all declared within a single module. A useful

development of the programming language would be to allow processes to exist outside a module definition. This would permit the construction of general purpose processes.

The strict compile time type checking of mailboxes makes it impossible to write general purpose processes which can receive messages of any type. This problem is particularly acute when attempting to write a process which provides a general service to users. At the time the process is written, the variety of types which may appear within the messages it receives from user processes is, in general, unknown. Some form of mechanism is required to overcome the strict compile time type checking. This is provided in the Pascal-m programming language by the special type "ANY" which may be used to encapsulate any type and allows it to be passed within a message. The type which is encapsulated may only be decomposed by the process which originally encapsulated it.

6.1.2 Argus

The preservation of the consistency of data in the presence of concurrency and hardware failures is addressed by the proposals in the Argus programming language [Weihl and Liskov, 1983]. Traditionally, the problem of consistency has only been tackled in database management systems or in file systems. The solution adopted in this programming language is based on abstract data types with the additional properties of atomicity and resilience. It is motivated

by the desire to write programs for distributed computing systems where these issues will undoubtedly arise.

A basic structuring tool within the programming language is the "guardian". A guardian consists of a collection of data objects and processes. Associated with each guardian is a thread of control which allows execution of the different guardians to proceed concurrently. The processes within a guardian may manipulate the collection of data objects local to that guardian. Information may be transmitted between different guardians through the use of the processes of the particular guardian. There is no globally accessible memory component.

Certain atomic types are built into the programming language. The operations of these types are classified as "readers" or "writers" and concurrent access to data objects of the atomic types are excluded in the expected manner; several "reader" operations may proceed concurrently on an object of an atomic type, but any "writer" operation must have sole control of the object. Locking is provided to allow the user to obtain serial schedules when appropriate.

It has been recognised that these built-in atomic types limit the degree of concurrency which may be exploited. To increase the degree of concurrency, user defined atomic types are supported. The type "mutex[t]" defines an atomic type, based on the existing type "t", which provides mutual exclusion to the accesses made to values of

this type. The operation "seize", when applied to an object of such an atomic type, gains sole possession of that object. A further operation, "pause", causes the object gained to be released for a system dependent period of time before it is regained. It is guaranteed that the object will be regained before execution resumes.

The use of the operation "pause" allows a programmer to implement conditional critical sections. The operation "seize" is used to gain control of some object; assuming that the condition has been fulfilled, the process continues with sole control of the "seized" object. However, if the condition has not been fulfilled, the process may use the operation "pause" to release the control of the "seized" object, and be made to wait for some system dependent period of time before it regains control of the object; the process may then check the condition once again.

Two potential problems are not directly addressed. Firstly, it may be possible that some other process destroys the "seized" object during the period which some other process has "paused". Secondly, nesting of the use of the operation "seize" may lead to deadlock; the effect of the operation "seize" is the same as that of "lock" discussed in chapter five.

Atomic variants are also introduced into the programming language. These are not dissimilar to variant records in the programming language Pascal. An atomic variant is an atomic type. An object of

an atomic variant may be in one of a number of states. Each state is identified by a particular tag of the atomic variant. Certain operations are provided to change the state of an object of an atomic variant type. Such changes are, however, subject to confirmation. If the process which requested a change to the state of an object of an atomic variant type aborts, the state of that object is changed back to that it had before the process requested the change. Once again, the potential problem of the effect that restoration of a previous state of an object might have on processes which are dependent upon the current state of that object is not addressed.

No explicit operations are provided to signify that a process has aborted or committed the changes it has made to the different data objects. Rather, the run time support environment of the programming language must initiate the appropriate action when a process terminates.

6.1.3 Distributed Path Pascal

Distributed Path Pascal [Campbell, 1983] is an extension to the "P4" version of the programming language Pascal. The extension permits data encapsulation, open path expressions, and process structures.

Data encapsulation is implemented by the introduction of objects as an additional type constructor to the programming language. An

object may contain the declarations of variables which are local to each instantiation of the object. Structured types declared within the object may be exported to the environment surrounding the description of the object. Routines may be declared within the object to provide operations which manipulate the variables which are local to each instantiation of the object. Some of the routines thus declared may be classified as interface routines; these are accessible from outside the instantiation of an object. Initialisation code may be used within the description of an object. Whenever an instance of an object is created, the initialisation code is executed.

Since objects have been introduced as type constructors, an object value may appear wherever a value of a structured type, for example, record or array, may appear. It is also possible to transmit reference values of types between different objects. This allows one object to pass a reference value which denotes a reference to a variable within the local space of the instantiation to a second object. Consequently, this second object may manipulate the variable indirectly.

Open path expressions are used to specify the synchronisation constraints for a possibly concurrent set of executions. The description of an object contains a path expression which specifies the permitted orders of sequential and concurrent execution of the interface routines of the object. Three kinds of constraint may be

specified:

i. "A ; B" specifies that the execution of A must have terminated before execution of B may commence;

ii. "n: (C)" specifies that there may be at most n concurrent executions of C;

iii. "[C]" specifies that the concurrent executions of C are unrestricted.

The constraints may be combined to yield arbitrarily complex path expressions. For example, "1: (get); 1: (put)" specifies that there may be at most one execution of the routine "get" and one execution of the routine "put". Furthermore, the routines may not be executed concurrently.

The combination of objects and path expressions allows the necessary restrictions on the concurrent access to the objects to be specified which will enable correct synchronisation of requests to a common object. However, this synchronisation will only give the weak consistency outlined in chapter five.

To permit the expression of concurrency within a program, processes have also been introduced into the programming language. The process is a structuring unit which has its own independent thread of control. Surprisingly, information may be transmitted between processes through the use of a globally accessible memory component and not solely through the use of the interface routines.

The two structuring techniques introduced into the programming language are distinct. Processes are long-lived with their own thread of control. Information is transmitted between processes using shared memory. In comparison, objects are totally passive. Control may be passed to an object as the result of a call to an interface routine, but the thread of control is always passed back to the process which invoked the routine. Whilst the path expressions associated with each object will guarantee consistency in the weak sense, the introduction of processes into the programming language makes the problem of maintaining the consistency of the objects that much harder.

To cater for distributed computing systems, the concept of object has been broadened to include remote objects. A remote object has the same semantics as an object. Indeed, the remote procedure call semantics are used to hide the message passing which presumably takes place on the underlying communications medium when an interface routine of a remote object is invoked. Similarly, a remote object is a passive entity.

Each remote object has an address allocated to it by a system administrator. This address is unique in the whole network of computing systems which comprises the distributed computing system. Association of a particular remote object with a process may occur statically, by reference to the unique address within the textual description of the process, or dynamically, by invocation of the

operation "import" which returns a reference to the named object. In either case, the address of the remote object may be specified explicitly or implicitly. The naming scheme chosen gives a flat view of the objects within the computing system.

6.1.4 Occam

The programming language Occam [May and Taylor, 1983] has been designed for writing software for computing systems consisting of large numbers of interconnected subordinate computing systems. Typically, these subordinate computing systems will be Transputers. The aim of the design of the programming language is to be simple. It is based on concepts of concurrency and communication first proposed in the programming language CSP [Hoare, 1985]. Additionally, the programming language is also claimed to be its own formal semantics, and activities such as program transformation are possible.

The basic unit of structure supported within the programming language is that of the process. A process is an active entity which has associated with it a thread of control. Each process has a set of variables which are local to the instance of the process. These variables may be manipulated only by the routines declared locally to the process.

Communication may take place between processes using named channels. Each channel may transmit a single "word" value from one process to another process. The channels declared within a program have no type information associated. Consequently, the messages transmitted through a channel are not type checked. It is not possible to have more than one process transmitting information along a channel, or more than one process receiving information from a channel. The channels must be named explicitly in the textual description of the program and so it is not possible to alter the toplogy of the interconnection between the processes dynamically. Indeed, it is not possible to create processes dynamically.

Concurrency may be exploited by the programmer at an extremely low level. The constructor "PAR" is used to denote that two or more statements may be executed concurrently. The memory locations used by such statements must be distinct. However, the wider aspect of consistency is not addressed.

6.1.5 PS-Algol

An interesting extension to an Algol-like programming language, S-Algol, has attempted to bridge the gap between programming languages on the one hand and control programs on the other hand. The programming language PS-Algol [Atkinson, Bailey, Chisholm, Cockshott, and Morrison, 1983] has extensions which enable the data objects manipulated within a program to outlive the lifetime of that

program. The mechanisms to achieve this are those already built into in the programming language S-Algol.

The persistence of a data object is independent of the way in which that data object is manipulated by a program. Furthermore, the expression of what a program does is independent of the persistence of the data manipulated by that program. With conventional programming languages, the type information of long-lived data objects is "lost" between different executions of programs when those data objects are stored on long term storage media. Typically, the data objects must be mapped onto one of the different variety of file organisations which are supported by the control program of the computing system. This mapping is the responsibility of the user of the data objects and it is often necessary to write substantial sections of program to achieve it.

The persistent data objects are represented in a database which is held on the file system provided by the control program. Maintenance of this database is the responsibility of the run time support environment of the PS-Algol programming language. A database is organised as a table containing entries which denote the persistent data objects. Each entry of the table is a pair of values consisting of a key by which the data object may be accessed and a pointer to the data object itself. Several programs may have the same database open for reading at the same time, but only one program may have the database open for writing at any one time. The same database may be

opened and closed several times during the execution of a program. However, no other program may make changes to the database during the intervening period when the first program has closed the database and has not re-opened it.

The persistent data objects are those data objects which can be reached from the table. A scanning strategy similar to that found in traditional "mark scan" garbage collectors is used to identify such data objects. The changes made to the database must be committed explicitly using the operation "commit", otherwise all the changes since the last use of the operation "commit" are lost.

Concurrency may not be exploited by the programmer within a single program. However, the execution of several programs may be concurrent. Data objects cannot be shared between the concurrent execution of these programs since the restriction on the access to the database containing the data objects will result in a serial schedule of the programs.

6.1.6 Analysis

None of the programming languages outlined above comes sufficiently close to the requirements of a programming language which is designed for the construction of software systems for the general purpose distributed computing systems described in chapter four. These requirements are the following:

- the instances of objects created as a result of the execution of a sequence of instructions may exist after execution of that sequence has terminated;
- the relationship between the execution of two sequences of instructions may be that of client and server;
- processes may be executed concurrently;
- the instances of objects in a computing system must be maintained in a consistent state.

These following mechanisms are required within the programming language if the requirements enumerated above are to be supported:

- persistence of data;
- communication between different sequences of instructions;
- concurrency;
- concurrency control.

The following table shows how the different programming languages outlined above rate on each requirement:

	Persistence of Data	Communication	Concurrency	Concurrency Control
Pascal-m	No	Yes	Yes	No
Distributed Path Pascal	No	Yes	Yes	Yes
Argus	No	Yes	Yes	Yes
Occam	No	Yes	Yes	No
PS-Algol	Yes	No	No	Yes

Different mechanisms to support communication between sequences of instructions and concurrency have been designed. These mechanisms

are outlined below.

Two different approaches to communication between sequences of instructions have been taken by the designers of the programming languages surveyed above. Since the computing system for which these programming languages are designed are geographically distributed, the transmission of information between the different subordinate computing systems of the distributed computing system takes place over some communications medium. Typically, these transmissions will make use of the underlying primitive operations "send" and "receive" by which messages are transmitted between the subordinate computing systems.

Programming languages such as Pascal-m and Occam permit the programmer to use these primitive operations. The use of messages allows the representation of a very generalised flow of data between the different processes in a particular computing system. In particular it supports communication between processes which is directed through other processes. For example, a client process may require some service from a server process. Rather than communicate directly with the server process, the client process may have to communicate with some "directory" process which routes the messages to a particular server process. All communication between the client process and the server process may take place throught the agency of this directory process. Furthermore, the server process may not respond immediately to the messages received from the client process;

they may queued awaiting some convenient time when the server process becomes free to process them.

In contrast, the primitive operations of the underlying communications medium may be hidden from the programmer. The semantics of the procedure call has been used to provide a higher level inter-process communication mechanism - the remote procedure call. A client process makes a request of a server process as if it was issuing a procedure call. The client process is made inactive, and control is passed from the client process to the server process. The server process becomes active and responds to the request. It then becomes inactive and control is returned to the client process which is made active again. Data may flow between the client process and the server process as control is passed between the two. The procedure call is good at representing a direct relationship between the client process and the server process where the server process responds immediately to the request from the client process.

The use of messages appears to reflect the freedom presented by the underlying communications medium. A process which transmits a message to some other process does not need to receive a reply from that process. Furthermore, messages may be received at any point during the execution of a process simply by use of the operation "receive". In contrast, the semantics of the procedure call restricts this freedom. Every process which receives a request must make a response to that request at some time in the future.

Furthermore, since the receipt of a request by a process causes a procedure call to be made to some routine to handle that request, the requests may only sensibly occur at the top-most level of the process. Thus, the body of a process is implicitly a loop which continually receives messages and discriminates between the different requests, causing the relevant handling procedure to be invoked. The differences and similarities between procedure calls and messages have been discussed in [Stankovic, 1982]. In particular, the effect that the different mechanisms have on the communication patterns between sequences of instructions is outlined, and the relevance of the mechanisms to the exploitation of parallelism is shown.

The programming language Occam is the only one of the five surveyed to allow the programmer to express concurrency directly. The other four programming languages all support concurrency at a coarser level which cannot be controlled directly by the programmer. In the case of the programming language PS-Algol, this concurrency is so coarse as to be of little or no interest here. Of the remaining three programming languages, different processes may be executed concurrently in each. Giving the programmer the ability to express concurrency directly does not necessarily yield an efficient program. From the estimates of the execution speed for the various constructs given in [INMOS, 1984], an analysis of a program text will determine when concurrency may actually yield a program which is faster than its strictly sequential counterpart. Additionally, for concurrency to be generally useful, either the semantics of the programming

language must be restricted, or constructs to control the concurrency must be introduced to ensure that the concurrency does not lead to the kind of problems described in chapter five. For example, statements which make use of shared variables are not suitable candidates for concurrent execution.

The concurrency control mechanisms required of a distributed computing system must ensure that all objects within the computing sytem are maintained in a consistent state at all times. In particular, the issues discussed in chapter five must be considered. The programming languages Argus and Distributed Path Pascal both have concurrency control mechanisms which will only maintain the consistency of the objects within a computing system in the weak sense. As was shown in chapter five, such consistency is not sufficient. Of the remaining three programming languages surveyed, neither Occam nor Pascal-m address this particular issue. The concurrency control mechanisms required must be implemented by the programmer. The concurrency control mechanism for the programming language PS-Algol provides a satisfactory solution, but as was noted earlier, little or no concurrency may exploited within programs written in that programming language.

6.2 THE BASIX PROGRAMMING LANGUAGE

The BASIX programming language [Gouveia Lima, et al., 1983] was a first attempt at an implementation of a high level version of the recursive control flow model of computation and attempts to unify concepts of programming languages (e.g. Basic, Lisp) with those normally associated with control programs (e.g. Unix shell). Within the BASIX programming language there is a single notion of object which serves the role of variables, lists, messages, programs, files, and directories. There are a number of long term goals for the BASIX programming language:

- the programming language should provide a complete programming environment such as that found with an object-oriented programming language;
- the programming language should have control mechanisms for the management of concurrent processes.

6.2.1 Overview

The syntax of the Basix programming language is presented more fully in [Gouveia Lima, 1984]. In what follows, a short description of the programming language is given. In [Gouveia Lima, 1984] the semantics of the programming language have been presented very informally through the use of example programs. In the subsequent section the attempt to present the semantics more formally is described.

All users of the BASIX programming language on a particular computing system share the same information structure and interact with this structure via the processes which they activate. At any time a user has access to a particular object within the information structure. The user may select which object is required by giving its name. New information entered by the user changes the contents of the object, but does not cause execution. Commands may be issued by the user. These are executed and may cause changes to the information structure of the computing system. Such changes are visible to all users of the computing system.

Information is represented as a single nested structure which merges the concepts, found on contemporary computing systems, of directory, file, array, variable, message, and program. Each is a named object whose specific semantics are defined by which of the five system-wide operators (LOAD, STORE, TAKE, PUT, EXECUTE) is performed on the object. A named object (i.e. the contents of a memory cell) may be accessed as a "variable", as a "message", or as a "program". These are distinguished in the language in the following ways:

Semantics	Operation	Usage of Name
variables	LOAD	... name ...
	STORE	name := ...
message	TAKE	... name[] ...
	PUT	name[] := ...
program	EXECUTE	name object ...
	EXECUTE	name(...)

A name consists of one or more selectors "{/}selector{/selector}" defining a path to the target object. Selectors are interpreted from left to right, each selector moving the remainder of the name to an adjacent context. A selector may be:

- an alphanumeric character string;
- a numeric character string;
- a bracketed object whose execution yields the selector;
- a character defining one of the four accessible contexts:

Context	Character	Explanation
local	.	local objects of a program
parameters	$	parameters of a called program
non-local	..	non-local objects of a program
current	/	current context i.e. the directory of the program; this character may optionally occur at the start of a name.

For example "$" is used to access standard input "$/I", and standard output "$/O", and the parameters "$/1", "$/2" ... of a process.

Any program consists of a list of commands separated by control symbols, this being represented as "command { control command } ...". The control symbols define the order of execution of the two adjacent commands, which may be sequential ";", pipelined "|" or parallel "&". They also define how the standard inputs and standard outputs of the commands are connected together. BASIX accepts commands of the form:

name : object

object

The first command is a declaration used to create and label an object relative to the local context. Both the "name" and the object are evaluated before the assignment. The second command is then immediately executed and either returns some value to the user or makes some change to the information structure. An executable object is a list of objects separated by blanks where "blank" may be a sequence of spaces, a comma, or a newline. The leftmost object of the list defines the task to be performed. There are three types of executable objects:

	BASIX Format	Example
procedure call	object { object object } ...	sort in out
statement	keyword { object } ...	if a < b ...
expression	object { operator object } ...	c + d

Keyword commands define conditional "if", repetitive "do", and replicative "for" execution etc. Conditional and repetitive commands centre on the conditional "object -> object" which specifies that the second object is only executed if the result of the first object is "true". The notation for the conditional is taken from [Dijkstra, 1976]. The command "if ... fi" consists of a list of commands which execute in turn until a conditional is "true". This command may be used in the following ways:

Traditional Construct	BASIX Format
IF ... THEN ...	if object -> object fi
IF ... THEN ... ELSE ...	if object -> object; object fi
IF ... THEN ... ELIF ... THEN ...	if object -> object; object -> object;

```
ELSE ...                    object          fi
```

The command "do ... od" consists of a list of commands which execute repeatedly until a conditional is false. The statement may be used in the following ways:

Traditional Construct	BASIX Format
WHILE ... DO ...	do object -> object od
REPEAT ... UNTIL ...	object; do object -> object od

The command "for ... rof" has the following format:

```
for alphanumeric = object do object rof
```

This "for" command evaluates the first "object" and then replicates the second "object" replacing "alphanumeric" for each component of the resulting object. By using a "quote object" which returns an unevaluated object, and a "to" operator, that generates sequences, the statement may be used in the following ways:

Traditional Construct	BASIX Format
FOR i IN @a @b @c DO !i := 0	for i = quote (a b c) do ./(i) := 0 rof
FOR i := 1 TO n DO a[i] := 0	for i = 1 to n do a/(i) := 0 rof

Lastly, an object may be any recognisable construct such as:

Construct	Example
expression	a + b - c
pathname	x/y/1
number	10
data structure	(a 10 (11 12))
function call	f(d, e) or f d e
process	(merge a1 a2 a3 a4 a; sort a b)

The object handles both procedure calling and expression evaluation.

As a final illustration of the BASIX language a recursive Quicksort program "rquick" is shown below. The essential idea in Hoare's sorting algorithm is to partition the list of numbers to be sorted into two sublists. The first sublist will contain all numbers less than some arbitrary value ("pivot" - the element v[hi], where the boundary of the list is v[lo] ... v[hi]) chosen from the list, and the second sublist will contain all numbers greater than or equal to the value. This partitioning is recursively applied in turn to the two sublists (rquick(lo, i - 1), rquick(i + 1, hi)) until each sublist contains only one element. When all sublists have been partitioned, the original list of numbers has been sorted.

```
(* the array to be sorted v[0] v[1] ... v[n - 1]  *)

v: (512  87 503  61 908 170 897 426
    765 275 154 509 612 677 653 703)

(* recursive Quicksort - rquick(lo, hi : integer) *)

rquick:
  (lo := $/1 & hi := $/2;
   if lo < hi ->
      (i := lo & j := hi;
       pivot := v/(j);       (* pivot value *)
       do i < j ->
          (do (i < j) and (v/(i) <= pivot) -> i := i + 1 od
           do (j > i) and (v/(j) >= pivot) -> j := j - 1 od
           (* v/(i) and v/(j) are out of order *)
           if i < j -> exchange(v/(i), v/(j)) fi
          )
       od;
       (* move pivot to v/(i) *)
       exchange v/(i) v/(hi)
       rquick i+1 hi & rquick lo i-1
      )
   fi
  )
```

```
(* call Quicksort "rquick(0, n-1)"                    *)
rquick 0 15
```

The Quicksort program shown above may be divided into three sections: at the top is the declaration of the array "v" to be sorted, in the middle is the declaration of the program object "rquick", and at the bottom is the call to "rquick". The array to be sorted is, in fact, the sixteen numbers, 512 ... 703. The corresponding implicit address selectors, from the left, are "0 1 2 3 ..."; alternatively the selectors could have been declared explicitly:

```
v:( 0:512 1:087 2:503 3:061 4:908 ... 15:703 )
```

This is necessary when alphanumeric selectors are used. In the program object "rquick", storage for the variables "lo hi i j pivot" is created on demand. The first line of rquick initialises "lo" and "hi" from the first and second parameters in the call to "rquick"

```
lo := $/1 & hi := $/2 ;
```

The control symbol "&" defines that the two commands are to be executed in parallel. This is followed by the body of the Quicksort which contains calls to two procedures: "exchange" which swaps two elements that are out of order, and the two calls of "rquick" that sort the subsets in parallel. Two formats for calls are illustrated in the above program, the traditional syntax "exchange(...)" and the list of objects "exchange ...". However, the meaning is identical. It should also be noted that the array elements are accessed as "v/(i)" and not as "v/i".

6.2.2 Formal Semantics

No attempt has been made in the work published to describe the semantics of the Basix programming language formally. A formal definition of the semantics of any programming language is useful since it can be used to determine the meaning of a statement written in that language. It has, however, proved extremely difficult to specify the semantics of the Basix programming language formally. An attempt to do so is described below and the reasons for the complexity of the semantics and the ultimate failure of the attempt is outlined.

Intuitively, the operation 'Fetch' which retrieves the value of an object within the memory component of a distributed computing system is relatively simple. Given a memory component and a pathname, this operation yields the value of the object specified by that pathname in the memory component.

Each object in the memory component may either be atomic or compound. An atomic object is simply a number. A compound object consists of a sequence of component objects with which names may be associated. There is an obvious restriction that a name may not be associated with a non-existent object.

```
object = number U composite

composite :: directory : map name to N1
             entries   : seq of object
```

To retrieve an object given a pathname, the selectors in that pathname must be evaluated in a given context. There are two possible contexts - the current context and the dynamic context.

```
state :: current : pathname
         dynamic : pathname
         memory  : object
```

The operation 'Fetch', given a pathname and an initial state, will retrieve the object defined by the pathname within the memory component represented by the state. Additionally the initial state may be changed since evaluation of the selectors within the pathname may have side-effects.

```
Fetch(p : pathname, s : state) o : object, s' : state =
    let s1, i = Transform(p, s, [])
    in  o, s' = Fetch_Object(i, memory(s1))
```

The operation 'Transform' transforms a pathname, which may include symbolic names among the selectors, into a sequence of numeric selectors.

```
Transform(p : pathname, s : state, i : seq of N1)
              s' : state, i' : seq of N1 =
    if len p = 0
    then s', i' = s, i
    else if hd p IN number
    then s', i' = Transform(tl p, s, i ~ [hd p])
    else if hd p IN name
    then s', i' = Transform(tl p, s,
                            i ~ [Fetch_Object(i, memory(s))(hd
p)])
    else if hd p IN object
    then let v, s1 = Evaluate(hd p, s)
         in  s', i' = Transform(tl p, s1, i ~ [v])
    else if hd p = dynamic
    then s', i' = Transform(dynamic(s) ~ tl p, s, [])
    else if hd p = superior
    then s', i' = Transform(tl p, s, i(1, .., len i - 1))
```

```
        else if hd p = current
        then s', i' = Transform(current(s) ~ tl p, s, [])
        else s', i' = s, []
```

Finally, the operation 'Fetch_Object' retrieves an object given a sequence of numeric selectors.

```
    Fetch_Object(s : seq of N1, o : object) o' : object =
        if len s = 0
        then o' = o
        else o' = Fetch_Object(tl s, o(hs s))
```

These specifications ignore the possibility that a side-effect of evaluation of a selector may be to remove an object from the memory component. For example, suppose that the object "o" is the compound object shown below:

```
    o = make_composite({ a -> 1, b -> 3 },
                       [ 9, 12, 15 ])
```

In the context of the object "o", the pathname "/b" denotes the third component object, that is the object with value "15". Removal of the second component, the object with value "12", changes the compound object "o" to be that show below:

```
    o = make_composite({ a -> 1, b -> 2 },
                       [ 9, 15 ])
```

Now consider the case when the third object of the object "o" is itself some other compound object; that is, the object "o" is the compound object show below:

```
    o = make_composite({ a -> 1, b -> 3 },
                       [ 9, 12, make_composite({ x -> 1 },
                                               [ 15 ]) ])
```

If the object denoted by the pathname "/b/(rm /2; 1)" is retrieved, intuitively the object yielded would be that with value "15". However, that is not the case using the operations described above. Evaluation of tThe second selector in the pathname has the side-effect of removing the second component object, that is the object with value "12". Since the first selector "/b" in the pathname has already been transformed into a numeric selector, namely "/3", the operation to retrieve the object will fail to retrieve the correct object.

One possible solution to this problem is to re-evaluate the selectors in a pathname as soon as it is known that one selector has caused a side-effect. However, this solution would result in the side-effect caused by the evaluation of some earlier selector being made to happen again. An alternative solution is to associate a unique identifier with each object in the memory component.

```
object = number U composite

composite :: directory : map name to address
             entries   : seq of address

state : current : pathname
        dynamic : pathname
        memory  : map address to object
        root    : address

Fetch(p : pathname, s : state) o : object, s' : state =
    let s1, a = Transform(p, s, root(s))
    in  o, s' = memory(s1)(a), s1

Transform(p : pathname, s : state, a : address)
          s' : state, a' : address =
    if len p = 0
    then s', a = s, a
    else if hd p IN number
```

```
        then s', a' = Transform(tl p, s, memory(s)(a)(hd p))
        else if hd p IN name
        then s', a' = Transform(tl p, s, memory(s)(a)(directory(hd
    p)))
        else if hd p IN object
        then let v, sl = Evaluate(hd p, s)
             in s', a' = Transform(tl p, sl, memory(sl)(a)(v))
        else if hd p = dynamic
        then s', a' = Transform(dynamic(s) ~ tl p, s, root(s))
        else if hd p = current
        then s', a' = Transform(current(s) ~ tl p, s, root(s))
        else s', a' = s, NIL
```

It has been assumed that sequences of selectors in the form "/a/b/..", where "a" and "b" are arbitrary selectors, have been reduced to "/a".

The drawback to this specification is that it now requires a globally unique identifier to be associated with each object in the distributed computing system. Furthermore, whilst this specification shows the meaning of retrieval, it does so in isolation from all the retrievals that might take place simultaneously.

6.2.3 Informal Semantics

Given the difficulty in presenting the semantics of the Basix programming language formally, the semantics are given informally in this section. but in more detail than in any of the work already published.

Commands

The user interacts with the distributed computing system by issuing commands. These commands are executed and may cause changes to the state of any subordinate computing system of the

distributed computing system.

I : O

An unused memory cell is associated with the identifier I in the current context. The value obtained from evaluation of the object O is stored in this memory cell. An error will result if the identifier I is already associated with a memory cell in the current context. No value is returned as the result of evaluation of this command. There is actually a confusion between the static and dynamic use of this command. The static use allows labels to be attached to portions of the program code. These labels may then be used as the destination of GOTO statements. The dynamic use allows a name to be associated with a memory cell as the program is executed. The interpretation used here is the latter.

O

The object O is evaluated in the current context. The value obtained through evaluation of the object O is not returned.

Objects

Objects are the basic building block of the programming language.

E

The expression E is evaluated in the current context. The evaluation may cause changes to the state of the computing system. The value obtained through evaluation of the expression is returned.

S

The statement S is evaluated in the current context. The

evaluation may cause changes to the state of the computing system. No value is returned.

(O1 ... On)

Each object in the list of objects is evaluated concurrently in the current context. The evaluation of any of the objects may cause changes to the state of the computing system. The order of the evaluation of the list of objects may be visible. The evaluation of the whole construct may cause changes to the state of the computing system. The value returned is a list of values, each of which corresponds to the value obtained through the evaluation of the corresponding object.

(C1 ; C2)

The command C1 is evaluated in the current context. The evaluation may cause changes in the state of the computing system. The command C2 is then evaluated in the new state obtained through the evaluation of the command C1. The evaluation of the command C2 may cause further changes in the state of the computing system. No value is returned.

(C1 | C2)

The commands C1 and C2 are evaluated concurrently in the current context. The standard output of the command C1 is attached to the standard input of the command C2. Either or both of the evaluations may cause changes to the state of the computing system. The changes made during the evaluation of one command are visible to the evaluation of the other command and vice versa. No value is returned.

(C1 & C2)

The commands C1 and C2 are evaluated concurrently in the current context. Either or both of the evaluations may cause changes to the state of the computing system. The changes made during the evaluation of one command are visible to the evaluation of the other command and vice versa. No value is returned.

Expressions

N

The location associated with the name N is retrieved; the state of the computing system may be changed.

N [ ]

The value associated with a message variable with name N is retrieved.

B

The value of the basic value B is returned; the state of the computing system is not changed.

( )

Evaluation is suspended until some other processing activity causes the undefined value to be over-written with a defined value.

QUOTE O

The text of the object O is returned; the state of the computing system is not changed.

O1 X O2

The value of evaluating the operator X with the two objects O1 and O2 as operands is returned. The two objects O1 and O2 are

evaluated concurrently in the current context. Either or both of the evaluations may cause changes to the state of the computing system. The changes made during the evaluation of ane object are visible to the evaluation of the other object and vice versa. Furthermore, the operator X may be the assignment operator ':='. Evaluation of the expression 'O1 := O2' changes the state of the computing system by causing the value obtained from the evaluation of the object O2 to be stored at the memory cell associated with the name gained by evaluation of the object O1.

```
N ( )
N ( O1 ... On )
```

The name N is evaluated, and the value stored at the memory cell associated with the pathname is taken to be some Basix source text. The objects O1 ... On are stored at memory cells associated with the pathnames $/1 ... $/n. The evaluation of the Basix source text occurs in a context in which the arguments are accessible using the selector '$'.

```
O O1 ... On
```

The object O is evaluated and the value is taken to be the Basix source text of some object, perhaps obtained by evaluation of an expression of the form 'quote O'. The objects O1 ... On are stored at memory cells associated with the pathnames $/1 ... $/n. The evaluation of the Basix source text occurs in a context in which the arguments are accessible using the selector '$'.

Statements

```
IF ( O11 -> O12 ;
          :
```

```
        On1 -> On2 ;
        Ox         )
    FI
```

Each of the objects Oil, 0 < i <= n, is evaluated in the current context until one such Oil is found which evaluates to the boolean value "true". The corresponding object Oi2 is then evaluated in the current context. The evaluation of the objects Oil, 0 < i <= n, may cause changes to the state of the computing system. Similarly the evaluation of the object Oi2 may cause a change to the state of the computing system. If no object Oil, 0 < i <= n, is found which evaluates to the boolean value "true", the object Ox is the evaluated in the current context.

```
    DO ( Oil -> Oi2 ;
           :
        On1 -> On2 ;
        Ox         )
    OD
```

Each of the objects Oil, 0 < i <= n, is evaluated in the current context until one such object is found which evaluates to the boolean value "true". The corresponding object Oi2 is then evaluated in the current context. The evaluation of the objects Oil, 0 < i <= n, may cause changes to the state of the computing system. Similarly the evaluation of the object Oi2 may cause changes to the state of the computing system. Once the object Oi2 has been evaluated, the process is repeated. If no object Oil, 0 < i <= n, is found which evaluates to the boolean value "true", then the process stops with the evaluation of the object Ox in the current context.

```
    FOR I = O1 DO O2 ROF
```

The object O1 is evaluated in the current context. It is expected that this evaluation will return a list of values (V1 ... Vn). The identifier I is then associated with each of the values Vi, 0 < i <= n, and the object O2 is evaluated in a context in which the identifier is accessible. Consequently the object O2 is evaluated n times with the identifier I bound to a different value of Vi on each iteration. Each evaluation of the object O2 may cause changes to the state of the computing system.

GOTO N

The pathname N is evaluated and is expected to refer to some memory cell which contains Basix source text. Evaluation of the Basix source text continues from this point.

CD N

The current context in which names are resolved is changed to the context denoted by the name N. The name N is not evaluated.

RM N1 ... Nn

Each of the names Ni, 0 < i <= n, is evaluated to refer to some memory cell. The memory cell thus referenced is then made inaccessible; any future references to these memory cells will yield an error.

Pathnames

Pathnames are used to access variables. A pathname consists of a sequence of selectors which determine a route to a particular object within the information structure. The first selector of a pathname determines the starting point for the route; by default this is the information structure which is locally accessible to

the command being executed.

/P0/P1/ ... /Pn

The memory cell referenced is found by evaluating each of the selectors Pi, 0 < i <= n, in the local context. The memory cell referenced is that which matches the name starting at the root of the memory component of the computing system.

$/P1/ ... /Pn

The list of parameters in the current context are referenced. The selectors Pi, 0 < i <= n, are each evaluated to determine the actual parameter to be referenced.

../P0/ ... /Pn

The superior context of the current context is referenced. The selectors Pi, 0 < i <= n, are evaluated to determine the memory cell to be referenced.

./P1/ ... /Pn

The current context is referenced.

I/P1/ ... /Pn

The memory cell associated with the identifier I in the current context is referenced.

Selectors

A selector identifies a particular object within an information structure. The information structure reference is that derived by evaluation of the preceding selectors in the pathname. By default this is the information structure which is locally accessible to the pathname when it is evaluated. Named objects within the information structure may be accessed by using

identifiers. This corresponds to the access of simple variables or fields of record variables in programming languages such as Pascal and Algol 68. The equivalent of array element access is obtained by using a parenthesised object as a selector.

I

The identifier I selects a memory cell in the path context which has the identifier I associated with it.

..

The superior context to the path context becomes the path context for any remaining names in the pathname.

B

The basic constant B is evaluated. It should evaluate to a number n. The nth item in the path context becomes the path context for any remaining selectors in the pathname.

(O)

The object O is evaluated in the current context. It should evaluate to a number n or an identifier i.

6.2.4 Analysis

Unlike the programming languages surveyed in this chapter, the Basix programming language represents a very simple approach to the problem of designing a programming language for the construction of software for general purpose distributed computing systems.

Restrictions have been introduced into many of the programming languages designed for this area to enforce some form of order in the

potential chaos. For example, restricting the communication between processes to named channels, as is found in both Pascal-m and Occam, restricts the interface between processes. Consequently, when the processes are executed concurrently, the only interactions which may occur do so by the sending and receipt of messages through channels. In contrast, the Basix programming language reflects a total generalisation of the control flow philosophy. That philosophy is probably best displayed in the freedom allowed in the programming language Basic. In that programming language there is a minimum number of restrictions, but the programming language presents very few "high-level" features to the programmer. For example, there is no concept of scope, and the subroutine mechanism is extremely crude, allowing neither parameters nor recursion. One statement in a program can have a side-effect which affects statements far away. Additionally, there may be several threads of control within a program which, given the simplicity of the programming language, actually makes it quite difficult to understand what affect a program will have on the computing system. There are, for example, no mechanisms in the Basix programming language to enforce a serial scheduling on the concurrent execution of objects. Execution of an object may cause, as a side-effect, some non-local change to the computing system. It is these characteristics which have made it difficult to define the semantics of the programming language formally.

The complexity in understanding the semantics of the programming language Basix arises because of the side effects which may occur during the evaluation of the different constructs of the programming language. Firstly, consider an expression of the form "O1 X O2", where O1 and O2 are both objects and X is an operator. The context in which the objects O1 and O2 are evaluated must be considered, and it must be determined whether any side effect in the evaluation of the objects O1 and O2 should be made visible to either evaluation. Furthermore, the order of evaluation of the objects O1 and O2 may effect the value of the whole expression. Secondly, consider an object of the form "C1 & C2", where C1 and C2 are both commands. The concurrent evaluation of the commands C1 and C2 may result in the state of the computing system being changed. These changes should be visible to each evaluation as they are made.

The side effects of the evaluation on one construct of a program is propogated to all concurrent evaluations of the other constructs of the program. This has made the production of the formal semantics for the programming language extremely desirable, yet somewhat difficult; it is desirable to understand the combination of these side effects with the concurrent evaluation of the constructs of the programming language. Furthermore, this difficulty is compounded since these side effects are not only visible to the user of the computing system for whom they were made, but also to all other users of the computing system. Consequently, to construct a proof for a program, it is not sufficient to consider that program in isolation

from all other programs.

One of the strengths of the Basix programming language is the lack of restrictions. However, at the same time, this is also one of the weaknesses of the programming language. It is precisely the lack of these restrictions which makes it hard to reason about the behaviour of a program written in the Basix programming language. Another strength is the total integration of the hierarchical nature of a distributed computing system into the programming language. For example, access to a file within the fielstore of a distributed computing system is no different from access to a program variable. Yet once again this strength is also an area of weakness. Within a filestore it is possible to insert new files and to destroy existing files. These operations can be performed dynamically. Within a program the conventional way of creating new variables and removing old variables is, respectively, through explicit declaration at block entry and implicity at block exit. Combining the files of the filestore and the variables of the programs in this simple straightforward manner does not seem to be appropriate. Additionally, the hierarchical structure is globally accessible which leads to the obvious problems of concurrent sharing.

6.3 A DECENTRALISED CONTROL FLOW PROGRAMMING LANGUAGE

The programming language proposed for the decentralised control flow computing systems is still under active consideration. In this section some of the design issues for the proposed programming language are outlined.

The proposal is to follow the route taken by many other designers; the programming language Pascal can be used as a core on which different features may be built. Using the programming language Pascal as the basis of development also has the advantage that compilers for the programming language are readily available.

6.3.1 Overview

The concept of object on which the decentralised control flow model of computation is based has played an important role in the design of the programming language. Objects are used by the programmer to describe both the data and control characteristics of various entities. For example, an object may describe both the underlying data representation of a stack, and the means by which that data representation may be manipulated to give the behaviour of a stack. Control structures are associated with the object thereby describing how the data structures may be accessed. It is argued that the way in which the data structures are accessed is as an important part of the description of the whole object as the

representation of the data itself. It is generally accepted that a data type is not merely a set of values but also a set of operations which may be used on those values. Abstract data types have played a significant role in the design of objects. Indeed, the concept of object can be used to remove the distinction between those entities, such as files, which conventionally have an existence beyond that of a program and those entities, such as variables, which exist only during the execution of a program.

The concept of object which is used to encapsulate data and control has been stressed in contrast to that of program which conventionally only represents control. Firstly, what is considered to be a program today might form only a small part, either statically or dynamically, of some larger program tomorrow. It is often desirable to be able to re-use programs, or parts of programs. Sometimes what is required is the inclusion of the data representation and associated code in some other program so that a particular data structure can be used in that other program. Alternatively, access to some pre-existing data structure is required, the actual one not necessarily being known at the time that the program was written. It can often be achieved dynamically through the use of facilities provided by the control program of the computing system. Such facilities are not described in terms of the semantics of the programming language.

In some von Neumann programming languages it is difficult to construct programs from existing programs. For example, in the programming languages Pascal and Fortran the program is an independent unit which may only be executed. The program text cannot be used to form a section of some larger program nor may one program communicate with some other program. In the programming language Algol 68 it is possible to use the text of one program within some other program, but again it is not possible for different programs to communicate. In the programming language C a program may be invoked dynamically as a routine of some other program through the use of the system call "EXEC". The called program is compiled separately from the calling program. Both the calling program and the called program are executed as commands which are active concurrently. The calling program and the called program may communicate through the use of pipes and files, facilities which are provided by the control program rather than by the programming language.

Secondly, the concept of object permits the dynamic reconfiguration of software systems. Changes may be made to a large complex system by the construction of a new object to replace an existing object. Clearly any references to the existing object must be satisfied until no references remain anywhere throughout the whole distributed computing system at which point the existing object may be removed. This is obviously inappropriate for those objects which provide a basic service to the whole distributed computing system since it may never be possible to remove all references to such

objects. In those instances it will be necessary to place the distributed computing system into some quiescent state and then make the replacement.

Thirdly, association of data structures and control structures allows the data stored within the computing system to be described more precisely. That is, rather than simply having an object represent some data, information about the useful operations which may be performed on that data is also represented. This information is associated with the data itself and not with the programs which use that data as is often the case in conventional computing systems.

6.3.2 Syntax

The textual description of an object consists of three distinct sections. Firstly, there is the description of the local data space which will come into existence whenever an instance of the object is created. Secondly, there is the description of the group of routines with which the local data space may be manipulated. Thirdly, there is the description of the additional routines which provide the interface between an instance of this object and instances of other objects.

An outline of the syntax for the description of objects is given below:

OBJECT <identifier>;

```
  <constant declarations>
  <type declarations>
  <variable declarations>
  <routine declaratrions>
  INTERFACE
    <routine declarations>
BEGIN
<statement list>
END;
```

The declarations between the reserved words OBJECT and INTERFACE are used to describe the local data space of an instance of the object and the group of routines which may be used within the instance of the object to manipulate the local data space. The scope of the identifiers introduced in these declarations extends from the point of declaration up to the reserved word END which encloses the description of the object. The declarations between the reserved words INTERFACE and BEGIN are used to describe the additional routines which define the external interface to an instance of the object. The nature of these routines is discussed in more detail below. The scope of the identifiers introduced in these declarations extends from the point of declaration up to the reserved word END which encloses the description of the object. The external interface to the local data space should be be kept independent of the nature of the local data space. To achieve this it is necessary to restrict the types of the parameters which may be specified on the interface routines. Clearly it is not possible for the interface to be defined in terms of types which are local to the object itself. Furthermore, it is possible that the type of a parameter is unknown - it depends upon the use being made of an object.

Provision has been made for the inclusion of initialisation code within the description of an object. Any statements which appear between the reserved words BEGIN and END in the definition of an object are executed as soon as an instance of that object is created.

As mentioned above, to make objects generally useful, the programming language will need to support polymorphic types. This allows the programmer to describe the behaviour of a family of related objects with a single textual description. For example, the description below could be used to describe the behaviour of a stack for a particular data representation based on arrays.

```
OBJECT StackHandler(t : TYPE);
  CONST StackSize = 100;
  TYPE  Stack = RECORD
                  NrOfEntries : 0 .. StackSize;
                  Entries     : ARRAY [ 1 .. StackSize ] OF t
                END;

  VAR S : Stack;

  PROCEDURE Empty;
    BEGIN
    S.NrOfEntries := 0
    END { Empty };

  INTERFACE

    PROCEDURE Push(x : t);
      BEGIN
      S.NrOfEntries := S.NrOfEntries + 1;
      S.Entries[S.NrOfEntries] := x
      END { Push };

    PROCEDURE Pop;
      BEGIN
      S.NrOfEntries := S.NrOfEntries - 1
      END { Pop };

    FUNCTION Top : t;
      BEGIN
```

```
      Top := S.Entries[S.NrOfEntries]
      END { Top };

    FUNCTION IsEmpty : Boolean;
      BEGIN
      IsEmpty := S.NrOfEntries = 0
      END { IsEmpty };

    FUNCTION IsFull : Boolean;
      BEGIN
      IsFull := S.NrOfEntries = StackSize
      END { IsEmpty };

  BEGIN
  Empty
  END { StackHandler }
```

The formal parameter "t" to the object "StackHandler" may be replaced by any valid type. The underlying type of the elements of the stack is only defined when a particular instance of a stack is required. For example, replacing the formal parameter "t" by the type "char" defines stacks with elements of type "char". For it to be possible to pass type information as an actual parameter, the information about a type must be both available inside and outside of a program text. Clearly some types, such as "char" and "integer", can be regarded as built-in types. However, other types will be defined by the users of the computing system.

It is not possible to use procedures and functions as the parameters of objects. This restriction has been adopted purely for pragmatic reasons. It seems unreasonable to expect a procedure or function to be used a parameter for the following two reasons. Firstly, the procedure or function must by virtue of the scope rules be independent of the definitions of the object. Secondly, on application of the procedure or function control is passed back out

from the instance of the object. If the procedure or function has side-effects then these will affect the 'called' instance. However, it is possible to pass a reference to an instance of an object as a parameter. References to instances of objects may be assigned to variables in the expected way. For example, the following two lines declare two variables as references to objects.

```
VAR o : INSTANCE;
    s : StackHandler(char);
```

The variable "o" may contain a reference to any object, whilst the variable "s" may contain a reference to any object which is a member of the set of "StackHandler" objects with an elements of type "char".

Two alternative strategies have been considered for the description of the additional routines which describe the external interface of an object. These strategies reflect the difference between messages and procedure calls outlined in this chapter. Since the instances of different objects may reside on different subordinate computing systems, the external interface between these instances must be implemented in terms of the underlying primitives of the communications medium which is used to connect the subordinate computing systems into a distributed computing system. Consequently, the external interface could be described by using the primitive operations which cause individual messages to be sent from one instance of an object to some other instance of an object. Thus the description of an object would include code first to receive a message from an instance of an object, then to process the request

borne within that message, and finally to send a message bearing a response to the instance making the request.

Alternatively, this relatively low level approach could be hidden by adopting the semantics of the remote procedure call. The semantics of the remote procedure call require that the instance of an object which makes a request of some other instance of an object waits until the request has been satisfied. In effect, control is passed from the requesting instance to the one to which the request is addressed; when the request has been satisfied, control is returned to the requesting instance. Naturally this will restrict the degree of parallelism which may be achieved in comparison to that which could be exploited using the message passing primitives.

Use of the primitive send and receive operations does allow other instructions of the instance of the object making the request to be executed whilst that request is being serviced. This allows both the instructions of the instance making the request and those of the instance to which the request is addressed to be executed concurrently. However, some care must be exercised if this strategy is adopted. The instructions executed between the sending of the message making the request and the receipt of the message bearing the reply must not be dependent upon any information contained in the reply. It is possible by an analysis of the source program during compilation to identify those instructions of the programs which may be executed during the processing of the various requests addressed

to other instances of objects. Some form of data flow graph must be constructed so that the instructions which may be executed between the send and receive operations may be identified. Generally, the send operation should be executed as soon as possible, whilst the receive operation should be executed as late as possible.

6.3.3 Semantics

The decentralised control flow model of computation is reflected within the single language which is used both as the control program of a subordinate computing system and as the programming language used to write software for that computing system. A user works within an environment known as the current context consisting of those objects to which he or she has access. As the user works, new objects may be created and existing ones destroyed.

The current context is simply an instance of a special variety of object, a directory, which contains a mapping from names to references of instances of objects. These instances represent information which in traditional computing systems would be classified separately as programs or data. Associated with each object defined by the user are instances of two other objects. An instance of the first object is used to represent the textual description of the object. It corresponds to a text file containing the source program in a conventional computing system. Various operations may be performed on an instance of such an object. One

such operation is compilation. Compilation of the instance produces an instance of the second object, an internal representation of the object which is used by the run-time system to determine the type of the object and to create new instances of the object.

The definitions of the objects made by a user which are represented by the instances of these two objects are all entered into the current context. The message "NEW" when sent to the current context with the name of an object, together with the actual arguments required, causes a new instance of that object to be created. For example, using the description of an object given earlier, the command "NEW StackHandler(char)" sent to the current context in which the definition of "StackHandler" is held returns a reference to an instance of this object.

To enable an instance of an object to use the facilities of other instances a mechanism is required to associate a reference to an existing instance of an object with a variable within the local data space of some other instance of an object. Associated with each instance of an object is a reference to the instance of the directory object in which the definition of that object is held. Use of this reference allows other references to instances of objects accessible from that directory to be obtained. This gives a static name scope. Alternative scopes may be implemented by transmission of references to objects. Additionally, an instance of an object can be saved in an instance of the directory object.

As mentioned earlier, type information is associated with each object. This information describes the external interface and allows the requests which instances of other objects may make of an instance of this object to be checked for type correctness. The type of an object persists throughout the extent of that object. It must be available to both the compiler and the run-time system.

The textual description of an object might state precisely the instances of the various objects with which it will communicate. Such a description can have strong typing applied to it when it is compiled.

Alternatively, the textual description of an object might give no details of those instances of the objects with which it will communicate. Such a description will require run-time type checking to be performed. This will be used to ensure that a request made of an instance of an object is defined.

The need for run-time type checking arises because a reference to an instance of an object may be sent to some other instance. This second instance may then make a request of the instance whose reference was received. That request must be checked for validity. The ability to send references to instances of objects has the advantage that it allows the topology of the objects to change dynamically. A general purpose server object may assume that an object to which it has a reference has certain properties. For

example, the object which controls a printing device will receive the references to those objects which are to be printed. The objects to be printed will be assumed to have a standard operation which will yield some printable representation of that object.

For example, it may be assumed that the operation "unparse" is defined for every object. This operation produces a printable representation of an instance of that object as a sequence of characters. This sequence of characters may then be displayed on some suitable output device. When an instance of an object which controls such an output device receives a request to print an instance of an object, the operation "unparse" is invoked to yield the string representation of the instance, and this is then displayed. Clearly, for this scheme to work, the operation "unparse" must be defined for all objects which may be printed. The object controlling the output device will be written without knowledge of the possible instances of objects which it may receive requests to print. Consequently, it is possible that the operation "unparse" may not be defined for a particular object, and some appropriate action must be taken. The responsibilty of providing some mechanism to produce a printable representation of an instance of an object thus lies with the user who provided the definition of that object.

The binding of a reference to an object to a particular name within the text of an object must be delayed until run time for external objects. Compile time binding will make it impossible to

replace objects dynamically since the reference becomes fixed within the text.

The data structures of an object are local to that instance of the object. The control structures may be shared among several instances of different objects. Receipt of a message causes the control structures of an object to be executed; the execution of these may cause changes to the data structures.

An instance of an object consists of a local data space and a group of routines to manipulate this data space. The local data space of an instance of an object is accessible only to the group of routines of the same instance of that object. To allow other instances of objects to gain access to the local data space of an instance of an object, some additional routines must be provided. These additional routines define the interface between the instance of an object and all other instances of objects in the computing system. Again, it is necessary for type information to be available outside the textual description of an object so that users of instances of that object may send requests to those instances.

To ensure that the local data space cannot be accessed or modified by instances of objects other than the one with which it is associated, various restrictions have been made on the information which may be passed between instances of objects.

No reference or pointer values may be passed between instances of objects. This ensures that execution of an instruction of one instance of an object may not access or modify the local data space of some other instance of an object indirectly. Clearly there are other reasons for this restriction beyond that of simply minimising the effect that a globally accessible memory component has on the semantics of the programming language. Permitting a reference or pointer value to migrate from one instance of an object to some other instance of an object may result in a memory address for one subordinate computing system of a distributed computing system being transmitted to some other subordinate computing system. In order to use the reference value correctly some information about the origin of that reference value would also be required.

This restriction on the use of reference or pointer values means that not only can such values not be used as parameters to additional routines which describe the external interface, but also that the parameters of these routines must have call by value or call by value/result semantics. If a request is made to an instance of an object which requires some modification to be made to a variable within the local data space of the instance of an object making the request, then, since the modification can only be made by the instance of the object associated with the local data space, the new value of the variable must be sent from the instance of the object to which the request.

For example, given the following declaration for a routine for an external interface,

```
PROCEDURE P(VAR X : INTEGER)
```

use of this routine might be written as

```
[X P(A)]
```

where X is a local variable containing a reference to an instance of an object for which P is declared as an interface routine, and A is a variable within the local data space of the instance of the object making the request. To comply with the restriction that no reference values may be transmitted between instances of objects, the use of the routine could be expanded out into the following sequence of instructions using the primitive send and receive operations of the underlying communications medium:

```
Request.RoutineName := "P"
Request.Argument[1] := A
SEND(X, Request)
RECEIVE(X, Reply)
A := Reply.Argument[1]
```

6.3.4 Analysis

The decentralised control flow programming language has taken the route of those programming languages surveyed in this chapter. The programming language is restrictive so that order may be brought out of potential chaos. In particular, an attempt has been made, as outlined in chapter five, to address the problem of concurrent access to an instance of an object from several other instances. The problem is to maintain the different instances of the objects in a consistent state in the face of these concurrent requests. In

particular, the requirement is to ensure that a schedule equivalent to a serial schedule is always achieved. This issue is not addressed by the progamming languages surveyed. For example, the programming language PS-Algol assumes that the unit of concurrency will be the whole memory component. This is too coarse a level for a distributed computing system. In contrast, the Occam programming language assumes that all the potential users of a process are known at the time that the process is written. The programming language Argus comes the closest to the requirement.

6.4 CONCLUDING REMARKS

When designing a programming language different aims must be kept in tension. It is essential that the programming language can be used by programmers without requiring them to be skilled mathematicians or logicians. To that end a programmer should be able understand the meaning of a program without recourse to complex statement in some special calculus. Additionally it is important that the programming language provides an environment in which the programmer can get on with the real task of programming and need not be hindered by low-level concerns. The provision of suitable abstraction mechanisms which allow the programmer to think in terms of the problem domain rather than the computing system obviously help to achieve this aim.

The Basix programming language is perhaps deceptively simple. It reflects a model in which any operation can be performed on almost any object. There are few restrictions in the programming language. However, it is this seeming simplicity which gives rise to the complexities in using the Basix programming language. No mechanisms for controlling concurrency have been provided. The globally accessible memory component is a hindrance to the understanding of program.

In contrast, the decentralised control flow programming language, through its restrictions, is altogether a superior programming language. The hierarchical structure of the distributed computing system is reflected through the abstraction of objects. This abstraction allows the objects defined to have a clear interface and the behaviour can be described formally.

7 CONCLUSIONS

The advances in the chip fabrication technology and the communications technology have brought about new possibilities for the design and construction of computing systems. In particular, the former has made it possible to design novel architectures of computing system whilst the latter permits several computing systems to be connected together to form a single distributed computing system. Such developments could be used to introduce a new era to computing science in which concurrency is the norm rather than the exception.

The design of the majority of contemporary computing systems still reflects the principles on which the computing systems of the mid 1950's were based. These computing systems existed in isolation from one another, thus making it difficult to share information between the different computing systems. Typically such computing systems comprised a processing unit, a memory component, and some input/output devices. A program for these computing systems consists of a sequence of instructions. Each instruction in the sequence is executed in turn by the processing unit.

The availability of cheap mini- and microcomputing systems has led to the use of computer technology in an increasing number of new application areas. The complexity of these tasks has resulted in an increased complexity in the software systems written for these

applications. By the mid 1970's a crisis in the development of software systems had been identified. In part this crisis was attributed to the inappropriate design principles of the computing systems for which these software systems were written.

To alleviate the problems encountered in the production of large software systems, novel models of computation have been developed. These models of computation have a formal mathematical basis which makes the construction of proofs of the correctness of programs an easier task than is possible for the von Neumann model of computation. One of the drawbacks in the use of programming languages based on these models of computation lies in the overheads incurred in the implementaions designed for von Neumann architecture computing systems. To support these novel models of computation more efficiently, novel architectures of computing system have been proposed. As yet, these computing systems are still in an embryonic state.

It has also been proposed that the chip fabrication technology will advance to the point where several computing systems may be constructed on a single chip. Such chips are being proposed as the building blocks of computing systems consisting of many subordinate computing systems. It will be possible to construct general purpose distributed computing systems from these smaller computing systems. Each such distributed computing system consists of a group of subordinate computing systems connected together by some

communications medium. Information may be transmitted between the subordinate computing systems and the programs executed on the different subordinate computing systems may co-operate.

7.1 AIMS

Whilst the work of those researchers investigating novel models of computation and novel architectures of computing system may well prove fruitful in the years to come, the possibilities of distributed computing systems are already here to be exploited. Such computing systems consist of a number of subordinate computing systems connected together by a communications medium. Each subordinate computing system is autonomous executing programs which may co-operate with the execution of other programs through the transmission of information across the communications medium. The availability of cheap, yet powerful, microcomputing systems, such as those based around the M68000 family of processing units, make the construction of these distributed computing systems an attractive prospect. This seems to be an exciting possibility for the development of the computing systems of the 1990's.

One of the aims of the work reported in this thesis has been to design a suitable model on which general purpose distributed computing systems can be built. Various different architectures of computing systems and models of computations have been analysed in this thesis. None of these architectures approximated satisfactorily

to the architecture of a general purpose distributed computing system outlined above. Consequently, none of the models of computation associated with these architectures appear suitable for the description of software systems to be executed on the distributed computing systems proposed.

An additional aim was to design a programming language with which software for these general purpose distributed computing systems could be written. Many programming languages proposed in the literature for this application area have been analysed. Again, it was found that none of these programming languages provided exactly what was required.

7.2 ACHIEVEMENTS

Two models of computation, based on two different architectures of computing system, have been proposed to take advantage of the developments outlined above. The recursive control flow architecture and and its associated model of computation originated in the work of Treleaven and Hopkins. This work was initiated by the interest in using VLSI components to build recursive computing systems. A recursive control flow computing system is composed of a hierarchy of subordinate computing systems. Each subordinate computing system has a processing unit, a memory component, and some input/output devices. Information may be transferred between subordinate computing systems. A subordinate computing system may be requested by some other

subordinate computing system to retrieve the value of some cell of the memory component. Thus, whilst the memory components of the subordinate computing systems are globally accessible, the access to a particular memory component only occurs through the agency of the subordinate computing system with which that memory component is associated. The processing units of the subordinate computing systems may execute different sequences of instructions concurrently.

The recursive control flow model of computation reflects the recursive control flow architecture. The memory components are globally accessible. There may be many threads of control within a group of instructions. Consequently, problems of inteference and integrity between the concurrent execution of sequences of instructions may arise. These issues have not been addressed by the designers of the model of computation.

The production of the formal semantics for this model of computation outlines some of the complexities inherent in this design. In particular, the combination of concurrency and the globally accessible memory component makes the construction of proofs of software systems extremely difficult. The level at which the recursive control flow system must be modelled in the formal semantics in order to capture the combined effects of the concurrency and the globally accessible memory component is that of the micro instruction. It seems unreasonable to expect a user of a recursive control flow computing system to be interested in this fine level of

detail.

Furthermore, since the recursive control flow computing system may be used by several users concurrently, each of whom may cause changes to the overall computing system, these changes must be made visible to all other users of the computing system. Consequently, in constructing a proof of some software system written for a recursive control flow computing system, it is not sufficient to regard that software system in isolation from all other software systems. The complexity of considering how any software system, including those yet to be designed or constructed, might interact with a particular computing system is far too great.

Clearly this problem is not confined to the recursive control flow architecture. Any computing system which can be used to support concurrency and has a globally accessible memory component leads to exactly the same problems. To verify that the behaviour of a software system is correct on such a computing system all other software systems whose execution could be interleaved with the first must be considered.

An obvious solution is to restrict the potential interaction or interference between different software systems. For example, on many contemporary computing systems the only interaction which may take place between different software systems is at the level of the file store maintained by the control program.

The programming language BASIX has been desgined as a sutiable programming language for the writing of software for distributed computing systems. It reflects the openness of the underlying recursive control flow computing system for which it is designed.

The alternative design of architecture proposed in this thesis for the construction of general purpose distributed computing systems is decentralised control flow. It has many similarities to recursive control flow. However, the important distinction is the lack of global accessibility to the memory components of the subordinate computing system. The subordinate computing systems still permit the state of the associated memory components to be examined, but the distinction is that more control is given to the subordinate computing system. The model of computation for the decentralised control flow architecture of computing system reflects this difference. The sequence of instructions executed by a subordinate computing system defines more rigorously what may be communicated to other subordinate computing systems. A clear interface may be defined between the different subordinate computing system. Additionally, an attempt has been made to address the issues of interference and integrity.

The programming language for the decentralised control flow computing systems uses the concept of an object to represent this interface. This programming language enforces separation between different software systems, and different parts of those software

systems.

7.3 FUTURE WORK

As yet no physical hardware system has been constructed which embodies the principles of the decentralised control flow architecture. A small test bed system is required. The programming language proposed for the decentralised control flow computing systems needs further attention; the formal semantics must be checked thoroughly, and an implementation of the programming language and the necessary run time support environment developed for the test bed system. The two can then be used to develop software systems.

Tools would be developed to investigate the performance of the decentralised control flow computing systems. Of particular interest are the degree of parallelism in the computing system, and the amount of roll-back which occurs. Ways in which the former can be maximised and the latter minimised would be investigated.

Another interesting area is the implementation of the decentralised control flow model of computation on the parallel control flow style computing systems. Such computing systems have a globally accessible memory component. The implementation of the decentralised control flow model of computation would be based on this memory component and not on message passing. This would allow the same model of computation and the same programming language to be

used for two distinctive styles of computing system. The aim would be to show that the decentralised control flow model of computation can be used just as effectively on either style of computing system.

BIBLIOGRAPHY

ABRAMSKY, S., BORNAT, R. (1983)
Pascal-m: A Language for Loosely Coupled Distributed Systems, in [Paker and Verjus, 1983], pp. 163-189.

ACKERMAN, W.B., DENNIS, J.B. (1978)
VAL - Preliminary Reference Manual,
Laboratory for Computer Science, MIT.

ALEXANDER, W.C., WORTMANN, D.B. (1975)
Static and Dynamic Characteristics of XPL Programs,
Computer, Vol. 8, No. 11, pp. 41-46.

AMBLER, A.L., GOOD, D.I., BROWNE, J.C., BURGER, W.F., COHEN, R.M., HOCH, C.G., WELLS, R.E. (1977)
Gypsy: A Language for Specification and Implementation of Verifiable Programs,
SIGPLAN Notices, Vol. 12, No. 3, pp. 1-10.

ARVIND, GOSTELOW, K.P., PLOUFFE, W. (1978)
An Asynchronous Programming Language and Computing Machine,
Department of Information and Computer Science,
University of California at Irvine.

ASHCROFT, E.A., WADGE, W.W. (1977)
LUCID, A Nonprocedural Language with Iteration,
Communications of the ACM, Vol. 20, No. 7, pp. 519-526.

ATKINSON, M., CHISHOLM, K., COCKSHOTT, P. (1981)
PS-algol: An Algol with a Persistent Heap,
Department of Computer Science, University of Edinburgh.

ATKINSON, M.P., BAILEY, P.J., CHISHOLM, K.J., COCKSHOTT, W.P., MORRISON, R. (1983)
PS-Algol Papers,
Department of Computer Science, University of Edinburgh, and
Department of Computational Science, University of St. Andrews.

BACKUS, J. (1978)
Can Programming be Liberated for the von Neumann Style? A

Functional Style and its Algebra of Prorams,
Communications of the ACM, Vol. 21, No. 8, pp. 613-641.

BEN-ARI, M. (1982)
Principles of Concurrent Programming,
Prentice-Hall.

BERKLING, K. (1975)
Reduction Languages for Reduction Machines,
Proceedings of the Second International Symposium on
Computer Architecture, pp. 133-140.

BROOKER, R.A. (1958)
The Autocode Programs Developed for the Manchester University
Computers,
Computer Journal, Vol. 1, No. 1, pp. 15-21.

BUCKLE, J.K. (1978)
The ICL 2900 Series,
London: Macmillan Press.

BURNETT-HALL, D.G., DRESEL, L.A.G., SAMET, P.A. (1964)
Computer Programming and Autocodes,
London: English Universities Press.

BURSTALL, R.M., DARLINGTON, J. (1977)
A Transformation System for Developing Recursive Programs,
Journal of the ACM, Vol. 24, No. 1, pp. 44-67.

CAMPBELL, R.H. (1983)
Distributed Path Pascal,
in [Paker and Verjus, 1983], pp. 191-223.

CHAMBERLIN, D.D. (1971)
Parallel Implementation of a Single Assignment Language,
Digital Systems Laboratory, Stanford University.

CLARKE, J.W., GLADSTONE, P.J.S., MACLEAN, C.D.,
NORMAN, A.C. (1980)
SKIM - S, K, I Reduction Machine,
Proceedings of the LISP-80 Conference.

COWLISHAW, M.F. (1984)
The Design of the REXX Language,
IBM Systems Journal, Vol. 23, No. 4, pp. 326-335.

DAHL, O-J., DIJKSTRA, E.W., HOARE, C.A.R. (1972)
Structured Programming,
London: Academic Press.

DARLINGTON, J., REEVE, M. (1981)
ALICE: a Multiprocessor Reduction Machine for the Parallel
Evaluation of Applicative Languages,

Proceedings of the Conference on Functional Languages and Computer Architecture.

DARLINGTON, J., HENDERSON, P., TURNER, D. (eds) (1982)
Functional Programming and Its Applications, an Advanced Course,
Cambridge: Cambridge University Press.

DIJKSTRA, E.W. (1976)
A Discipline of Programming,
Prentice-Hall.

ESWARAN, K.P., GRAY, J.N., LORIE, R.A., TRAIGER, I.L. (1976)
The Notions of Consistency and Predicate Locks in a Database System,
Communications of the ACM, Vol. 19, No. 11, pp. 624-633.

FLORES, I. (1971)
Job Control Language and File Definition,
Englewood Cliffs, N.J.: Prentice-Hall.

FRIEDMAN, D.P., WISE, D.S. (1979)
A Constructor for Applicative Multiprogramming,
Technical Report 80, Computer Science Department,
University of Indiana.

GLUSHKOV, V.M., IGNATYEV, M.B., MYASNIKOV, V.A., TORGASHEV, V.A. (1974)
Recursive Machines and Computing Technology,
Proceedings of the IFIP Congress, pp. 65-70.

GOLDSTINE, H.H. (1972)
The Computer from Pascal to von Neumann,
Princeton, N.J.: Princeton University Press.

GOUVEIA LIMA, I., HOPKINS, R.P., MARSHALL, L.F., MUNDY, D.H., TRELEAVEN, P.C. (1983)
Decentralised Control Flow - Based on UNIX,
SIGPLAN Notices, Vol. 18, No. 6, pp. 192-201.

GOUVEIA LIMA, I. (1984)
Programming Decentralised Computers,
Ph.D. Thesis, University of Newcastle upon Tyne.

GRIES, D. (1981)
The Science of Programming,
New York: Springer-Verlag.

HENDERSON, P., MORRIS, J.M. (1976)
A Lazy Evaluator,
Proceedings of the Third Conference on the Principles of Programming Languages, pp. 95-103.

HENDERSON, P. (1978)
Lispkit System, a Software Kit,
Technical Report 129, Computing Laboratory,
University of Newcastle upon Tyne.

HENDERSON, P., JONES, G.A., JONES, S.B. (1983)
The Lispkit Manual,
Programming Research Group, University of Oxford.

HENDERSON, P. MINKOWITZ, C. (1986)
The me too Method of Software Design,
ICL Technical Journal, Vol. 5, No. 2, pp. 64-95.

HENNESSY, J., JOUPPI, N., BASKETT, F., GILL, J. (1983)
MIPS, a VLSI Processor Architecture,
Computer Systems Laboratory, Stanford University.

HEWITT, C., BAKER, H. (1977)
Laws for Communicating Parallel Processes,
Proceedings of the IFIP Congress, pp. 987-992.

HIBBARD, P.G., SCHUMAN, S.A. (eds) (1978)
Proceedings of the IFIP Working Conference on
Constructing Quality Software.

HOARE, C.A.R., WIRTH, N. (1973)
An Axiomatic Definition of the Programming Language Pascal,
Acta Informatica, Vol. 2, pp. 335-355.

HOARE, C.A.R. (1985)
Communicating Sequential Processes,
London: Prentice-Hall International.

HOARE, C.A.R., SHEPHERDSON, J.C. (eds) (1985)
Mathematical Logic and Programming Languages,
London: Prentice-Hall International.

HOPKINS, R.P. (1984)
General Purpose Decentralised Computer Architecture,
Ph.D. Thesis, University of Newcastle upon Tyne.

INGALLS, D. (1978)
The Smalltalk-76 Programming System Design and Implementation,
Proceedings of the Fifth Conference of the Principles of
Programming Languages, pp. 9-15.

INMOS, (1984)
IMS T424 Transputer Reference Manual.

INTERNATIONAL BUSINESS MACHINES (1987)
IBM 370/XA, Principles of Operation.

JONES, C.B. (1986)
Systematic Software Development Using VDM,
London: Prentice-Hall International.

KAHN, G., MACQUEEN, D.B. (1977)
Coroutines and Networks of Parallel Processes,
Proceedings of the IFIP Congress.

KATZ, D.G. (1984)
A Machine Organisation for Combining Control Flow, Data Flow,
and Demand Flow,
M.Sc. Dissertation, University of Newcastle upon Tyne.

KERR, R. (1987)
A Materialistic View of the Software "Engineering" Analogy,
SIGPLAN Notices, Vol. 22, No. 3, pp. 123-125.

KNUTH, D.E. (1973)
The Art of Computer Programming, Sorting and Searching,
Addison-Wesley.

KNUTH, D.E., PARDO, L.T. (1976)
The Early Development of Programming Languages,
Digital Systems Laboratory, Stanford University.

LANDIN, P.J. (1964)
The Mechanical Evaluation of Expressions,
Computer Journal, Vol. 6, No. 4, pp. 308-320.

LANDIN, P.J. (1965)
A Correspondence Between Algol 60 and Church's Lambda Notation,
Communications of the ACM, Vol. 8, No. 2, pp. 89-101, and
Vol. 8, No. 3, pp. 158-165.

LANDIN, P.J. (1965)
An Abstract Machine for the Designers of Computing Languages,
Proceedings of the IFIP Congress.

LISKOV, B.H., SNYDER, A., ATKINSON, R.R, SCHAFFERT, J.C. (1977)
Abstraction Mechanisms in CLU,
Communications of the ACM, Vol. 20, No. 8, pp 564-576.

LISKOV, B., MOSS, E., SCHAFFERT, C., SCHEIFLER, R.,
SNYDER, A. (1978)
The CLU Reference Manual,
Laboratory for Computer Science, MIT.

LONDON, R.L., SHAW, M., WULF, W.A. (1978)
Abstraction and Verification in Alphard, a Symbol Table
Example,
in [Hibbard and Schuman, 1978], pp. 319-351.

MCCARTHY, J., ABRAHAMS, P.W., EDWARDS, D.J., HART, T.P., LEVIN, M.I. (1962)
LISP 1.5 Programmer's Manual,
Cambridge, Ma.: MIT Press.

NAUR, P. (ed) (1963)
Revised Report on the Algorithmic Language Algol 60,
Computer Journal, Vol. 5, pp. 349-367.

PAKER, Y., VERJUS, J-P. (eds) (1983)
Distributed Computing Systems: Synchronisation, Control and Communication,
London: Academic Press.

PATTERSON, D.A., DITZEL, D.R. (1980)
The Case for the Reduced Instruction Set Computer,
Computer Architecture News, Vol. 8, No. 6, pp. 25-32.

PATTERSON, D.A., SEQUIN, C.H. (1981)
RISC 1, a Reduced Instruction Set VLSI Computer,
Proceedings of the Eighth International Symposium on Computer Architecture, pp. 443-457.

QUINE, W.V. (1974)
Methods of Logic,
Third Edition,
London: Routledge and Kegan Paul.

RANDELL, B. (1973)
The Origins of Digital Computers, Selected Papers,
Berlin: Springer-Verlag.

RANDELL, B. (1983)
The Structuring of Distributed Computing Systems,
Computing Laboratory, University of Newcastle upon Tyne.

ROBINSON, J.A. (1965)
A Machine-oriented Logic Based on the Resolution Principle,
Journal of the ACM, Vol. 12, No. 1, pp. 23-41.

SAMET, J.E. (1969)
Programming Languages, History and Fundamentals,
Prentice-Hall.

SCHLAGETER, G. (1978)
Process Synchronisation in Database Systems,
ACM Transactions on Database Systems, Vol. 3, No. 3, pp. 248-271.

STANKOVIC, J.A. (1982)
Software Communication Mechanisms: Procedure Calls Versus Messages,
Computer Magazine, April, pp. 19-25.

STOY, J.E., STRACHEY, C. (1972)
OS6 - An Experimental Operating System for a Small Computer,
Computer Journal, Vol. 15, No. 2, pp. 117-124, and Vol. 15, No. 3, pp. 195-203.

SUSSMAN, G.J. (1982)
LISP, Programming and Implementation,
in [Darlington, Henderson, and Turner, 1982], pp. 29-71.

TRELEAVEN, P.C., HOPKINS, R.P. (1981)
Decentralised Computation,
Proceedings of the Eighth International Symposium on Computer Architecture, pp. 279-290.

TRELEAVEN, P.C., HOPKINS, R.P. (1981)
A Recursive (VLSI) Computer Architecture,
Technical Report 161, Computing Laboratory,
University of Newcastle upon Tyne.

TRELEAVEN, P.C., HOPKINS, R.P. (1982)
A Recursive Computer Architecture for VLSI,
Proceedings of the Ninth International Symposium on Computer Architecture, pp. 229-238..

TURNER, D.A. (1976)
SASL Language Manual,
University of St. Andrews.

TURNER, D.A. (1982)
Recursion Equations as a Programming Language,
in [Darlington, Henderson, and Turner, 1982], pp. 1-28.

TURNER, D.A. (1985)
Functional Programs as Executable Specifications,
in [Hoare and Shepherdson, 1985], pp. 29-50.

TURNER, D.A. (1986)
An Overview of Miranda,
SIGPLAN Notices, Vol. 21, No. 12, pp. 158-166.

UCHIDA, S. (1982)
Towards a New Generation Computer Architecture.
Technical Report, ICOT, Japan.

WADGE, W.W., ASHCROFT, E.A. (1983)
Why Lucid?
University of Warwick.

WILLEY, E.L., d'AGAPEYEFF, A., TRIBE, M., GIBBENS, B.J., CLARKE, M. (1961)
Some Commercial Autocodes, a Comparative Study,
London: Academic Press.

WILLIAMS, J.H. (1982)
Notes on the FP Style of Functional Programming,
in [Darlington, Henderson, and Turner, 1982], pp. 75-101.

WILNER, W. (1980)
Recursive Machines,
Palo Alto Research Centre, XEROX Corporation.

WULF, W.A., LONDON, R.L., SHAW, M. (1976)
An Introduction to the Construction and Verification of Alphard Programs,
IEEE Transactions on Software Engineering, Vol. 2, No. 4,
pp. 253-265.