NEWCASTLE UNIVERSITY LIBRARY

M 5

084 10628 7

Thesis L2868

Recursive Structures

in

Computer Systems

David R. Brownbridge

Ph.D. Thesis September 1984

Computing Laboratory,

University of Newcastle upon Tyne.

Abstract

Structure plays a important part in the design of large systems. Unstructured programs are difficult to design or test and good structure has been recognized as essential to all but the smallest programs. Similarly, concurrently executing computers must co-operate in a structured way if an uncontrolled growth in complexity is to be avoided. The thesis presented here is that recursive structure can be used to organize and simplify large programs and highly parallel computers.

In programming, naming concerns the way names are used to identify objects. Various naming schemes are examined including 'block structured' and 'pathname' naming. A new scheme is presented as a synthesis of these two combining most of their advantages. Recursively structured naming is shown to be an advantage when programs are to be de-composed or combined to an arbitrary degree. Also, a contribution to the UNIX United/Newcastle Connection distributed operating system design is described. This shows how recursive naming was used in a practical system.

Computation concerns the progress of execution in a computer. A distinction is made between control driven computation where the programmer has explicit control over sequencing and data driven or demand driven computation where sequencing is implicit. It is shown that recursively structured computation has attractive locality properties.

The definition of a recursive structure may itself be cyclic (self-referencing). A new resource management ('garbage collection') algorithm is presented which can manage cyclic structures without costs proportional to the system size. The scheme is an extension of 'reference counting'.

Finally the need for structure in program and computer design and the advantages of recursive structure are discussed.

Contents

Abstract (ii)

Contents (iii)

Acknowledgements (vi)

Dedication (vii)

Chapter 1 - Introduction 1

Chapter 2 - Recursive Structure 9

2.1 Modelling: Form vs. Content 9
2.2 Structure and Hierarchy 11
2.3 Recursive Structure 16

Chapter 3 - Multi-Level Computer Systems 19

3.1 The Model 20
3.2 The Language 23
Control and Data 25
Notation vs. Specification 25
Properties of Programming Languages as Specifications 26
3.3 Multi-Level Systems
Combining (C1+C2) 31
Stacking (P2/C1) 32
Interpreter Extension (P2↑C1) 33
Recursive Interpreter Extensions 35

Chapter 4 - Naming and Recursive Naming 38

4.1 Abstraction: Naming, Addressing and Routing 40
4.2 Classification/Analysis Bindings, Contexts, Resolving 43
Resolving a Name in a Context 44
Multiple Contexts 47
Resolving Names in Block Structured Contexts 50
'Block Structured' Naming 52
Pathname Naming 54
Alternative Naming Mechanisms 57
(1) Hybrid Naming Schemes 57
(2) Other Mechanisms 58
(a) Combined (Pathname and Block Structured) Naming 58
(b) Path-List naming 59
(c) Recursive Pathnames 60
Implications of the Naming Mechanisms 61
Static Resolving vs. Dynamic Resolving 66
Modularity vs. Textual Substitution 68

4.3 Recursive Naming 71
Recursive Contexts 72
Recursive Resolvers 74
Some Examples of Recursive Naming 75
Combining Contexts 75
Combining 'Flat' Naming Systems 76

Chapter 5 - UNIX United and the Newcastle Connection 82

5.1 Naming in UNIX 84
Kernel Naming Facilities 84
Utility Level Naming 87
Shell Level Naming 89
5.2 UNIX United 91
File Naming in UNIX United 92
Process Naming in UNIX United 95
UNIX United Sub-Systems 96
5.3 The Newcastle Connection 98
Choice of Level 98
Implementing the Newcastle Connection 99
File Names 100
Process Names 101
5.4 Some Other Distributed UNIX Systems 103
Review 105
Why Not UNIX? 106

Chapter 6 - Computation and Recursive Computation 109

6.1 Abstraction: Computation, Architecture, Hardware 110
Computation Organization 111
Program Organization 112
Machine Organization 113
6.2 Classification/Analysis: Computation Organization 114
Control Driven Computation 114
Data Driven Computation 116
Demand Driven Computation 117
Comparison of Data Driven and Demand Driven Computation 118
6.3 Recursive Computation 120
Recursive Program Structure 120
Recursive Control 121
Combining Programs 122
Some Examples of Recursive Computation 123

Chapter 7 - Recursive Computer Systems 126

7.1 Future Operating Systems 127
On the Concept of 'A UNIX System' 127
Distributed Operating Systems: Future Work 130
7.2 Earlier Work on Recursive Systems 135
'Recursive Machines and Computing Technology' 135
DDM/1 135
Wilner's Recursive Machine 137
Recursive Control Flow 138

7.3 Object Management in Recursive Systems 139
A Brief Survey of Object Management Techniques 140
The System Model 144
Reference Counting Object Management 143
The Standard Reference Count Algorithm 144
Strong and Weak Pointers 145
An Essential Implementation Trick 155

Chapter 8 - Conclusion 157

8.1 Summary
8.2 Future Research

Appendix - Object Management in Combinator Graph Reduction 163

References 168

Acknowledgements

I would like to thank Prof. Brian Randell and Dr. Philip Treleaven for providing a lively research environment and for their advice and encouragement throughout the course of this work. I am also grateful to the many other people who have aided and assisted me during my time at Newcastle.

Financial support for this work was provided by the Science and Engineering Research Council of Great Britain.

Dedication

To my Grandfather, R.E. Farra (1888-1984)

'Age is a quality of mind'.

Chapter 1

Introduction

> Ambitiously and successfully, interweaving the making of a film (of The French Lieutenants Woman) within the film, ... Meryl Streep takes the part of the title character Sarah Woodruff, and the actress who plays her.
>
> [Review of "The French Lieutenants Woman"]

A key problem in Computer Science is managing the complexity of computer systems. Rapid advances in computer manufacturing techniques have enabled a continuing rapid growth in the power of computers. Computers are built containing more and more elements, operating at higher and higher speeds. Not only is the power of the most expensive computer increasing dramatically, but also the cost of the cheapest useful computer is decreasing in a similar way. Manufacturing ability is the driving force of this monumental success. The scientific problem is to organize ever-increasing resources so as to achieve useful work without becoming ensnared in ever-increasing complexity.

The perplexing intricacy of large systems has been widely recognized. H.A. Simon's essay 'The Architecture of Complexity' [Simon 1969] examines the notion of system structure at a very general level and draws examples from a wide range of physical, social and biological fields. Simon's ideas are significant, because he recognizes that the structure and the complexity of a system can be examined independently of the nature of system components. In other words, any system or

organization can be viewed as a collection of abstract objects; the study of complexity is then only concerned with grouping of objects and how they interact with each other. Simon declines to define the 'complexity' of a system formally, but his informal definition is simple and worth quoting:

> 'Roughly by a complex system I mean one made up of a large number of parts that interact in a nonsimple way. In such a system, the whole is more than the sum of the parts, not in an ultimate, metaphysical sense, but in the important pragmatic sense that, given the properties of the parts and their laws of interaction, it is not a trivial matter to infer the properties of the whole.' [Simon 1969,page 86]

The work in this thesis covers many aspects of computer system design including programming languages, operating systems and computer architecture. In each of these areas, complexity in the past has caused problems in the construction and use of systems. Many systems take an excessive time to build and then fail to fulfill their intended purpose. As time goes by, these problems are made worse: 'raw materials' become more and more abundant, tempting the construction of bigger and bigger systems.

Programming languages and the associated production of large programs is one area where complexity has caused problems. Advances in communications technology have led to the concept of a 'network operating system'[Bochmann 1983, Lampson et al. 1981] yet more complex than the ordinary variety. Advances in processor, storage and communication technology have led to highly parallel (VLSI, Fifth Generation) computer architectures which embody many individual computing elements[JIPDEC 1982].

Not until the late 1960s was it generally realised - after many failures - how difficult it is to write computer programs. The intricate patterns of control created using the **goto** statement had made it difficult to understand a program except as a monolithic whole. A new era of `structured programming'[Dahl et al. 1972] began once the consequences of complexity in programs had been recognized. Commenting on an influential conference[Buxton and Randell 1969] Gries writes:

> `The complexity and size of [software] projects increased tremendously in the 1960s, without commensurate increases in the tools and abilities of the programmers; the result was missed deadlines, cost overruns and unreliable software . . .
>
> . . . For the first time, a consensus emerged that there really was a software crisis, that programming was not very well understood.' [Gries 1981,page 296]

It is significant that Gries' book, published more than ten years after that conference, can only begin to solve the problems of programming. The major change in the 1970s was not so much an improvement in programming standards but a recognition that programming is not easy, that better tools and concepts are needed.

An operating system is a large program in its own right and subject to problems of (internal) complexity which are common in large programs:

> `All repairs tend to destroy the structure, to increase the entropy and disorder of the system. Less and less effort is spent on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well ordered. Sooner or later the fixing ceases to gain any ground. Each forward step is matched by a backward one.' [Brooks 1975,page 122]

A way of tackling this internal complexity is to introduce structure within the program by constructing it as a hierarchy of separate 'layers', as suggested by Dijkstra[Dijkstra 1968b]. By formalizing the interface between layers, each can be designed and tested separately, with a good chance of the final composite system working correctly. By introducing structure the complexity of the system is reduced.

Another problem with operating systems is 'external' complexity. Quite separate from the internal complexity of the operating system as a program, there is the complexity of the facilities it provides - a tool must be easy to use, no matter how well it is constructed. When viewed as a whole, many operating systems provide a baroque, unstructured collection of facilities. A common characteristic is that each new external facility added to the system interferes with existing facilities, increasing the external complexity.

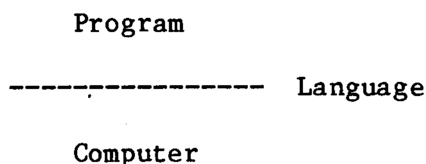
In the domain of computer hardware, electronic circuits were, until recently, designed by a skilled engineer who would organize the pattern of elements to ensure correct operation, the best path for signals and so on. With the continuing growth in the number of basic elements (gates) per circuit the human effort needed in design becomes the dominating cost in production. It becomes increasingly important to deal with such complex systems by structuring their design, whether the design process is manual or automatic:

> 'The process of designing a large-scale integrated system is sufficiently complex that only by adopting some type of regular, structured design methodology can one have hope that the resulting system will function correctly and not require a large number of redesign iterations.' [Mead and Conway 1980,page 60]

One way to write a thesis about structure and complexity would be to devote chapters to various problem areas: operating system complexity, programming complexity, VLSI complexity etc. Each subject has a large body of literature and could be treated in depth. Operating system complexity might be concerned with processes and file systems, programming complexity with abstract data types and modules, VLSI complexity with component layout and interconnection. In each chapter, important issues in each subject could be discussed in depth. But to deal with complexity in some other area such as data bases or communication networks, a separate new chapter would be necessary, because all the arguments about structure and complexity would have been restricted to specific applications.

In this thesis an alternative approach is taken, concentrating on _issues_ rather than _applications_. Two issues, naming and computation, are explored, with emphasis on the role they play in the structure of computer systems. Naming is concerned with identifiers in programs and how they are related to objects. Computation is concerned with the pattern of execution in a computer.

A _computer system_ will be viewed as a _computer_ plus the _program_ it executes.



The interface between the two is a programming _language_. Historically this interface has been the boundary between 'hard' and 'soft' parts of the system, although this is no longer true: 'multi-level' systems are built as a succession of layers, each acting as 'computer' for the one above.

A name is a token or identifier used in a program to stand for some object. The name of an object can be chosen arbitrarily, independent of the object's form, what it does, where it is stored and how it is accessed. Certainly, these attributes often constrain names in real systems, but only by convention. The power of naming is that a name can be chosen and used independently of the named object.

A contribution of this thesis in this area is a systematic synthesis of ideas on naming, brought together under the general theme of `structured systems'. Some of these ideas are taken from the work on names in programming languages[Barron 1977, Stoy 1977], adapted to the wider frame whilst others are new. The aim is to work towards a uniform theory of naming, independent of the kind of system in which names are being used and to describe `recursive naming'.

A major contribution of this thesis is the `UNIX United'/`Newcastle Connection' system[Brownbridge et al. 1982]. The system uses recursive naming to provide a homogeneous environment over multiple connected UNIX systems. In the description (Chapter 5) the emphasis is on naming and structure, describing a personal contribution to a joint project.

When a computer executes a program, control passes from one part of the program to the next according to a `computation rule'. In the majority of computer systems, control threads a single instruction-at-a-time path through the program and objects are held in a single store. In parallel or distributed systems, there are many threads of control forming an overall computation structure and many separate stores, with sharing and replication. This thesis examines the notion of computation, classifies computation mechanisms and describes `recursive computation'. This is a development of a contribution to a published survey of novel computer architectures[Treleaven et al. 1982].

Recursive computer systems can embody both recursive naming and recursive computation. Recursive naming makes it possible to combine programs to produce larger programs; recursive computation makes it possible for computers to co-operate in executing programs. Such systems are important if complexity is to be managed in the future.

Only a small number of published works have dealt explicitly with the use of recursion in system design. The work of Simon[Simon 1969] recognizing the hierarchical structure of many natural systems has already been mentioned. That work, together with C. Alexander's[Alexander 1964] show an awareness of design and structure as valid topics for scientific study. The arguments in favour of systematic design of buildings and towns[Alexander 1964] in the face of architectural complexity are very similar to those in favour of systematic programming in the face of complex computer systems[Hoare 1980].

The overall aim of this thesis is to show how structure, in particular 'recursive structure', can be used to deal with complexity in computer systems. Chapter 2 introduces the notions of modelling a system, structure and recursion. In Chapter 3 an abstract model of computer systems is introduced which show how naming and computation can be studied separately and how systems can be built from multiple layers. Chapter 4 discusses naming and recursive naming and Chapter 5 describes the UNIX United/Newcastle Connection system which is based on recursive naming. Chapter 6 examines computation: control driven, data driven and demand driven computation are described and then recursive computation is considered. Chapter 7 discusses the implications of recursive structure in computer systems' design, gives a brief survey of earlier work on recursive systems and introduces a new way of managing ('garbage collecting') structured objects. Finally, conclusions and directions for further research are presented.

An appendix describes a way of using the new object management algorithm in combinator graph reduction machines.

Chapter 2

Recursive Structure

> To abstract is to separate the qualities common to all individuals of a group from the peculiarities of each.
> [Jevons, Elementary Logic, 1870]

This chapter aims to explain what is meant by a recursively structured system and to clarify concepts of system model, structure and recursion. These are the foundation of the succeeding studies of recursive naming and recursive computation.

Recursive structures are a general form of hierarchical structure. They are a nesting of similar parts: any simple (atomic) element in a recursive structure can be replaced by a sub-structure. Combining recursive structures produces another recursive structure of the same kind.

2.1 Modelling: Form vs. Content

A model is an abstraction of a system. Using models it is possible to concentrate on the structure (form) of systems independent of the details of their content.

A model abstracts by hiding details and focusing on particular aspect(s). There are many possible models for a given system, corresponding to the many possible levels of detail and choice of

aspect. Given a system, a model can be constructed which exposes the structural relationship between the parts and hides the precise nature of those parts. This is exactly the technique used in text-books which treat data structures independently of the particular data being structured. In this chapter an attempt is made to examine pure structure as an abstraction of 'naming structure' and 'computation structure' - topics which are dealt with separately in Chapters 4 and 6 respectively.

The act of making a model is one of taking a system and producing a related 'model' system. Two factors must be taken into account:

(1) The aspect(s) of interest ('logical concerns')

(2) The level of detail ('atomicity')

Together they determine the relationship between model and 'original' system e.g. 'performance model of x' or 'structural model of x'. They define what kind of abstraction is being made of the system.

The aspect of interest controls which parts of the system are given prominence in the model: a repair engineer's model of a computer is very different from that of a programmer. One model is embodied in wiring diagrams, the other in the machine language specification. The models are consistent with each other, but reflect a different emphasis whilst describing the same thing. Although a model can encompass many aspects of a system at once, it is often useful to use separate models for each aspect of interest. By concentrating on one issue at a time, although aware 'in the background' of the others, the problem of designing or understanding systems is greatly reduced. E.W. Dijkstra has termed this method 'separation of concerns'[Dijkstra 1982].

The level of detail in a model can be chosen independently of the aspect of interest, whether one or many aspects of the system are incor-

porated into the model. The level of detail defines just how accurately the model reflects the chosen aspects of the original system. A finely detailed model gives a complete description of the system; a less detailed model may be more appropriate as it will be simpler. A level of detail is defined by a set of atomic objects which form its 'base'. Atoms are defined to be the most detailed objects in the model, they are always seen as whole entities with no internal structure. The level of detail will sometimes be referred to as the 'level of atomicity'.

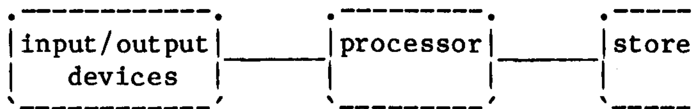
It is as well to be explicit about these two features, aspect and detail, which characterize models, although it is by no means novel to recognize their existence. Often the structured approach to system design runs into problems when only a single structural model is used. It must be emphasized that a system can concurrently have many models, each with a particular aim and structure. In some cases the relationship between models can be given formally; in particular it might be shown formally that two models indeed correspond to a single given system.

2.2 Structure and Hierarchy

Structure is concerned more with the arrangement of parts and their inter-relationships than the actual nature of the parts themselves. A major theme of this thesis is the importance of structure in computer systems. In particular, when many systems are to be combined into a distributed computer system, structure is as important as the design of the component parts.

Given a system, or equivalently, a model of a system, what does it mean to impose structure on it? First of all, there may already be structure apparent in the system itself. For example the following model of a computer system for many years corresponded to the physical

reality of machine room layout:



Sometimes, this `natural' structure will be enough. However in general, the problem of imposing structure on a complex system has to be faced. There are two complementary ways of describing this process of `adding structure'. On the one hand, it may involve grouping or aggregation, combining many simple elements into a structural element. On the other hand, it may take the form of decomposition or separation, dividing an amorphous system into a structured arrangement of elements.

Whichever description is used, the process is basically the same: a complex system is transformed into a simpler one by imposing structure; the only difference is the starting point. (No doubt there is some connection here with the Philosophies of Holism and Reductionism, but this will not be pursued here.)

Example

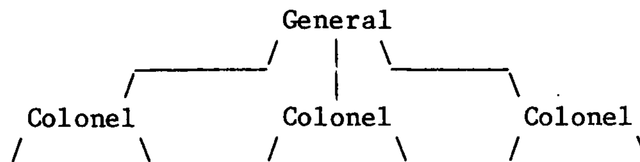
`Synthesised' or `Bottom-Up' Approach:

Solving a jig-saw puzzle reveals a hidden picture - starting from a complex arrangement of pieces, the structured whole is gradually revealed.

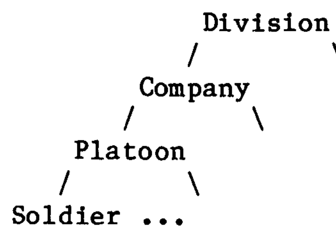
`Analytic' or `Top-Down' Approach:

An ant-hill is an ordered society to the insect-expert, but a teeming chaos to the novice - starting from the complex mass, the expert imposes a `social hierarchy' to achieve understanding.

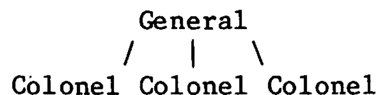
Structure may be imposed on a set of elements, either by arranging the elements in a structure, for example soldiers in a hierarchy of rank:



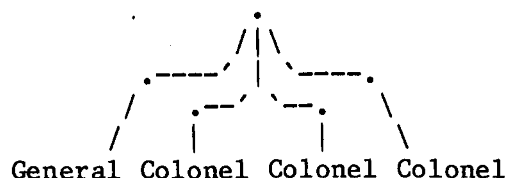
or, by adding structure, inventing new elements, such as the organizational sections of an army:



There is no hard distinction: for example both bottom-up and top-down parsers add structure onto the original program text. Finally, all these issues can be confused by representation, because



can be re-drawn as:



with anonymous structure elements and the highest 'rank' to the left. This representational difference is only important when it affects our way of thinking about the system in question. For example, in LISP systems all structure is built from CONS cells, information resides at the

'leaves'; in data base systems structure is built from many different kinds of 'records' which themselves contain information. In LISP one has a strong separation of structural objects (CONS cells) from data containing objects (ATOMS), whereas in databases structure and data are intermingled.

Hierarchy

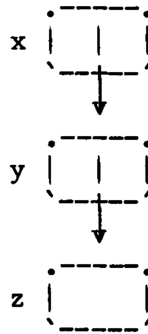
The most general structure, sometimes called a network, imposes no constraints on the relationship between parts of a system. Because we are interested in ways of simplifying systems, it is necessary to consider less general structures where the relationship between parts is more regimented. Hierarchical structure is of particular importance as it models 'encapsulation' of lower-level objects by those above. In a hierarchy there is a sense of 'above' and 'below'; lower elements are nested inside the element(s) above them.

The structure of a system can be modelled by an ordered pair (N,A). Let N denote the set of objects (nodes) in the system and A denote the structural relationship (arcs) between them; A is some subset of $(N \times N)$ (the set of pairs (n1,n2), such that n1 and n2 are members of N). For example the system (P,Q) where

$$P = \{x,y,z\}$$

$$Q = \{(x,y),(y,z)\}$$

can be represented as:



A network is a structure (N,A) with no restriction (on A) as to the relationship between the objects in N. A network can model any form of structural relationship. For example if a system allows new objects to be created and arbitrary pointers between them to be made then there is no way of restricting the structure to any hierarchical form. It is even possible to create cyclic structures where sequences of pairs from A form a loop:

$$\langle (a,b),(b,c),(c,d), \ldots (y,z),(z,a) \rangle$$

Such structures are regarded as paradoxical as system models: in the example object `a' is an abstraction of itself!

A hierarchy is a structure (N,A) where the relationship between elements is restricted so that any one element can be regarded as an `abstraction of' or `abbreviation for' the elements below it. Parnas defined hierarchical structure in terms of the levels of abstraction it can give rise to[Parnas 1974]. Here it will be defined solely as a partial ordering of elements. A system (N,A) will be said to be hierarchically structured if the relation A satisfies the following properties:

(1) No direct self-reference:

(x,x) is not in A (for all x in N).

(2) No cycles:

If reachable(x,y) then NOT(reachable(y,x))

Where `reachable(a,z)' is true if there is a path from `a' to `z' consisting of elements of A; i.e. a sequence

< (a,b),(b,c),(c,d), ... ,(x,y),(y,z) >

(By considering the empty sequence, `a' is reachable from `a')

(3) Encapsulation:

Given any two objects x and y in N, there exists some z which encapsulates them both; that is for every x and y in N there is a z such that:

reachable(z,x) and reachable(z,y)

2.3 Recursive Structure

A recursive structure is a hierarchy in which the elements are all similar. At every node in a recursive structure, the relationship to the surrounding structure is the same. This is particularly useful for modelling logical systems (e.g. programs) because recursive structures can be nested to an arbitrary depth whereas in most hierarchies there is a fixed `root'.

One property of recursive structures is that combining recursive structures produces another recursive structure of the same type. This is sometimes called `scale independence'[Scarrot 1982]. This property is especially important in computer systems where it is essential to be able to build new objects from old and then repeat the process to an arbitrary degree[Barron 1968].

Another feature of a recursive structure is the absence of a distinguished 'root' of the hierarchy. A root is a distinguished base element containing all others. There is always a de facto root, which is not nested in other items, but it is in no way distinguished from the other items. As time passes the de facto root may move as a structure becomes incorporated into others. Absence of a 'root' makes it easier to combine systems without first altering them. Many non-recursive hierarchic systems distinguish a special type of root element which may never be nested.

For example, the programming language Pascal does not have recursive structure: a program is the largest possible unit. Programs may never appear nested in any other Pascal unit or be built by combining other programs[Jensen and Wirth 1978]. In practice this is a significant problem with Pascal: extra facilities must be added to enable programs to be combined.

* * * * *

To recap this chapter, a viewpoint and level of detail are embodied in the model of a system. They may be implicitly or explicitly chosen, but always have a profound effect on the resulting model. A structured view of the system is achieved by aggregating parts of the model, thus reducing its complexity. In a hierarchic system, lower elements are 'encapsulated' by those above. Hierarchies of similar elements are called 'recursive structures'. At each point in a recursive structure the environment is similar; a combination of recursive structures is also a recursive structure.

As will be discussed in Chapter 4, recursive structure in naming makes it easy to combine programs, no limit is imposed on the depth of nesting (program within program) and identical naming mechanisms are

used for intra-program and inter-program naming.

Similarly in Chapter 6 it is shown that recursive structure in computation makes it straightforward for many computers to execute a program concurrently yet still giving the illusion that a single computer is in use; there is also no detectable difference between multiprogramming and multi-processing.

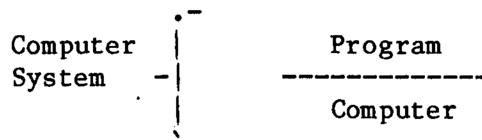
Chapter 3

Multi-Level Computer Systems

> But programming, when stripped of all its circumstantial irrelevancies, boils down to nothing more than very effective thinking so as to avoid unmastered complexity, to a very vigorous separation of your many different concerns.
> [Dijkstra 1982,p163]

This chapter introduces a model of computer systems which will be used to demonstrate the distinction between naming and computation. Within the model, naming and computation are separate concerns. The structure of naming and the structure of computation will then be examined separately in the succeeding chapters.

In the model, a computer system comprises two parts: (i) a program and (ii) a computer.



The program contains instructions and data items which can be interpreted by a computer to produce results. Programs will be viewed as textual objects which are manipulated by the computer. The computer will be viewed as a program interpreter which brings about changes to the state of the system. The precise distinction between control and data is discussed below.

Throughout what follows, the words computer and interpreter (or program interpreter) will be used as synonyms. They formally refer to the same thing, but quirks such as `Fortran-computer' and `IBM S/370 interpreter' are avoided. Also, I specify language to have the technical sense of programming language, as introduced above (and refined later), not the usual sense of `form of speech used by a people'[OED 1941]. In particular, I do not restrict `language' to mean `form of communication between man and computer' but include forms of communication between parts of a computer system; for example network protocols or job control languages.

3.1 The Model

In the previous chapter, structure, hierarchy and recursion were examined independently of any application they might have in computer system design; in this chapter I examine the notion of a `computer system' as a set of layers, without considering how each layer is structured within itself. In particular it will be important to see how separate components - each with internal design and structure - can be combined to form a larger aggregate system.

The separation of computer systems into two parts is an essential feature of the model, separating static and dynamic parts. Within this philosophy one may talk of the computer causing changes to the state of the program, but not of the program causing a given effect on the computer. Only execution of programs by computers causes an effect. The model of a static program plus dynamic interpreter is loosely based on the work of Jones[Jones 1981]. It is also related to other work on descriptions of system structure[Anderson and Lee 1981, LaPrie 1983, Parnas 1974].

A program is a piece of text consisting of instructions and data, a

sequence of symbols stored in the computer. The computer is a program-interpreting mechanism. Naming is concerned with how resources of the computer are used in programs (viâ the programming language). Computation is concerned with how programs are executed: the sequencing of execution and the range of effect of that execution.

The computer/program boundary may or may not correspond to the physical boundary of a machine. The boundary between 'hard' and 'soft' parts of the computer has become increasingly blurred by the widespread use of microprogramming, virtual memory and virtual machine techniques. In the latter case, a software program is implementing something which is to be regarded as the 'hard' machine. Similarly, there is a matching trend towards putting essential software into the hardware in the form of Read Only Memory (ROM). Here what seems to be soft is in fact hard!

In the model, the notion of 'program' is very general, encompassing any computable expression. A program is simply the description of any computable expression given in a programming language. (Notions of computability are defined in the literature e.g.[Brady 1977, Malitz 1979] and will not be explored here.) A suitable computer (one that implements the appropriate language), can execute the program, giving rise to some pattern of computation.

Similarly the notion of 'computer' in the model is very general, a computer is any programmable mechanism: anything for which a programming language can be defined. A computer is simply an interpreter (or execution mechanism) for some explicit or implicit programming language. Although a computer is a 'stored program' device, it need not be restricted to a single program. Execution of one program (e.g. a scheduler or a compiler) may cause another program to be executed. The least that is required is that some initial 'bootstrap' program be present when the machine is switched on.

The fact that program and computer are kept independent has obvious important implications. A program can be used on any computer which provides the right language. A program is independent of how the computer which executes it is constructed. A computer can of course execute any program written in the language which it interprets, not just a fixed program.

An example of separation is that in commercial computer hardware there is often a 'range' of machines, each implementing the same machine language. Machines in the range have very different constructions and prices but all provide the same language. Here the separation is made between language - the 'machine architecture' - and computer which is the hardware implementation of that architecture. Another example of separation of architecture from implementation is the concept of 'data independence' in database systems, where the conceptual view of data is independent of the storage method[Date 1975].

It is worth noting that nearly all the computer systems which rigorously separate language from interpreter do so by hardware. Few software systems are sufficiently disciplined or well designed to allow or need no dependence on the underlying implementation. Often separation is sacrificed in the quest for efficient execution, invariably tying programs to a particular implementation.

The model as outlined above bears a resemblance to 'module' constructs in languages such as Modula-2[Wirth 1983] and CLU[Liskov et al. 1970] (which are each ultimately based on the pioneering work in Simula[Birtwhistle et al. 1973]).

In Modula-2 for example, a module consists of two separate parts, an interface and an implementation. The interface describes the external behaviour of the module (i.e. the language it provides); the imple-

mentation describes how that behaviour is achieved using the constructs of Modula-2 including, recursively, modules. Because interface and implementation are separable, the implementation can be changed (programmed anew), without affecting the interface. In particular, other modules using a module are not affected if only the implementation and not the interface is changed.

Modula-2 provides fairly simple facilities for assembling complete systems from collections of modules: parts of an implementation cannot be shared by two interfaces, nor is there a way of extending an interface i.e. of providing extra facilities whilst leaving the original parts of the interface unchanged. The latter issue is related to the lack of module-combining operators to form aggregate interfaces. This will be discussed further in `Multi-Level Systems' (Section 3.3).

The language Mesa has an associated language, `C-Mesa', for building systems from Mesa modules[Mitchell et al. 1978]. In a truly general system, the higher level, system-forming language can be identical to the original language. Indeed, languages which are themselves `higher order' need no separate language to combine systems. Higher order functions can take functions as arguments and produce functions as results[Burge 1975]; similarly in a fully general `object oriented' language, `objects' can be sent in messages and returned in response to messages.

3.2 The Language

A programming language is the abstraction common to computer and program. Only if the languages of the program and of the computer match, will execution of the program by the computer be proper. Only computer systems where program and computer are properly paired will be considered in what follows. When the match is incorrect, the result of

execution is unspecified.

The language may be implicitly or explicitly defined. When a published language specification differs from an implementation it is inevitable that the implemented version of the language is the one used. Examples abound of extra undocumented 'invisible' operations or side-effects of operations, which subsequent implementations are forced to reproduce, simply because programs have used the actual interface not its specification. Indeed the idea that the actual interface is the final arbiter has on occasion been used as an argument against any formal specification of an interface. I would instead argue that the correct method is to expend as much effort as is necessary to ensure implementation and specification match exactly, with neither missing nor extra features.

In most of the programming languages in use today, the execution of a program must be specified with reference to some concepts not defined within the language itself. For example the current value of a variable may depend on the state of the run-time stack.

Backus has distinguished primary and secondary languages[Backus 1972]. In primary languages state transitions (computation) can be described as program transformations - each execution step transforms (or 'elaborates') a valid program into another valid program. In secondary languages the initial program is translated into an abstract machine state, and execution is successive changes of machine state.

Thus a execution of a primary language program can be explained without reference to external state. A secondary language program must become a process when it is executed. The process consists of the program text plus the current state. Processes are necessary not just for multi-programming but for execution of secondary language programs to

take place at all. Thus when we talk of executing such a program, what is often in fact being executed is a process consisting of the program plus 'run time system'.

Control and Data

In the present model, this distinction between control and data is only important below the interface, in the computer, where it may be useful to distinguish <u>instructions</u> from <u>data</u>, (even though a particular object may take either role from time to time). Note that by 'data' I mean values embedded in or calculated in a program, not values 'read in' as the program is executed. In terms of the language 'BASIC' data is 'DATA' not 'INPUT'! The latter, 'INPUT', is the case of a program acting as a (higher level) interpreter. Such multi-level systems will be considered in Section 3.3.

No claim is made that the control/data distinction is fundamentally wrong in programs, it is just much simpler to disregard it in most of the present study - both commands and data are <u>data</u> to the interpreter.

Notation vs. Specification

In the design and study of programming languages one must be aware that there are two aspects to their use:

* Notation - as an aid to thought, simply 'getting it down on paper'[Iverson 1980]. Major concerns: ease of use by <u>people</u>, expressiveness, brevity[Whitehead 1911].

* Specification - as a means of discipline when parts of a system interact, describing exactly what is to be computed. Major concerns: completeness, consistency.

Throughout the study of programming languages these two aspects are in

constant conflict. In particular the expressiveness of language is often in conflict with its consistency. Here, we will only consider programming languages as specifications - they characterize the relationship between program and computer. Issues of notational convenience and ease of use in programming as a human activity will not be considered. There are sufficient issues of completeness and generality to consider in connection with system structure.

The usual study of programming language design is biased rather more towards issues of notational convenience and ease of implementation[Barron 1977, Elson 1973, Ledgard and Marcotty 1981]. Here, completeness and consistency are the main concern.

Properties of Programming Languages as Specifications

* Soundness - there must be no way of using gratuitous implementation features; the implementation must exactly match the language specification.

* Consistency - a given object should have the same properties wherever in the program it is used.

* Completeness - generality; power to express all useful programs.

* Minimality - No feature of the language to be a combination of others.

Soundness is a very simple property: the computer must provide exactly the language as specified.

A sound implementation of a language will correctly evaluate all well-formed programs, but not any others. Soundness also implies a kind of atomicity. In a sound language it should not be possible to explore partial execution of basic (atomic) commands nor detect the internal

structure of basic (atomic) data items. (A sound implementation of an interpreter is analogous to the least fixed point of a recursion equation.)

Unfortunately many computer systems - especially those constructed from software rather than hardware - are not sound. It even seems unlikely that soundness was ever a major consideration whilst they were being built! Soundness is compromised in both directions: extra facilities, which are not part of the language, are provided, while some parts of the language are not implemented. This is usually because this was convenient to the implementor. In such a case the language is 'weak' because it does not enforce a total separation of language from the computer implementing the language.

Consistency is a regularity property which can be paraphrased as 'absence of special cases'. A consistent language is easier to use because there is a harmony between the parts of it: the language itself has a regular structure. Consistency of a language with itself is related to 'orthogonality'[Barron 1977] which is 'non-interference of concepts'.

Completeness is a second regularity property: 'ability to program arbitrary tasks' and 'ability to construct arbitrarily large programs'.

One aspect of completeness is that the language must provide a sufficiently rich set of constructs to span the space of programs (akin to being able to program all useful functions). The other aspect of completeness is that the language should provide combining forms - structuring operators - to combine groups of commands and/or data into compound objects (possibly named), so that these can be nested to arbitrary depths. The first aspect has been dealt with at great length elsewhere in the literature of computability theory. Grouping to an arbitrary

depth is the essence of recursive structure (see Chapter 2).

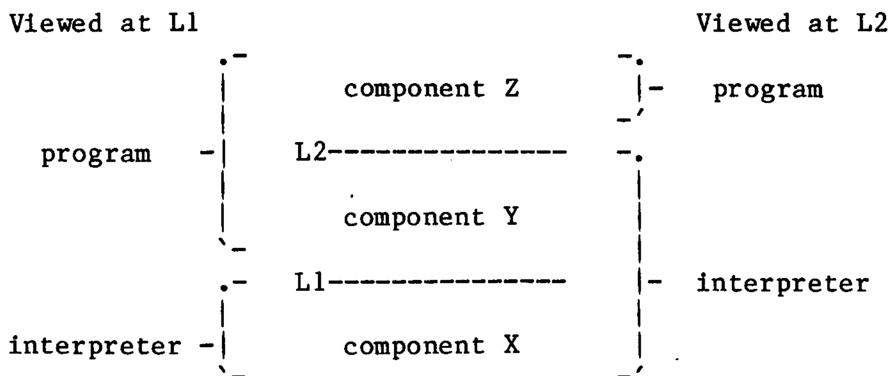
Minimality implies a sparseness of language features - no command or data provided by the language can be programmed in the language.

From the point of view of notation minimality may be less desirable, but a language specification is best simplified by minimizing it. The combination of minimality and completeness suggests languages should consist of a small number of basic objects together with a general way of combining them.

Each of the above properties - soundness, consistency, completeness and minimality - has been defined informally and briefly but I believe that together they form a good basis for language design and implementation from the point of view of specifying interpreters.

3.3 Multi-Level Systems

So far, a single level system has been considered, with one program, one language and one computer. For design and for understanding, it is essential to look at one level at a time, concentrating on the program, language and computer forming the computer system at that level.



A system can have many levels - many linguistic interfaces - and still be studied in this way.

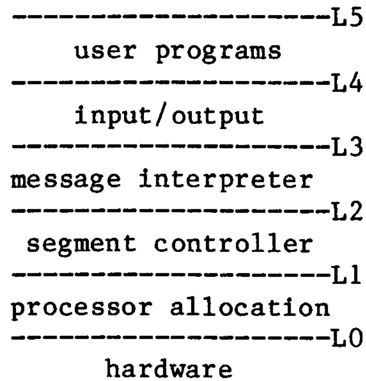
The present section will examine the structure of multi-level

systems in order to explain how systems can be combined to produce another composite system. The aim is to understand what happens when systems are combined - not to describe the art of choosing the layering. The main addition here is the notion of 'recursive interpreter extension': an extension to the system which is exactly compatible with the original (Tanenbaum uses the term 'self-virtualizing machine'[Tanenbaum 1984]). This is an abstraction of a design principle used in many systems: e.g. 'Virtual Memory'[Kilburn 1962] is a recursive extension of the address space; 'virtual machines' (e.g. VM/370) are a recursive extension of machine language making many separate 'machines' available concurrently.

One way to understand multi-level systems is to recall the concept of 'separation of concerns' (Chapter 2): let each level implement a particular concern. As each level is added to the system some new facility is provided.

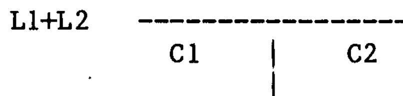
The idea of structuring a system using separate layers has been used extensively in the literature, most notably by Dijkstra in the T.H.E. operating system[Dijkstra 1968b]. In that system, the layering was based on a separation of concerns. Unfortunately, it is by no means clear that an arbitrary given system can be divided into separate layers, without considerable thought and effort on the part of the designers.

Consider the structure of the T.H.E. operating system:

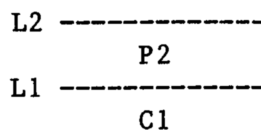


At each successive level, better abstractions are provided. At L1 processor allocation is hidden and so on up the hierarchy, each level providing a better 'virtual machine'. In other work, levels are seen as a consequence of the relationship between components, structure is not necessarily imposed explicitly by the designer[LaPrie 1983].

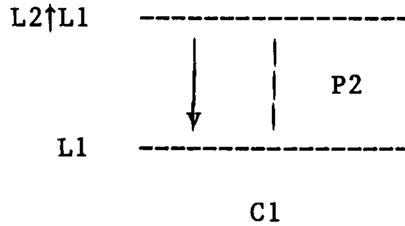
Three ways of building interpreters will be examined. The first is simply combination, merging two interpreters (C1,C2) to provide a single joint interpreter (C1+C2) with corresponding language (L1+L2):



The second is stacking one interpreter (P2) on top of the other (C1) giving system (P2/C1):

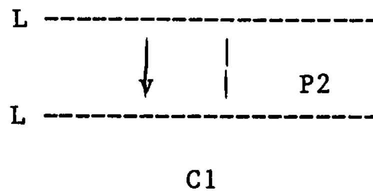


The third, interpreter extension, is more subtle - the second interpreter (P2) extends or modifies interpreter (C1) giving (P2↑C1):



The vertical arrow denotes that parts of a program may be interpreted directly by C1.

The final variant is the recursive interpreter extension where the same language is provided at the upper boundary but with some 'value added' (e.g. robustness, security) by the extension:



Combining (C1+C2)

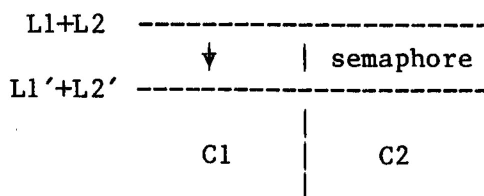
Combining two interpreters produces a interpreter supporting the 'sum' of the two component languages. The language of the combined computer contains all the languages of its components. For two systems to be combined they must be compatible: the resulting language (L1+L2) must be well defined. This means that either L1, L2 are disjoint, or it does not matter if some construct which is in both L1 and L2 is evaluated by either C1, C2 or both of them.

For example, when C2 provides a very simple addition to L1, it is straightforward to ensure that L1+L2 is well defined. When C1 is a minicomputer and C2 is a 'floating point unit' the languages are L1 = standard instruction set, L2 = floating point instructions; then C1+C2 is

well formed because C1, C2 do not interfere by both trying to execute the same instruction.

The crux of a combined system is the interaction between the two components; they must have been designed to prevent accidental interference, e.g. by using a small shared data area for control information.

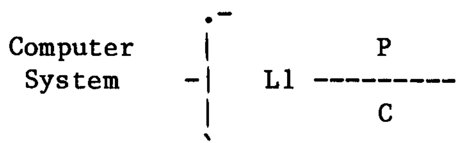
For example in the IBM S/370 there is a mode of multi-processing where two identical computers cooperate to share storage using special instructions.



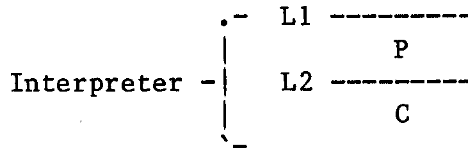
This is essentially a very 'thin' interpreter extension which is kept invisible. Languages L1',L2' are both the normal S/370 instruction set augmented by operations using the semaphore to avoid interference when accessing shared resources.

Stacking (P2/C1)

The idea is simple, but nevertheless worth describing explicitly for the purposes of comparison. Take a interpreter C and a program P and combine them to produce an aggregate interpreter. When P is itself an interpreter for language L1, the computer system



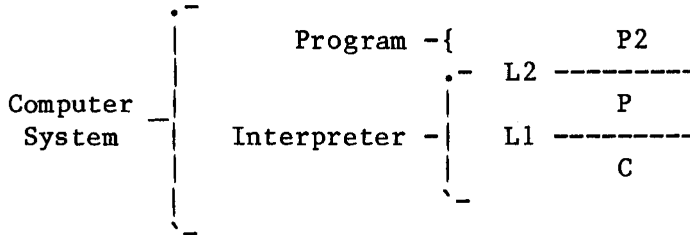
(denoted P/C and pronounced 'P over C') can be viewed at a higher level as a single interpreter implementing L1:



which can execute any program P2 in the language L2. Here the new language L2 is defined as the combination of an interpreter and a program:

L2 ::= P/C

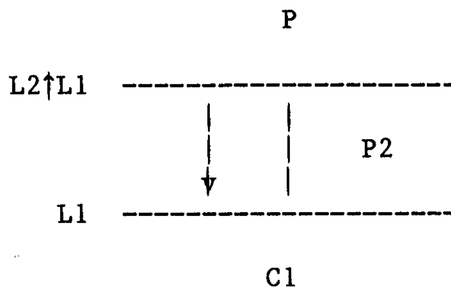
Programs in L2 can be executed in the usual way:



This common construct is a straightforward way of building systems as a set of layers. In general, L1 is not the same as L2 because the purpose of the extra layer (P) is to `add value' to language L1 by increasing the level of abstraction.

Interpreter Extension (P2↑C1)

Here, a program extends the language provided by an interpreter (A↑B is pronounced `A extending B'):

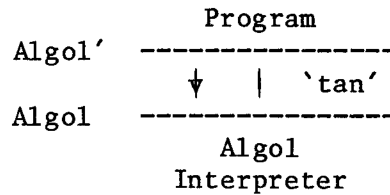


Computer C1 implements language L1; P2 is an interpreter for language L2 and is written in language L1. At the upper interface both L1 and L2 are available. The advantage of interpreter extension is that it adds to a system by extending it, not by hiding (as is the case with stacking). Both the added language and parts of the original language are made available. Viewed at L1, P2 is a program to be interpreted along with P. Viewed at L2↑L1, P2 is a part of the composite interpreter. During execution, C1 will sometimes be executing parts of P directly and at others will be executing P by means of P2. C1 interprets both P2 and the parts of P written in L1; P2 interprets the parts of P written in L2.

An interpreter extension is asymmetric - one interpreter is a program which extends the other. This resolves the problem of shared control and also of name clashes between L1, L2. Recall that in C1+C2 anything common to L1 and L2 had to be interpreted by either C1 or C2 or by both with well-defined results. With P2↑C1 it seems profitable to incorporate the following rule to avoid clashes when interpreting programs in the language implemented by P2↑C1.

> 'If something is in both L1 and L2 then the definition in L2 is used'.

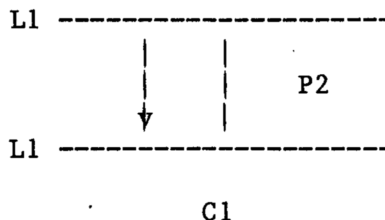
It is reasonable for constructs to be interpreted in the 'nearest' suitable interpreter in this way. This rule allows some part of L1 to appear, enhanced, under the same name in L2; or for parts of C1 to be deliberately hidden. An example may clarify such a situation: let P be an ALGOL program, C1 be an ALGOL interpreter which includes a trigonometric function 'tan' which will fail with argument '90'. The original 'tan' function can be embedded in an extension that first checks for argument '90' and then calls the original 'tan':



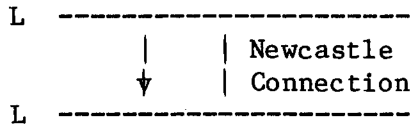
The new function can be known by the same name by using simple renaming. In effect the function is installed as an extension of standard ALGOL.

Recursive Interpreter Extensions

An interpreter extension P2↑C1 is recursive if the languages L2 and L1 provided by P2 and C1, are identical. (The term 'self-virtualizing' has often been applied to systems of this form[Tanenbaum 1984]). At first sight this might appear to be a nonsensical construct. In fact it has great value because a recursive interpreter extension is a transparent extension. Value is added to an interpreter without changing it: there is no detectable difference between original and extended systems as far as the programming is concerned. Intuitively the extension 'P2' takes part or all of L1 and in some way makes it better, without changing its appearance:



A recursive extension might take L1 and improve it by abstracting some of the unreliability of implementation C1. The Newcastle Connection (Chapter 5) is a recursive extension of UNIX



UNIX

where L is the set of UNIX `system calls' [Kernighan and McIlroy 1978]. The Newcastle Connection adds the ability to name objects on more than one UNIX system, increasing the name space without changing the set of commands (or data types).

Consider `transparency' defined as follows: an interpreter extension P2↑C1 is <u>transparent</u> (i.e. `P2 is a transparent extension of C1') if and only if any given program P which is executable by C1 is executable by P2↑C1 with identical result. That is, as far as is detectable from within a program P written in language L1, there is no difference between the two interpreters C1 and P2↑C1. Practical advantages of recursion are obvious: given C1 and P, P2 can be added at a later date without altering P.

All recursive interpreter extensions are transparent, but not all transparent interpreter extensions are recursive. For an extension to be transparent, language L1 must be provided at the upper (P2↑C1) interface; for an extension to be recursive, <u>only</u> language L1 must be provided at (P2↑C1).

Transparent non-recursive extensions are a kind of `natural extension' of L1. That is, existing programs can be run unchanged and will produce the same results, but also new programs can be written taking advantage of the extended interpreter.

* * * * *

To conclude, the theme of this chapter is the ability to deal with multi-level systems by taking one level at a time. All `above' the level is program, all `below' is interpreter. Execution of a program by an interpreter causes changes in the system. Multi-level systems can be built by combining programs and interpreters to build new interpreters. A recursive interpreter extension is a special case where no linguistic features are added but instead `value' is added in the form of better performance or access to a larger set of named objects.

Multi-level structure may play an important part in the design of future systems, by enabling systems to be assembled from sets of ready-made components and also to be transparently extended.

The next three chapters will examine issues in naming - identification of objects in programs - and computation - the manipulation of programs, with specific reference to recursive structure. This separation of naming from computation corresponds to the two parts of the model introduced in this chapter.

Chapter 4

Naming and Recursive Naming

> In fact the name of a command is irrelevant, it's what it **does** that's important. It's a big mistake to confuse the name with the properties of the thing it refers to; calling a cabbage a turnip won't alter the fact that it's green, got leaves and tastes awful;
>
> [Banahan and Rutter 1982]

Naming - the use of names to stand for objects - is an important and pervasive feature of computer systems. In particular, names define the external view of a system, denoting the data that can be manipulated and the commands that can be used to manipulate it. Program variables, file names and virtual addresses are all examples of names at various levels. Similar principles apply to naming at all levels in computer systems and to the naming of many kinds of object. This leads to the belief that there may one day be a uniform theory of naming, independent of the application where naming is used. Needless to say, the present study is only concerned with naming in abstract systems, not with the naming of concrete objects in everyday language.

Naming is a particularly important aspect of computer system design as it allows powerful forms of abstraction. A name is an identifier (a label or token) which stands for the named thing. Something can be identified by name without the need to describe it any further. Names allow sharing, the same object being denoted by each of the many

occurrences of its name. The present chapter explores the structure of naming in computer systems by examining how programs are built from the basic facilities (i.e. names) of the programming language provided by a computer.

All but the most rudimentary programming languages provide the ability to name objects constructed within a program as well as objects supported directly by the language. For example, subroutine names identify commands constructed within the program and similarly there may be named data structures. The naming of objects constructed within the program is the primary concern of this chapter, which examines the <u>structure</u> of naming, in the same way that Chapter 6 will in turn examine the structure of computation.

Some issues commonly associated with 'naming' are specifically excluded from this study: the syntax of names and the choice of a suitable name for a particular object will not be discussed, although in many situations the actual form of a name may indeed be important. For example, in the design of 'interactive' programs it is important to use mnemonic names. In other cases, it can be important to restrict the size of names or to limit the range of symbols to be used within names. The design of most programming languages give little consideration to these issues; in any case, syntactic issues are less important than the structure (or 'architecture') of naming. This chapter concerns naming, not the choice of name for an object.

Another issue not considered here is the choice of objects to be named in a programming language. This is more than a language design issue, it is a question of how systems are separated into layers.

4.1 Abstraction: Naming, Addressing and Routing

Naming is an abstraction which can be used to conceal irrelevant properties of the named objects.

In practice, naming has been neglected as an explicit issue in system design. The main studies of naming have been in the area of programming languages[Barron 1977, Elson 1973, Ledgard and Marcotty 1981] and in computer networks[Saltzer 1982, Pouzin 1976, Watson 1981]. In programming language design, naming is well defined in terms of declarations, scope rules, parameter passing etc. In marked contrast, in computer networks _ad hoc_ naming schemes are often used, although in the established telephone network, naming (i.e. telephone numbering) is carefully structured and controlled.

A name is simply a _token_ which identifies the named object. There is no need for the name to reflect the nature of the named object. For example, the file `junk' may indeed contain rubbish, but it may contain precious items. _Convention_ may dictate that `pi' is the name of a certain mathematical ratio, but in general, `pi' could in fact denote any constant or varying quantity. There is no need for mnemonic names to have their expected meaning, neither is there a need for names to be mnemonic. The property of separation, whereby a name can be chosen independently of any property of the named object, is a form of `referential transparency'[Stoy 1977].

Naming is essential for the separation of programming language from implementation. Because names provide an indirect model, they are decoupled from any actual implementation. A _design_ or _interface_ can be imposed (embodied in a programming language), separately from the structure of the implementation. Such separation is a crucial part of system design. In one direction the notion of separation allows the computer

to be restructured without changing programs. In the other direction, new programs can be written without redesigning the computer.

A simple rule-of-thumb introduced by Shoch[Shoch 1978] has gained currency as a way of sorting out the mess of terminology. It is convenient when starting to think about naming but does not recognize the possibility of naming at multiple levels in a system.

* The name of a resource indicates what we seek,

* an address indicates where it is, and

* a route tells us how to get there.

A 'name' is concerned with use of an object, 'address' and 'route' are concerned with locating and reaching the object.

A mistake which might be made is to assume there is exactly one level of naming in a system - that the 'naming layer' has some absolute position in the system hierarchy. In fact naming occurs at each level in the system. For example in networks, one must beware of thinking that, say, host-naming is conceptually different from node-naming: in particular it is crucial to understand that the same naming techniques are applicable at every level.

J.H. Saltzer has distinguished three levels of naming in computer networks[Saltzer 1982]:

(1) name of resource

(2) name of a host computer

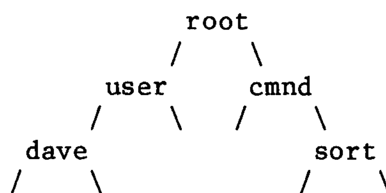
(3) name of network attachment point

In particular, he points out the need to separate these three levels: identifying a computer with its network connection (as in the 'Ethernet'

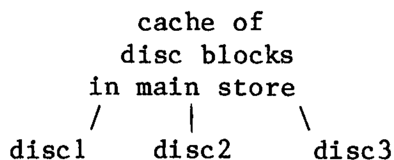
[Metcalfe and Boggs 1976]) or identifying a resource with its host computer, leads to confusion. Changes in any one level of the system will affect other levels unless this separation is enforced; for example, changing the network connection name in an 'Ethernet' also renames the attached computer because the two names (host,connection point) are not distinct.

The approach taken in this thesis is to treat naming as a system structuring issue which is relevant at many levels. (The closest similar description is that of Su[Su 1983]).

Consider an example of multi-level naming where the logical structure may be very different from the underlying low-level structure:

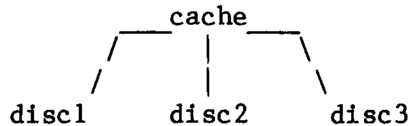


Names: The File System

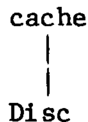


Implementation: Disc Hierarchy

At a lower level in the system, roles may be changed, what was previously regarded as the low-level implementation is now the naming structure, with its own lower-level implementation. The three virtual discs can be implemented on a single large disc (this is common on small UNIX systems).



Names: Disc System



Implementation: One Disc

The important point is that one layer's `address' is another's `name' and so on down the system hierarchy. For a clear understanding it is best to examine one interface at a time - names appearing above the interface, addresses/routes below. Only then can naming be studied as an abstraction, separately from how it is implemented.

4.2 Classification/Analysis: Bindings, Contexts, Resolving

Having insisted that names are mere tokens which tell nothing of the named object, some way of linking names to objects must now be introduced. An ordered pair

(name,object)

will be termed a binding. By creating such a binding, the given name is associated with the object. For example, the binding

(pi,3.142)

associates the name `pi' with the value 3.142. Similarly,

(today,`25th December')

associates the name `today' with value `25th December'. Bindings are the fundamental link between names and objects.

A collection of bindings forms a context: an environment where names can be used[Fraser 1971]. For example in a programming language, the context is the set of variables currently in scope. A context makes it possible to use names meaningfully.

In the context of a Physics textbook, one might read `e is an important constant' and correctly interpret `e' as the electronic charge; in the context of a Mathematics textbook, the phrase `e is an important constant' would be interpreted as a reference to `e', the base of natural logarithms. A context is used to define the meaning of names.

Resolving a Name in a Context

In a given context,

resolve(**N**)

is a function from names to objects, such that

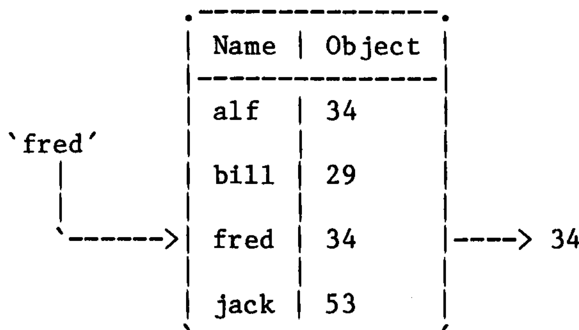
resolve(**N**) = object

if and only if

(**N**,object)

appears in the context.

Resolving the name `fred' in a context consisting of four bindings (`alf',34), (`bill',29), (`fred',34), (`jack',53) results in the value 34.



Every program has an implicit (local) context. In simple systems, where new names cannot be coined within a program, the implicit context is just the set of names built into the programming language (e.g. `read', `write' etc.). In more sophisticated systems, the language allows names to be introduced for objects built within the program.

It may not always be possible for a name to be resolved in a given context: there may be no appropriate binding (**N**,object) in the context. Names which can successfully be resolved (as in the example above) will be termed <u>bound</u> names (they have a binding to some object); names which cannot be resolved are termed <u>free</u> names (they are free of any interpretation in the context). The bound/free distinction is also important to descriptions of the lambda calculus[Burge 1975], which itself can be seen as a model for naming[Fraser 1971].

Sometimes free names occur by mistake: e.g. undeclared variables, sometimes the present context is too `small' and may subsequently be enlarged to include a suitable binding (as when run-time libraries augment the declarations in a program).

In other cases, it may not be possible to resolve a name uniquely. A name **N** is <u>ambiguous</u> if the resolver finds more than one binding for it. That is, the value of

resolve(**N**)

is 'over-determined' because more than one object is bound to **N**. In this case 'resolve' is no longer a deterministic (many to one) function, but is a (many to many) relation. Such names may be treated as an error (as in the case of doubly declared variables); or alternatively a 'non-deterministic' choice is made. The choice is called non-deterministic because there is no way, at the level of abstraction where names are being used, of telling which will be chosen. In fact, at some lower level of abstraction the choice may be completely well determined. For example in an operating system a file name may be bound concurrently to several versions of a file. The ambiguity is resolved by selecting the <u>newest</u> version; the other versions are kept hidden in case the latest version becomes lost or corrupted.

Because naming is part of a logical view of a computer, some objects may perhaps not have a name in that logical model. These are called <u>anonymous objects</u>: objects which do not have a name, but are none-the-less part of the computer. One use of anonymous objects is for 'information hiding'[Parnas 1972] which forms an essential part of the multi-level approach to system design (see Chapter 3). Anonymous objects are securely hidden, in the sense that they cannot be accessed without violating the rule which keeps a program and its interpreter separate. Anonymous objects are used to hold part of the computer state secret from any program. Controlled access is provided viâ routines whose implementation (within the computer) uses the low-level name of the object[Liskov et al. 1970, Wulf et al. 1976]. (A frequently given example is that of a stack - the operations 'push' and 'pop' are made available to programs by name, but the items stored on the stack are hidden; these anonymous objects are manipulated indirectly.)

To summarise the terminology:

Name - some symbol, identifier or token used to denote an object.

Binding - an ordered pair (name,object), indicating that the given name is associated with the given object.

Context - a set of bindings. Every program has an (implicit) local context associated with it.

To Resolve - to take a name and find the corresponding object; a resolver is a function from (name and context) to (object).

In much of the literature `binding' is used in a least three senses, which are not always made explicit or kept distinct:

(1) Binding (noun): A name/object pair (as above)

(2) Binding (transitive verb): The act of creating a name-object pair.

(3) Binding (transitive verb): The act of resolving a name (as just described).

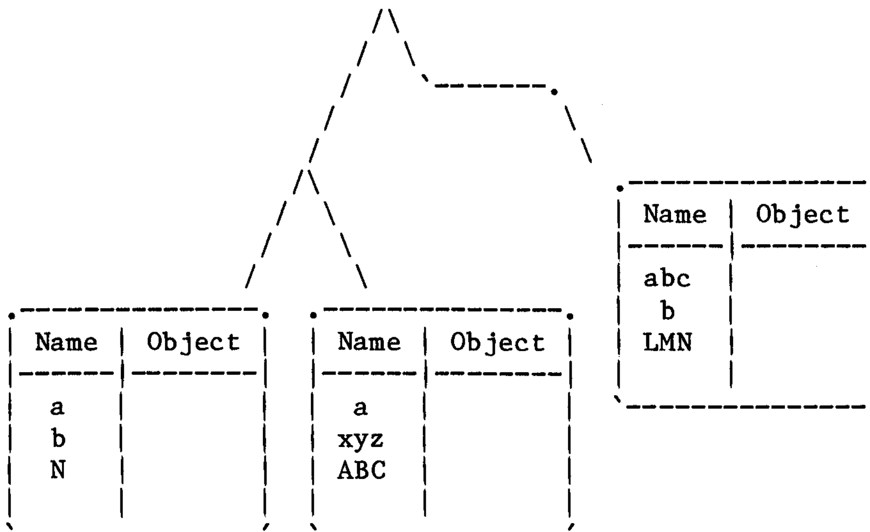
In the present work, `binding' and `context' are used as nouns, `bound' as an adjective and `resolve' as a (transitive) verb. No special term is used for (2), the act of creating a binding. (These definitions are similar to those used by Saltzer[Saltzer 1978].)

Multiple Contexts

So far a single context has been assumed when resolving a name. In general, there may be several contexts. In all but the simplest systems, names can be introduced for objects constructed within the program (subroutines, data structures, etc.). This introduces the possibility of

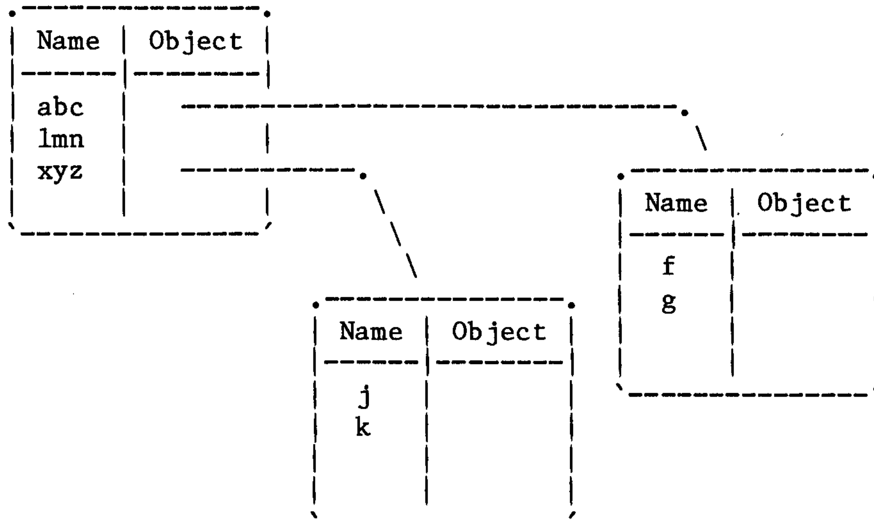
different contexts for different programs (and parts of programs) corresponding to the sets of names introduced there. The many available contexts may be arranged as a structure, forming what can be termed a single structured context.

The structured context may be anonymous, its components not explicitly nameable:

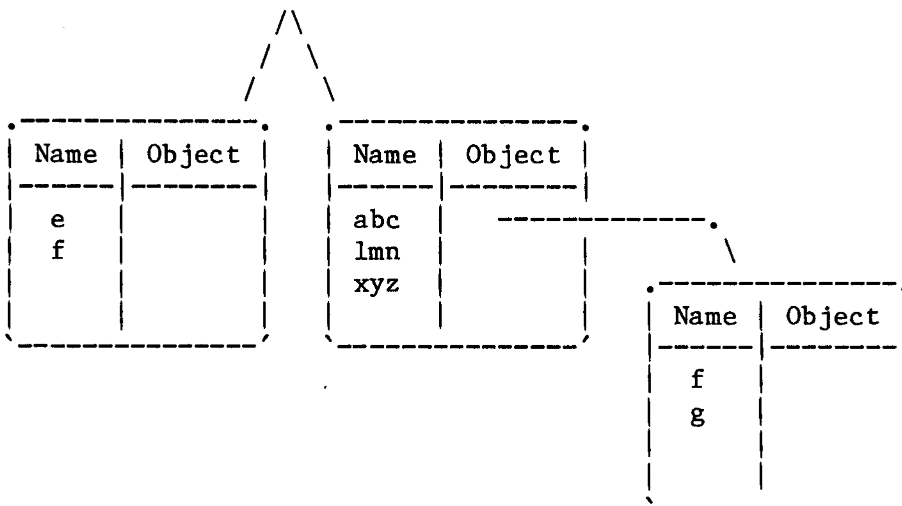


In this case we assume some hidden naming mechanism for selecting parts of the context structure, a mechanism which is only available within the resolver.

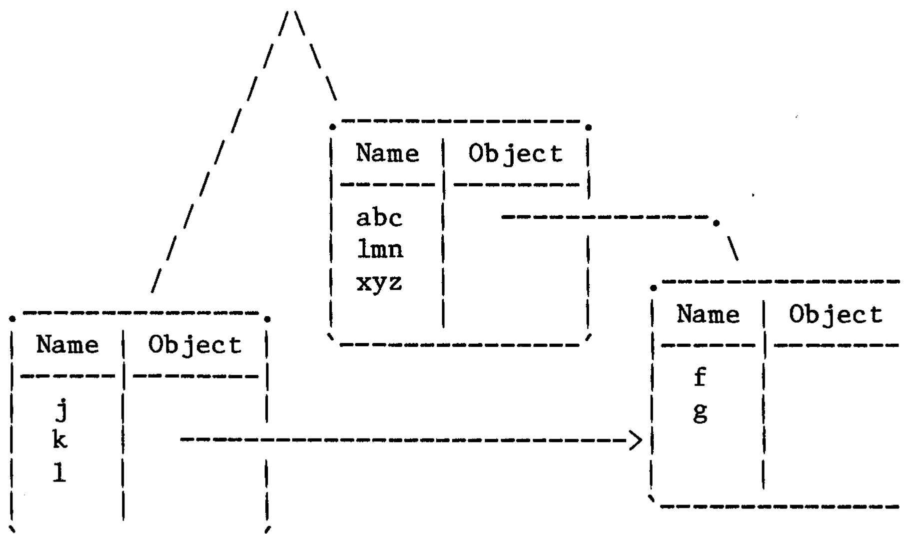
Alternatively, contexts may be named, making it possible to choose a particular context by giving its name:



A third possibility is a 'hybrid' context structure where some contexts are named and others are not.



Finally, contexts may be shared:



In extreme cases, the context structure may be a general graph, not a hierarchy of any kind. This structure is sometimes called a 'naming network'[Saltzer 1978]. In such a network there may be cycles of contexts which name each other. This can cause problems with the termination of a name resolver and with the management of storage for contexts. A solution to the latter problem is given below (in section 7.3).

Resolving Names in Structured Contexts

It is straightforward to resolve a name in a single context; where there are several contexts forming a structure, as in the examples above, the resolver can be more sophisticated. The simplest extension is to make explicit the context currently being searched, which can be done by adding a context parameter to the resolver.

The 'local' or ('implicit') context is always the one to be searched first. In a simple system the implicit context is the underlying language: it contains the names of commands and data provided by the underlying computer. In more general systems there will also be names for objects constructed within the program. There are various conventions for choosing the implicit context of a name; these will be

discussed later (see `Static Resolving vs. Dynamic Resolving' below). For the present it is quite sufficient to disregard the actual choice of implicit context. By convention we assume that if:

$$r(\mathbf{N},\mathbf{C})$$

denotes the result of resolving name **N** in context **C** then:

$$r(\mathbf{N})$$

will denote the result of resolving **N** in the implicit local context of **N**. This has the advantage that the definitions of various resolvers can be presented independently of the choice of the implied starting context. Only within the definitions of the resolvers does the notion of current context become explicit as the resolver traverses the structure.

In many practical naming systems, the first context to be searched is completely implicit. Usually, an explicit choice of starting context is only allowed in the lower levels of systems, usually in the form of `base registers' from which names are resolved.

Corresponding to anonymous and named contexts, two possible naming schemes are: `block structured' naming and `pathname' naming. Identifiers in an ALGOL program are used in the context of surrounding blocks of program text. The collection of declarations from each block forms a set of anonymous nested contexts. A name (ALGOL identifier) is resolved by starting in the local context (enclosing block) and searching outwards until a binding (declaration) is found. Here we have an implicit search of anonymous nested contexts.

A pathname is a compound name formed from a sequence of names. Pathnames are resolved by starting from the local context and resolving the first element of the name; the resulting object is a context, used to resolve the second element of the name and so on until the path is

exhausted. This is an explicitly directed search of named contexts; if at any stage no binding is found for the current pathname element, the name cannot be resolved.

'Block Structured' Naming

'Block structured' naming involves an implied search of anonymous contexts. Contexts are arranged in a tree structure. Only objects in the current context and those contexts enclosing it in the structure may be named. Objects in outer contexts are hidden if their names are bound in the inner context.

In block structured naming, contexts are arranged in a tree structure. Examples of such nested contexts are the block structure of ALGOL 60 programs[ALGOL 60 1963], or 'delimited strings' used to represent programs in some systems[McCarthy 1962, Wilner 1980].

The notion of 'enclosing context' is important in block structured naming. The resolver checks first the local context, then the enclosing contexts. The enclosing context is the one which contains the current context. In particular

enc(**C**)

is defined to be the smallest context enclosing **C**; smallest in the sense that enc(**C**) contains no context which itself encloses C. In a tree structure, enc(**C**) is uniquely defined for all but the outermost context (which is not enclosed in any other). Resolving a name proceeds as follows:

Definition

A name **N** is resolved in a context **C** by the block structured name resolver **Rblock** as follows:

```
Rblock(N,C) ::= r(N,C) if r(N,C) ≠ ?
                else Rblock(N,enc(C))
```

Where:

enc(C) is the smallest context enclosing **C**, as defined above.

r(**N**,**C**) denotes resolve(**N**) the single-context resolver used in context **C**.

? denotes that r(**N**,**C**) is undefined (i.e. **N** is free in **C**); or that enc(**C**) is undefined.

Exceptional case:

When **N** is free, eventually enc(**C**) is evaluated in the 'outermost' block where enc(**C**)=?

so let:

Rblock(N,?) ::= ?

End of Definition

In the degenerate structure with just a single context, block structured naming behaves exactly as the simple naming scheme described above.

Notice that the result of resolving a name **N** is the 'nearest' binding of **N**, that is, the <u>most local</u> binding is always the one used. In this way outer bindings to a name become 'obscured' by inner bindings to the same name. There is no way to override this choice and explicitly select an outer binding.

Note

When describing the lambda calculus, a rather different expla-

nation of block-structured naming is given; it takes bindings and searches for the names to which they correspond. The reasoning is turned around the other way. Instead of an 'outwards' approach, starting from a name and searching outwards for a binding, an 'inwards' approach is used, starting from the binding and looking inwards for occurrences of the bound name in the enclosed contexts.

The outwards approach is easier to understand as it avoids confusion when a name is bound at more than one point, by always selecting the 'closest' binding. In the lambda calculus, it is sometimes necessary to 'systematically re-name' (rule alpha) to ensure names are all distinct throughout[Brady 1977, Burge 1975].

End of Note

Pathname Naming

Pathname naming involves an <u>explicitly</u> <u>directed</u> search of <u>named</u> contexts. A pathname is a list of names of a succession of contexts. Contexts may be arranged by name in a graph structure. Any object which can be reached by following a chain of bindings from context to context can be named. Conversely if an object is hidden it cannot be reached in this way.

pathname ::= string string ...

In practice the syntax of pathnames requires (at least) two distinguished symbols which are not part of simple names; these are used to delimit the pathname as a whole and to separate its component names.

For example in the UNIX file system, the character '/' is separator

and the NULL character is terminator. Any other characters may be used to form (simple) names.

```
pathname ::= string [ /string ]* NULL
           | NULL
```

where `string' is any sequence of characters not containing `/' or NULL. (Naming in UNIX is discussed more fully in Chapter 5).

A pathname is resolved by resolving each of its elements in turn, using the resultant context to resolve the next element. The key to resolving a pathname is that each successive part of the pathname names the context for the rest. Obviously for this to work properly, every part but the last must be bound to a context, not to some other kind of object.

Definition

A name **N** is resolved in a context **C** by the pathname resolver **Rpath** as follows:

Rpath ((**P1**:**P**),**C**) ::= **Rpath**(**P**,r(**P1**,**C**))

Rpath (**N**,**C**) ::= r(**N**,**C**)

Where:

P1:**P** denotes a multiple element pathname whose first element is **P1** and remaining elements **P**. For example in the pathname `a/b/c/d', **P1**=`a' and **P**=`b/c/d'.

N is a single element pathname e.g. `a'.

r(**N**,**C**) denotes resolve(**N**) the single-context resolver used in context **C**.

? denotes that r(**N**,**C**) is undefined (i.e. **N** is free in **C** or **C** is not a context).

Exceptional cases:

When **P1** is free in **C**, r(**P1**,**C**)=?

so for any **P** let:

Rpath(,?) ::= ?

Also when **P1** is bound to a non-context '**D**', and **Rpath** will be called recursively with a non-context as its second argument; so for any **N** and any non-context **D** let:

Rpath(**N**,**D**) ::= ?

End of Definition

In the degenerate structure with just a single context, pathname naming behaves exactly as simple naming as described earlier. For a single context a pathname has but one element so:

resolve(**N**) = **Rpath**(**N**)

The resolver may proceed right-to-left or left-to-right depending on convention. For example pathnames in UNIX and structure selectors in Ada, PL/1 and Euclid have the starting context at the left emphasising the path through the structure starting from the local context:

first/second/last

one.two.three

On the other hand, structures in ALGOL 68 and COBOL have the starting context on the right emphasising the selected component:

last **of** second **of** first

item **in** record **in** file

Alternative Naming Mechanisms

Other naming mechanisms exist, some using anonymous contexts, some named contexts or a hybrid context structure.

(1) Hybrid Naming Schemes

It is possible for the two mechanisms described above to be combined in various ways to form a hybrid naming mechanism. For instance, block-structured naming may be used to resolve the first element of a pathname and pathname naming can be used for the rest. Consider array element naming in Pascal: a multi-dimensional array can be viewed as a set of nested arrays

array [1..5] of array [1..5] of array [1..5] of T

an element is named:

a[3,4,5]

or equivalently: a[3][4][5]

This name can be considered as a pathname of four elements `a',`3',`4' and `5'. The first element `a' is resolved using the block structured resolver and provides a starting context for a search through the nested arrays resolving the remaining `pathname' [3][4][5].

Another possibility is to use block structured naming for single element names and pathname naming for compound names.

resolve(**N**) ::= if rest(**N**)=NIL then **Rblock(N)** else **Rpath(N)**

This is rather like the convention used for command names in the UNIX shell (as will be described in Chapter 5).

(2) Other Mechanisms

As well as combining the above mechanisms, it is of course possible to invent totally different ones.

(a) Combined (Pathname and Block Structured) Naming

Another possibility is to repeatedly use **Rblock** (the block structured resolver) on each element of the pathname. This 'Combined Naming' **R** has the advantages of both block-structured and pathname naming.

Definition

A name **N** is resolved in a context **C** by the 'Combined' resolver **R** as follows:

R((P1:P),C) ::= **R(P,Rblock(P1,C))**

P(N,C) ::= r(**N,C**)

End of Definition

Note that this is the same as **Rpath** except that a call of 'r' has been replaced by a call of **Rblock.**

The resolver defined above has interesting properties:

if **Rblock(N)**=X then **R(N)**=X

if **Rpath(P)**=Y then **R(P)**=Y

Also, for all names which are bound in the local context:

R(N) = **Rpath(N)** = **Rblock(N)** = r(**N**)

For all the names which a block structured resolver or pathname resolver

finds a binding, this scheme finds the same binding. In addition a new class of names is now resolvable in which each element of a pathname is a block-structured name for the context of the rest of the path.

For example when two contexts `a' and `b' are at the same level of nesting, objects bound to names in `b' can be named from `a':

```
begin
    a: begin
         integer i;

       end

    b: begin

         print(a/i);

       end
end
```

Here `a/i' is resolved to the variable `i' in context `a'. Another example will be given below, when the implications of Combined Naming are discussed.

(b) Path-List naming

A directed search can be made: a hidden list **L** of names is kept by the resolver, naming a series of contexts; these are searched in turn until a binding is found. Unlike block structured naming, the series of contexts is defined explicitly, independent of program structure. Altering **L** alters the contexts to be searched. Unlike pathname naming, the series of contexts searched is not indicated in the name itself: the successive elements of **L** can be unrelated and independent of the context structure.

Definition

A name **N** is resolved by the path-list name resolver **R** as follows:

```
R(N,(L1:L)) ::= r(N,L1) if r(N,L1) ≠ ?
                        else R(N,L)

R(N,C)      ::= r(N,C)
```

Where

L1:L Is a list of context names whose first element is **L1** and remaining elements **L.**

C Is a single context name.

End of Definition

For example in the UNIX 'shell' (command interpreter) a variable 'PATH' is used to hold a colon-separated list of names. The default value is

PATH=:/bin:/usr/bin

That is, three contexts are named, the local context (named by the initial null name) and two standard directories '/bin' and '/usr/bin'. This system is inherited from MULTICS[Organick 1972] where subroutine names are resolved this way.

(c) Recursive Pathnames

Pathnames can be extended to have a more general structure where a pathname is a sequence of pathnames instead a of a sequence of simple names. This can be called 'recursive pathname' naming.

```
name ::= string
       | name name
```

In practice additional symbols are needed to delimit the nested names. For example UNIX pathnames could be extended by using '(' and ')' to group sub-names:

a/(b/c)/d

The resolver for such names is derived from **Rpath** by replacing one occurrence of 'r' by '**Rpath**' in the main loop making it recursive.

Definition

A name **N** is resolved in a context **C** by the recursive resolver **R** as follows:

R((**P1**:**P**),**C**) ::= **R**(**P**,**R**(**P1**,**C**))

R(**N**,**C**) ::= r(**N**,**C**)

End of Definition

As will be discussed below, the exact meaning of such recursive pathnames is by no means obvious.

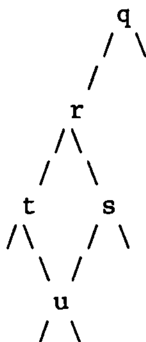
Implications of the Naming Mechanisms

Block structured naming does not require contexts to be named. Provided that the context containing the binding required is not deleted, other contexts can be added or deleted freely. Alternatively, if a new binding of the name is introduced, it may over-ride the existing one because the resolver finds the new one first. Consider an example of hiding in Block-Structured Naming:

```
begin
    integer x;

    begin
        integer x;

        comment outer 'x' inaccessible here;

    end
end
```

Block structured naming only allows naming of (i.e. access to) objects in the current and enclosing contexts, not contexts 'within' the current context or which do not enclose the current context. This enforces a certain modularity; changes in two contexts are independent unless one encloses the other.

Because the notion of an 'enclosing context' is fundamental to block structured naming, it is usual for contexts to form a tree and not be named. Block structured naming can be extended to encompass shared contexts (i.e. contexts which have names bound to them in more than one other context), provided that enc(C) could be correspondingly extended. As long as the resulting structure is still a hierarchy the enclosing context of C can be defined as the context enclosing all contexts naming C. For example:



with five contexts q,r,s,t,u arranged as shown, the enclosing context of 'u' is 'r'.

It might be necessary to insist that the names bound in 't' must be distinct from those bound in 's'. This kind of situation where the enclosing context is not unique has been called 'multiple inheritance' (by those working on the language Smalltalk) and is an area of current research.

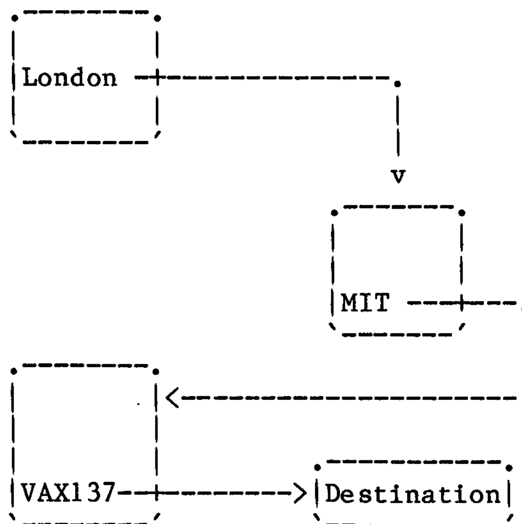
Pathnames require contexts that are named. This brings a close coupling between names and the context structure: pathnames must reflect

the context structure, otherwise they cannot be successfully resolved.

For example consider the names of computers in a network. Let each computer keep a context with names bound to the addresses of its neighbours in the network. A `source route'[Sunshine 1982] is then a pathname over this system which selects a path across the network; the path is defined by the initiator of a message before the message is sent. Like the national and international telephone numbering system, the structure of the `source route' reflects the location of the site being named, but it is still separate from the actual physical route used to reach it.

Starting from Newcastle:

send(London:MIT:VAX137 , <message>)



The sender must have information about the logical network structure, a change in names at MIT requires a change in the `source route' stored at Newcastle (and in every other route to or through MIT). Changes in telephone dialling codes require similar upheaval. At the time of writing, the local telephone network in the North East of England is being changed from a two level system

<town> <number>   e.g. 0632-329233

into a three level system.

<region> <area> <number> e.g. 091-287-9233

This is an operation which will have considerable impact on telephone users over several years.

A pathname becomes invalid if any binding on its length is destroyed. On the other hand, pathnames can go 'down' into subparts of the structure which are inaccessible using block structured names. In this way, pathnames can be used in non-hierarchical systems where the contexts form a general naming network. This is a useful feature provided it is not used to violate the modularity implied by the naming structure.

Combined or 'hybrid' naming mechanisms exist, as shown above, since the simple mechanisms are sometimes found inadequate.

* Block structured names are short, but only access enclosing blocks

* Pathnames allow access to everything at the price of length and structure dependence.

The example given above, array indices in a programming language, shows where one mechanism is a base and the other is added to overcome its deficiencies. Such hybrid schemes reflect a duality of purpose in names. For example, in naming an element of a record in Pascal:

employee.birthdate.year

the first element is a <u>variable</u>, naming the record amongst all the objects in the program. The subsequent elements are <u>selectors</u> picking fields within the record.

In the alternative naming schemes, scheme (a) (Combined Naming) is

an invention of my own which has the useful property that it is a combination of both block structured and pathname naming.

* Names need not always mirror the context structure - the enclosing context is <u>implicit</u>, it need not have a name (e.g `generic' names for the enclosing context such as `..' in UNIX and `super' in Smalltalk-80 become unnecessary).

* `All' objects in the system can be named - objects in any named context can be named; objects may also be deliberately hidden by placing them in an anonymous context.

This provides a useful generalization which may have significant advantages for naming in computer networks and for `attribute inheritance' in object oriented languages. (It is rather similar to the way in which messages are routed in Smalltalk-80 but allows arbitrary length paths.)

Combined naming is in a sense less strict than attribute naming in, say, SIMULA. A SIMULA class attribute is named by a pathname of the form <class>.<attribute> where the specified attribute must be defined in the named class. Using the Combined Naming it is not necessary to be so exact. A SIMULA name `a.b' is interpreted as `name b bound to an object in class a'; a Combined Naming name `a/b' is interpreted as `name b as resolved from class a'. With Combined Naming, class naming and attribute naming are symmetric.

Consider an example showing use of this:

```
begin
    integer i;

    a: begin
           integer i;
           i:=1;

       end;

    b: begin
           i:=2;

       end;

    print(a/i);
    print(b/i);

end
```

The name `a/i′ names the `i′ in context `a′. The name `b/i′ is resolved to the `i′ in the outer context in exactly the same way as is the `i′ used within context `b′.

Scheme (b) (path-list naming), is in practice the same as using an array of contexts (see `Multiple Contexts′ above). The power and danger of the scheme lies in the ease with which **L** can be arbitrarily defined. For example one choice of **L** can be used to mimic block structured naming:

$$\mathbf{L} = \mathbf{C}\ ,\ enc(\mathbf{C})\ ,\ enc(enc(\mathbf{C}))\ ,\ enc(enc(enc(\mathbf{C})))\ \ldots$$

Scheme (c) is essentially a general form of recursive naming where pathnames are not restricted to being a list of simple elements. This system will be discussed in section 4.3.

Static Resolving vs. Dynamic Resolving

Given the mechanisms for resolving names in contexts, how are these to be put to use in the naming of objects? An important consideration for the modularity of systems is the question of where a name is resolved.

The system behaviour is very different depending on whether names are resolved where they were created or in the context where they are used. Two of the many possible ways of using names will now be described, one of which is especially suited to modular systems.

The naming systems described so far have relied on the concept of an implicit local context as a starting point for resolving a name. In a simple system, no other context is recognized; in structured systems, the local context was identified as the first to be searched for a suitable binding of a name. Now it is appropriate to explore the concept of 'implicit' or 'local' context further and see its connection with the ideals of system structuring.

So far, a single local context has been assumed. In practice many contexts may be available concurrently as the starting point for the resolver. It is a matter of naming policy which local context is used. Naming can be thought of as a combination of two parts: mechanisms, and policies for using those mechanisms. Two policies of particular interest are:

(1) Static Resolving - names are resolved in the context where the program containing them was created - the defining context.

(2) Dynamic Resolving - names are resolved in the context where the program containing them is being used (which may not be the defining context).

These two alternatives are widely known in the study of programming languages, but are also valid policies in any naming system. For example, ALGOL 60 uses static resolving, as is demonstrated by the program fragment:

```
begin
    integer i;
    procedure addone;  begin i:=i+1; end;
    i:=1;
    begin
        integer i;
        i:=1;
        addone;
        print(i);
    end
    print(i);
end
```

The output is `1' followed by `2'.

The example demonstrates that the choice of starting context (i.e. static vs. dynamic resolving) is separate from the choice of resolver. The `i' inside `addone' is resolved in the (static) context of the outer block where `addone' was defined, despite the redeclaration (re-binding) of `i' in the inner (dynamic) block where `addone' is used.

If ALGOL 60 had dynamic resolving (as has LISP 1.5) then `i' in `addone' would be resolved in the inner (dynamic) block producing output of `2' followed by `1'.

Modularity vs. Textual Substitution

Naming allows objects to be shared; modularity is a discipline on sharing. Indirection, an important feature of naming, separates definition from use. Once a binding is set up, a single object can be used by name in many places. Modularity implies encapsulation, so that it is immaterial whether an object is used directly or by name. The issue of static vs. dynamic resolving is a question of modularity: in naming terms modularity means being able to use an object without knowing about its internal names (which Saltzer calls `Modular Sharing'[Saltzer 1978,p 101]) and with the same behaviour at every use (the latter sometimes

being called `Referential Transparency'[Henderson 1980]).

* Static resolving encourages modularity, dynamic resolving does not. Naming within an object is independent of where and when the object is used.

* Dynamic resolving is equivalent to textual substitution. Naming within an object is completely dependent on where and when the object is used by naming it.

Consider the internal names embodied in an object. With static resolving, these internal names are resolved in the context where the object containing them was created (defined). When the object is itself named in other contexts, the internal names are still resolved in the defining context. For example, consider this fragment of ALGOL 60:

```
begin
    integer i;
    procedure increment;  begin i := i+delta; end;
    i := 1;
    begin
        integer delta;
        increment;
    end
end
```

In the outer block `increment' is bound to a procedure body containing further names. The program is badly formed ALGOL 60; even though a name `delta' is declared (bound) in the inner block (context), static resolving means that `delta' must be resolved in the outer block and it is free (i.e. undeclared) there. No bindings at the point of use of a named object can change in any way how the names within the object are resolved. This feature of static resolving reinforces modularity and encourages abstraction. Internal names can only be re-bound by parameter passing. The object is encapsulated at the point of definition, it is not subsequently changed by features of the environment in which it is used.

When dynamic binding is used, internal names are resolved in the contexts where the object is being used (i.e. where it is named). The binding used for a name in an object will vary as the object is used in different contexts. If ALGOL 60 were to use dynamic resolving, the example above would have also been a valid program: both 'i' and 'delta' are bound at the point where 'increment' is used. Only bindings available at the point of use are utilized by the dynamic resolver. This feature of dynamic resolving encourages naming to be viewed as 'abbreviation' or 'shorthand' for the named objects. Internal names of the named object are exposed at the point of use and resolved there. In this way their binding may change entirely between one use and the next. The choice of names in the contexts of use are closely related to the choice of names in the named object itself.

Dynamic resolving is often used in macro-processors[Henderson and Gimson 1981]; each occurrence of a macro name is literally replaced by the corresponding object. Naming is simply a <u>textual substitution</u>.

With static resolving, all bindings to objects at the point of <u>use</u> must be imported into an object through parameters. This is a rigorous programming discipline which enforces modularity (as described above). However, if some binding is the same at nearly every use, only changing a small proportion of times, static resolving will nevertheless require <u>all</u> uses to specify it as a parameter. There is an advantage here for dynamic resolving: it allows names to depend on context of use without being imported as parameters. The fundamental disadvantage of dynamic resolving remains, that is that the context of <u>use</u> can change the named object.

To get around the problem of 'nearly constant' parameters, some systems use the notion of 'default' parameter values. This idea could be pursued further, especially in relation to the Combined Resolver, but

is beyond the scope of the present work.

In a system with dynamic resolving, static resolving can be attempted by using unique names, that is, names that will be resolved to the same object whether starting from the defining context or the context of use. With pathnames, it is sufficient to use a name long enough to 'span' the two contexts. Alternatively some loop-hole may allow direct access to objects, not viâ a name: e.g. address calculation. An address can be built into an object to bind it to another without using naming at all (this subversion is not recommended).

4.3 Recursive Naming

Recursive naming is the combination of recursive contexts and a recursive resolver. A more accurate description would be 'recursively structured' naming or 'recursively defined' naming. Recursive naming brings the advantages of recursive structure (Chapter 2) to naming:

* Combination: RN + RN = RN - a combination of recursive naming systems is another of the same type.

* Location independence - every context has the same basic properties, so can be used by the same resolver. Conversely a resolver can start to resolve a name from anywhere in the context structure.

* Extension - the set of bound names can be extended by embedding one context in another.

Recursive naming can be provided in a number of ways, many of which have been mentioned already without actually pointing out that they were recursive. The current section will describe recursive naming and identify those naming schemes which implement it.

Two ingredients

* Recursive context structure

* Recursive name-resolver

characterize recursive naming. Recursive contexts provide a recursively extensible environment and a recursive name resolver provides a completely general way of using that environment. It is not necessary for names themselves to have a recursive structure (as in `a/(b/c)/d') for naming to be recursive. A recursively structured name `a/(b/c)/d' is not the same as the pathname a/b/c/d. A particular difficulty with such recursive names is that precedence must be defined: when resolving the name a/(b/c)/d the nested component (b/c) may be resolved to, say, `X' by starting in the implicit local context, and to `Y' in the context `a'. If `parentheses' are evaluated before anything else, the name is parsed as

a/X/d

but if names are evaluated left-to-right it is parsed as

a/Y/d

Recursive naming is concerned with the recursive definition and use of naming. It is not necessary to provide `names-of-names'. Such bindings of name to name are merely a notational convenience adding nothing to the structure of naming (for example with statements in Pascal or renames in COBOL[Ledgard and Marcotty 1981].

Recursive Contexts

Recursive contexts - contexts within contexts - are recursive structures where each non-atom is a context and where contexts may be nested to any depth. Such a recursive context might be defined as

```
context ::= binding binding ...
binding ::= name object
object  ::= context
          | atom
```

In general, atoms will represent basic objects, such as storage locations, files, network stations, etc.

Some context structures are not recursive structures. Consider a system where only two levels of nesting are allowed and there is a type of context corresponding to each level.

```
network_context ::= network_binding network_binding ...
network_binding ::= name host_context
host_context    ::= name host_binding
host_binding    ::= name file
```

Such a system does not implement recursive naming, as it does not provide recursive contexts. As a result

(i) there is no facility for combining networks because it is not possible to provide inter-network naming.

(ii) it is not possible to hide the distinction between <u>host</u> and <u>file</u> because contexts in each level of the structure can only contain bindings to fixed classes of objects.

The UNIX United/Newcastle Connection scheme, discussed later in Chapter 5, exploits recursive naming extensively and these limitations would be disastrous to it.

Recursive Resolvers

A recursive resolver is a flexible general purpose mechanism which:

* Can be used in any context

* Works independently of the size and 'shape' of the context structure: changes and additions to the structure do not require changes to the resolver.

A recursive resolver is a general resolver which can be defined recursively. **Rpath** and **Rblock**, described above, are recursive resolvers in this sense. Some resolvers are not recursive resolvers. Consider the context structure in the example above, which might be used with two resolvers

Rnet : (name, network_context) -> host_context

Rhost : (name, host_context) -> file

These are specific to each level of the system and cannot work on more general context structures. On the other hand a more general resolver (which might even be recursive)

Rgeneral : (name, context) -> object

could be used in the above structure, but the resulting system would still not have recursive naming because both recursive contexts and a recursive resolver are required.

Some Examples of Recursive Naming

(i) Variable names in ALGOL 60

Recursive context: <blocks> are contexts and may be indefinitely nested

Recursive resolver: **Rblock** is used; it can be used from any block, in any structure of blocks.

(ii) File names in UNIX

Recursive context: directories may be nested to any depth

Recursive resolver: **Rpath** is used; it can start from any current directory and can traverse the whole file system.

The advantages of recursive naming have been demonstrated in other situations as diverse as menu-based searching[Apperley and Spence 1983] and the naming of registers in a microprocessor[Woods and Wheen 1983].

Combining Contexts

Recursive naming is essential if contexts are to be freely combined. If contexts can be nested then two may be combined by constructing a new context with them both as sub-contexts.

```
                     C3
C1 + C2    -->      /  \
                   C1   C2
```

The combination has the useful property that any name bound in the original contexts retains the same binding.

In non-recursive contexts where there is some distinguished top level or restriction on depth of nesting. The worst case is in single 'flat' naming systems consisting of but a single context. The following discussion of problems in 'flat' systems is equally applicable to the top-most level of 'shallow' systems.

Combining 'Flat' Naming Systems

The difficulty with 'flat' naming systems is the lack of a mechanism for combining separate systems into a whole; each flat system will recognize

no other. If flat systems are combined, some change or extra mechanism is inevitable. Examples of systems with flat naming include

* Host names on some computer networks e.g. Cambridge ring, Ethernet

* Names in some programming languages e.g. variable names in BASIC, function names in C

* Storage location names in some machine code programs e.g. the `absolute' code produced by some assemblers and loaders

In each case the flat naming structure is sufficient for a single system. When there is more than one network, more than one sub-program or more than one binary module, there are problems caused by flat naming.

Three methods of combining flat naming schemes will be explored (two of these three are mentioned in[Sunshine 1982]). Each is applicable also to `shallow' naming (of the kind in the network_context-host_context example above) where only a limited depth of context nesting is allowed, or a non-recursive resolver is used.

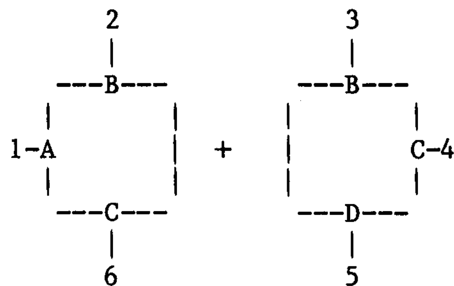
(1) Merging - the two systems are simply merged objects with non-unique names being rebound.

(2) Extension - adding a new level to the context hierarchy to add pathnames for inter-system naming. This requires changes to the existing resolver for it to process the new pathnames.

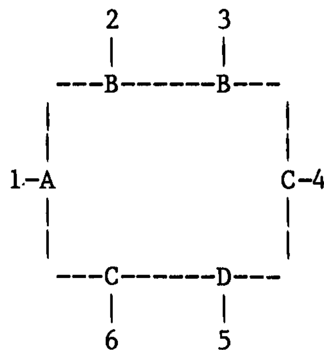
(3) Mapping - names between systems. `Surrogate' names are introduced for inter-system naming. This requires only additional mechanism, not changes to existing names or resolver.

Firstly consider how two separate naming systems can be merged together and treated as the same for naming purposes. For example two BASIC programs might be copied into a file to produce a single combined program.

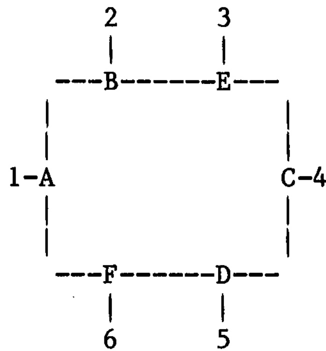
No changes are needed to the name resolver but changes are needed to the names. For example if `T' has been used to represent `time' in one program and to represent `temperature' in the other then some compromise must be made. Although in each constituent program `T' is used for a unique purpose, it is no longer unique in the combined program. This kind of clash also occurs when two networks are joined together retaining the original names. Here two communication rings with stations named A,B,C and B,C,D and host computers 1,2,3 and 4,5,6, are combined. The two systems:



become one larger system:



Names `B' and `C' are multiply-defined in the combined ring and must be altered to avoid ambiguity. Every use of the original `B' and `C' must be systematically changed, as well as changing the definitions in the new combined context.



An alternative way to combine flat naming schemes is to augment the naming to include system names:

systemname.original_name

Provided that

original_name

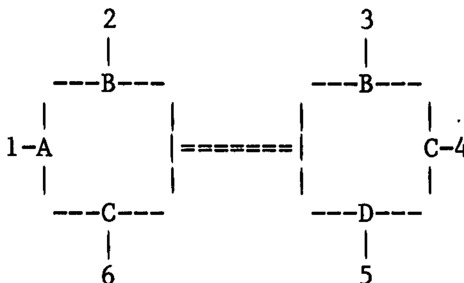
is interpreted as

local_system.name

there is no need to change existing names. However the existing resolver must be changed to deal with this new syntax. This may not be a trivial change: in the Cambridge ring network, name resolvers are a collection of chips in the ring station: so changes are impossible without a complete re-design of the station hardware! Such changes to embrace multiple networks must be implemented at a higher level. This approach is used in the Ethernet PUP architecture which imposes a structure at the higher (protocol) level whilst relying on flat naming in the physical links[Xerox 1981].

The third alternative is to introduce a mapping of names between the previously-separate systems. Names are translated en route between systems. The naming schemes are logically combined, requiring no

changes to either system. Referring to the example above, logically combining the networks corresponds to placing a gateway between them:



The gateway will contain mapping tables which ensure that neither names nor resolvers need be changed. The tables record the correspondence (equivalence) between names of the two systems. Such a combination of systems is represented by

Flat + Flat = R(Flat , Flat)

where R is a relation between the names in the two systems. In the example, R might be

(BB-> B)
(D -> D)

where (a -> b) means that name \`a' in the first system corresponds to name \`b' in the second system: if \`b' names object \`O' in the second system, then \`a' names \`O' from the first. In the example \`BB' is used to name host 3 and \`D' to name host 4. The relation can describe mapping in both directions, for example when all hosts are namable in both systems:

(BB-> B)
(D -> C)
(B <- E)
(C <- F)

In one sense name mapping is just a dynamic version of the ambiguous name re-binding scheme described above. Its advantage is as a

natural extension, requiring no changes to existing names or resolvers; in other words it is completely transparent. The only requirement is that the added resolving mechanism can distinguish local from remote names. Indeed this is precisely how UNIX process names and user names are mapped between systems in UNIX United (see Chapter 5). A simple way to do this is to try to resolve a name locally, if this fails, to consult the mapping relation.

Name mapping can be set up to have various useful properties, it can be:

Symmetric for all a,b:

if (a->b) holds then (a<-b) also holds

Transitive for any three names a,b,c:

if (a->b) and (b->c) hold then (a->c) also holds

These or other properties can make the combined system easier to use, but it is still not as general as a recursive naming system. In particular, by carefully controlling the mapping relations it is possible to simulate most forms of non-flat naming. This simulation is more attractive than an undisciplined mapping between systems for which the above properties may not hold.

* * * * *

The present chapter has attempted to show that naming is a pervasive issue in computer system design, that there is a useful model which allows naming to be discussed explicitly and that recursive naming has particular advantages of locality and extensibility. The research contribution here is not so much in novelty of material but as a synthesis of existing ideas. It is a bringing out into the open of

some design issues often (mistakenly) left implicit.

It is by no means universally recognized that naming is an issue at _many_ levels in a typical computer system; neither is it recognized that _every_ computer interface is a programming language. In emphasising many-level naming I am simplifying system design by showing the similarity and separability of naming at each level. At each level the same design issues recur and the same problems have to be solved in the circumstances of that level.

The model of naming presented here draws from existing material, organizing it and adding a discussion of structure and modularity. In addition the 'Combined' name resolver is an original contribution, synthesising block structured and pathname naming. I am not aware of any language which has used such a scheme, despite its simplicity.

Recursive naming is an important concept to consider in the design of computer systems. If naming is not recursive then there is a point where it is no longer possible to combine (or sub-divide) systems without affecting the ability to name things. Recursive naming is more than just structured naming and is far removed from flat naming which relies on unique names for each object. Recursive naming provides an all-important flexibility enabling the arbitrary combination and subdivision of systems.

Chapter 5

UNIX United and the Newcastle Connection

Recursion is not an elitist plaything
[Headline, "Computing" October 13th 1984]

This chapter demonstrates recursive naming in practical use and is sufficiently large to be separated from the above general discussion of naming. It describes a distributed computer system whose design (UNIX United) and implementation (the Newcastle Connection), show the generality that can be achieved using recursive naming.

'UNIX United' is a design for a distributed computer system based on UNIX. The design simply makes the existing UNIX naming scheme apply over multiple UNIX systems. In UNIX United, both local and remote objects are named with the same (UNIX) naming. Because file naming in UNIX is recursive, it can be expanded in UNIX United to allow naming of files in other systems without changing syntax or semantics. Unfortunately process naming (and 'user' naming) in UNIX is flat: generality has to be compromised for the sake of compatibility; processes and users can only be named in a limited way between systems in UNIX United.

The 'Newcastle Connection' is a software 'layer', added to standard UNIX systems which enables UNIX United naming to be implemented. The Newcastle Connection 'layer' is a recursive interpreter extension. Operations on local objects are passed through to the local system.

Operations on remote objects are diverted to the appropriate remote system. The Newcastle Connection extends the UNIX low-level (Kernel) interface by allowing existing, unchanged programs, to access and manipulate remote objects in the same way as objects on the local system.

The UNIX United/Newcastle Connection work has already been published in a joint paper by its main designers: L.F. Marshall, B. Randell and myself[Brownbridge et al. 1982]. The present chapter will concentrate on those areas of the design where my own influence was the strongest: the naming architecture of UNIX United and the notion of the Newcastle Connection as a transparent extension of the UNIX kernel, (as opposed to a modification of it). The 'UNIX server' and remote execution part of the design are due mainly to L.F. Marshall. The Remote Procedure Call mechanism used for inter-UNIX communication is due to F.E. Panzieri and S.K. Shrivastava[Shrivastava and Panzieri 1982]. (Other work, on highly reliable and highly secure distributed UNIX systems, has been pursued by others[Rushby and Randell 1983], using the Newcastle Connection as a base.)

The first three sections of the chapter cover naming in UNIX (Section 5.1); UNIX United, showing how standard UNIX naming can be extended across multiple systems (Section 5.2) and the Newcastle Connection, a recursive extension of the UNIX kernel. Finally some other distributed UNIX systems are reviewed.

Note that throughout this chapter, 'UNIX' refers to Bell Laboratories UNIX Version 7, unless otherwise stated; similarly, citations of the form 'name(N)' refer to section 'name' in part 'N' of the UNIX Version 7 Programmer's Manual[Kernighan and McIlroy 1978]. 'Version 7' forms the basis of the more recent varieties of UNIX (System V, 4.2BSD, 8th Edition ...) and was also the one on which our implementation work was performed. The reader is referred to the original UNIX

paper[Thompson and Ritchie 1978] for a summary of UNIX, to 'UNIX: The Book'[Banahan and Rutter 1982] for a concise tutorial on UNIX and 'The UNIX Programming Environment'[Kernighan and Pike 1984] for an in-depth treatment of UNIX programming.

5.1 Naming in UNIX

Naming in UNIX is provided by three components of the system:

shell (command interpreter)

utilities (standard commands)

kernel (resident supervisor)

These can be arranged as a hierarchy, each level modifying the naming provided by the level below.

```
-------------  command interpreter
    shell
-------------  standard commands
  utilities
-------------  system calls
   kernel
```

This structure will be used to describe naming in the UNIX system.

Two classes of UNIX system objects are provided: files and processes. If an operating system is regarded as a (virtual) computer, then files correspond to the store and processes to the processor. We shall be interested here with the naming for files and processes provided by the kernel, utilities and shell and not with the various other objects in the programming languages available on UNIX systems.

Kernel Naming Facilities

The kernel is the base component of a UNIX system which provides files as an abstraction of the store (including secondary store) and processes as an abstraction of the (multi-programmed) processor. The kernel

provides naming for files and processes and access to them through its programming language interface: the `system calls' (described in Section 2 of the `UNIX Programmer's Manual'[Kernighan and McIlroy 1978]). Names appearing as parameters to system calls are resolved by the kernel.

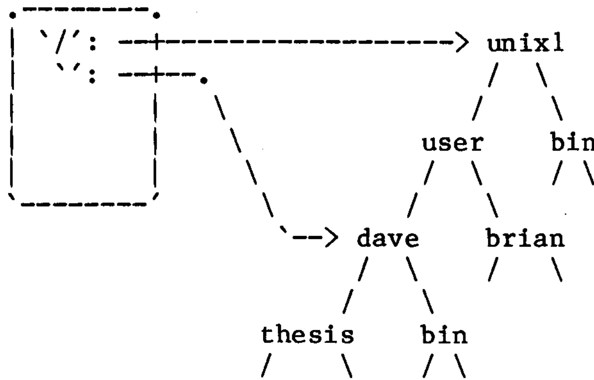
The UNIX kernel provides pathname naming for files. One type of file, called a `directory', is interpreted as a context. Directories can contain bindings to any type of file, including `directory'. The kernel checks the types of files named in system calls, preventing (most forms of) untoward interference with the file system structure.

Files (of all types) are named by pathnames with the following syntax:

```
pathname  ::=  path
            |  /path

    path  ::=  name  { /name }
            |  <null string>

    name  ::=  <string of 1 to N characters>
```

(where `{ x }' indicates `zero or more occurrences of x' and N=14 in UNIX Version 7). These names are interpreted by a standard pathname resolver (defined in Chapter 4).

The kernel maintains two special bindings to the `working' and `root' directories associated with each process. The working directory is always bound to the null name; the root directory is always bound to the name `/'.



In the example above, the names '/user/dave/thesis' and 'thesis' refer to exactly the same file.

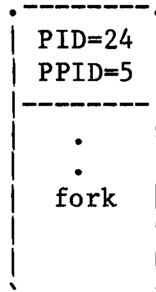
When a process makes a system call, pathnames beginning with '/' are resolved starting from the root directory and all other pathnames are resolved from the working directory. (Note: '/' is used both to indicate 'root' at the start of a pathname and to separate the components of a pathname). A process may change root or working directory by making a system call (chroot(2), chdir(2)); the directories are initially inherited from the 'parent' process (see below).

It may seem strange to have these two alternative starting contexts, but it will be seen later that they roughly correspond to the notion of 'current system' and 'current workspace'.

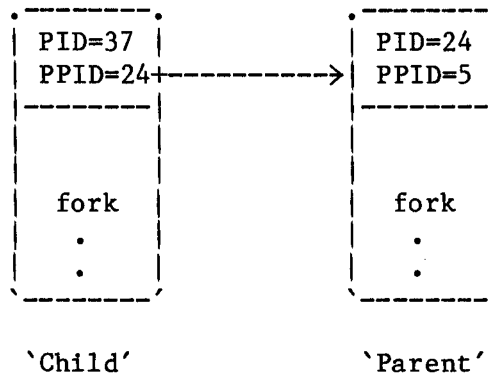
Process naming in UNIX is flat. There is a single context (the system) and process names are unique within it. The UNIX 'process identifiers' are names: they provide indirect means of manipulating processes, independent of their size, priority, etc.; but on the other hand they are all coined by the system and there is no way of forcing a process to have a given name.

For each process, the kernel records the 'parent' process which created it. Each process has an associated process name (called the

PID) and name for its parent process (PPID). Although process naming is flat, there is a tree structure overlaid on it by this parent-child relation. The 'parent' creates a 'child' by making the system call 'fork' which. Before the 'fork':



After execution of 'fork', process '37' is created as a copy of process 24, with PPID=24:



A final naming mechanism for processes, 'process groups', is a marginal feature of UNIX Version 7, although it has been developed in later editions of UNIX[System V 1983]. A process group is a named sub-tree of the parent-child process structure. It is significant in providing a way of naming groups of processes as one.

Utility Level Naming

The utility programs are those UNIX programs (documented in Section 1 of the UNIX manual[Kernighan and McIlroy 1978]) which reside in every UNIX system file store. Here, I am only concerned with the utilities that

affect file naming and process naming (ln(1), mkdir(1), rmdir(1), ps(1)), etc.). Other naming issues, for example naming within the individual languages provided as utilities, will not be considered (except the special case of the `shell' which is treated separately below).

As implemented by the kernel, the file system can have an arbitrary structure, possibly with some unconnected components. For various reasons, the utilities restrict the file system to be a (single) tree-like structure where only non-directory files may be shared (i.e. appear in more than one binding). If only the utilities are used, it is not possible to build any other file system structure.

The directories do not strictly form a tree because of the `.' and `..' bindings. When a directory is created (mkdir(1)), bindings to itself (named `.') and to its parent (named `..') are always created in it. However these are special cases and no other non-tree bindings can be made. The `.' and `..' bindings provide an explicit name for the local and the enclosing (`parent') directory.

In a file system built using the utilities, the following properties will hold, given that the relevant files are accessible (`cd' changes the current working directory):

```
 file = ./file
a/./b = a/b
  ./. = .
 file = (cd d; ../file)
 file = (cd d1; cd d2; ../../file)
```

A utility (ps(1)) is provided to determine the names and attributes of the processes which exist at any time. Curiously this facility is not provided directly as a system call (by analogy with `stat(2)' for

files). It is implemented in an eccentric way by searching a file '/dev/mem' (i.e. the main store), to find the appropriate tables within the kernel.

Shell Level Naming

The naming provided by the kernel and utilities is extended and modified by the shell to produce the view of UNIX typically seen when logged on. File names are modified by the use of abbreviations, command names are resolved using a path-list resolver (as described in Chapter 4) and the notion of 'executable file' is extended to include shell programs.

The following abbreviations are recognized in arguments given to to shell commands:

* Matches any string including the null string.

? Matches any single character.

[...] Matches any one of the characters enclosed. A pair of characters separated by '-' matches any character lexically between the pair.

These allow a file name to be partially specified

```
da* -> datafile
?akefile -> Makefile
```

Sets of names can similarly be abbreviated

```
file? -> file1 file2
*.c -> a.c b.c cc.c
ch[1-4] -> ch1 ch2 ch3 ch4
```

These abbreviations can be used in pathnames to provide searching:

```
../*/lib/cmd -> ../dir1/lib/cmd
```

Note that names are only expanded if the corresponding files exist: otherwise they are left unaltered by the shell.

The shell provides a programming language with its own (`built-in') commands and naming (the latter by the use of shell variables). In addition to the built-in commands, any utility can be invoked as a shell command:

command argument1 argument2 ...

Such command names are treated specially. They are resolved by a path-list resolver, which uses a list of directories stored in shell variable `PATH'. Each directory in the list `PATH' is searched in turn in an attempt to find the named command. Most utilities reside in directories `/bin' or `/usr/bin' so these are in the default PATH. The search is not applied to names containing the character `/' thus allowing explicit naming to override the PATH search:

./localcmd
(names a file in the working directory)

/user/dave/cmd
(names a file relative to `/')

Because `PATH' is an ordinary shell variable it can be modified in the usual way, so causing other directories to be searched instead of (or as well as) the standard ones.

Once a command name is resolved one of two actions is taken:

(a) If the file appears to be an object program it is executed as a sub-process.

(b) Otherwise, a sub-shell is generated to read and execute the commands it contains.

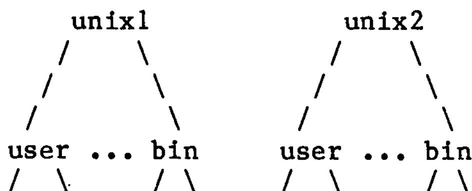
In this way, shell programs have the same status as compiled programs. Because only programs with 'execute permission' are ever executed, it is rare for a data file to be executed by mistake!

5.2 UNIX United

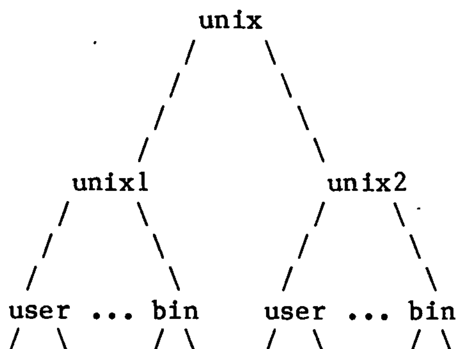
UNIX United extends the UNIX name space over multiple systems, without change to any of the UNIX naming mechanisms described above.

The aim is to provide an otherwise unaltered UNIX which extends across many actual systems. This is done by extending the file name space so that existing commands can manipulate both remote and local objects with equal ease. For processes, names are <u>mapped</u> (as described in Section 4.3) between the flat name space of each system in a way which conceals remote execution.

For files, which have recursive naming, the concept is simple, two separate file systems:



can be 'united' as sub-directories of a new directory:



Any file in the new 'united' system can be named using existing file

naming; no new commands or syntax are needed to deal with remote files.

File Naming in UNIX United

In the file naming tree, systems are incorporated into a global tree. Each is a part of the whole, rather than each viewing (i.e. naming) all others as subordinates. This generalized approach is only made possible by recursive naming of files: systems can be arbitrarily combined to produce a new larger system. In fact any directory in the UNIX United naming tree can be the root of a UNIX system, with arbitrary nesting of systems within systems.

Typical shell level usage of UNIX can be extended over multiple UNIX United systems. Consider a shell process with root (`/') at `unix1' and working directory at `/user/dave' (`cp' is the command to copy files, `ls' lists the names in a directory):

```
        cp ch1 ../archive
                         (local copy)

        ls /user/dave/thesis
                         (local directory search)

        cp ch1 /../unix2/user/dave/archive
                         (copy to remote system)

        ls /../unix2/user/bin
                         (search remote directory)
```

All the shell level naming facilities can be used in this way (shell variables are set by assignment of the form `name=string' and are named by the construct `$name'):

```
U2=/../unix1/user/dave
                  (set shell variable U2)

cd $U2
                  (change working directory
                   to /../unix1/user/dave)
ed text
                  (edit a file in that
                   remote directory)

cp text /../unix1/user/dave
                  (copy back to original system)
```

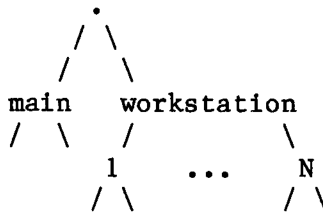
For instance, a new version of a utility program can be delivered to a set of systems using the shell built-in command `for' (e.g. `for x in list do c done'; the commands between `do' and `done' are executed with variable `x' set to each element of `list' in turn):

```
for SYS in unix1 unix2 unix3
do
        cp ed.new /../$SYS/bin
done
                  (copy ed.new to /../unix1,
                    /../unix2, /../unix3)
```

Similarly output and input can be re-directed to and from remote systems and `pipelines' can be set up across systems:

```
sort <a >b
                  (input and output re-direction
                   using `<' and `>':
                   sort takes input from `a' and sends
                   output to `b')

sort </../unix1/user/dave/f  >ff
                  (sort remote file to local one)



sort a | lp
                  (pipeline using `|': output from `sort'
                   becomes input to `lp')

sort $U2/file | $U3/bin/pr | lp
                  (sort a remote file,
                   format (pr) it remotely and
                   print (lp) it locally)
```

When there are a number of small computers ('work-stations') and a single large system ('main'), some services might be centralized, for example the receipt of 'electronic mail'.



The standard shell inspects the file named by variable 'MAIL' every time a command is completed, printing 'you have mail' when appropriate. A useful convention would be to receive all mail in a single 'mailbox' on the main system:

```
MAIL=/../main/usr/spool/mail/dave
                  (set watch on mailbox on 'main')
.
.
you have mail
                  (message from shell)
.
.
mail
                  (run mail program to read mailbox on 'main')
```

(To be exact, the program 'mail' is ignorant of the variable 'MAIL' so it must be explicitly forced to read the non-standard mailbox by using 'mail -f $MAIL'.)

The smaller systems might not be capable of executing some of the larger utility programs. These large programs could be held in a directory on the main system, be accessed transparently by changing the shell variable 'PATH' to cause a search of that directory:

```
PATH=$PATH:/../main/bigbin
                 (append to current PATH value,
                  so that /../main/bigbin will be
                  searched after the local /bin
                  directory)
.
.
nroff file
                 (use `nroff' on main machine
                  i.e /../main/bigbin/nroff)
```

The utility programs can manipulate local and remote objects. The UNIX editor (ed(1)) makes a copy of the file being edited in a local buffer; the `w' command writes the buffer onto the original file. When editing a remote file, the buffer and original file reside on different systems, every `w' saves the file onto the remote system.

```
ed /../unix2/thesis/ch4
                 (local edit of remote file)
.
.
w
                 (write out editor buffer to
                  remote file)
.
.
w ch4
                 (make a local copy of file)
```

In practice `w' is used frequently in a editing session to `checkpoint' work done so far; in the above example there is the added advantage that each `w' protects against a crash of the local machine as well as against the usual editing mishaps.

Process Naming in UNIX United

In UNIX United there is no change to the way processes are named: at the shell level a process name (PID) is returned (as usual) when an asynchronous process is started (by adding the suffix `&' to a command):

```
nroff file >file.lp&
234
                    (start a background process
                     sending output to `file.lp')
```

The shell prints `234' the PID of the process, and is then ready to continue. The same happens when the command is held on a remote system.

```
$U2/bin/nroff file >file.lp&
567
```

The PID `567' can be used to name the process

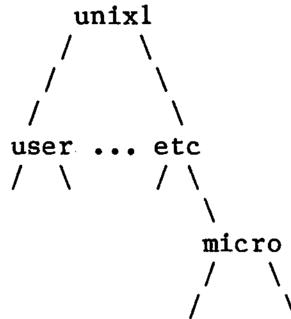
```
kill 567
                    (send an interrupt to process 567)
```

The Newcastle Connection software delivers the interrupt transparently to the remote system. Similarly, execution can be suspended until an asynchronous process terminates (using the usual `wait' command).

```
spell thesis > tmp &
890
                    (start process 890 and carry on
                     with independent work)
.
.
.
wait; ed tmp
                     (wait for 890 to finish,
                      then edit its output)
```

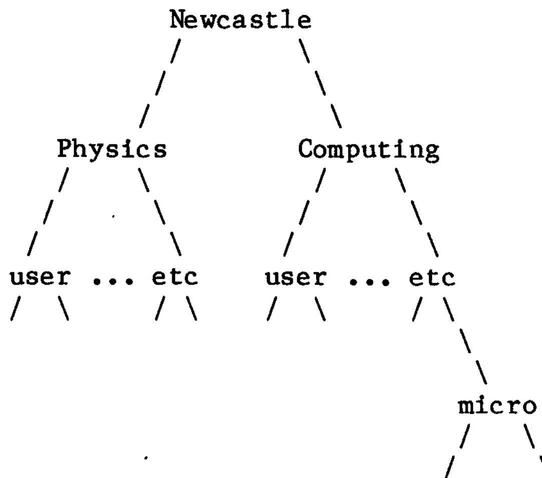
UNIX United Sub-Systems

In UNIX United, a system can include sub-systems. A sub-system is a system which appears within the file system of another.



In the diagram, '/etc/micro' might name a small UNIX system attached to the main system. The files in the '/etc/micro' directory (and its sub-directories) are a UNIX sub-system.

There is no need for the naming structure to correspond to the physical connection topology, although often physical locality will correspond to locality in the naming tree. A typical University system might be partitioned by department, with a main departmental system containing local sub-systems.

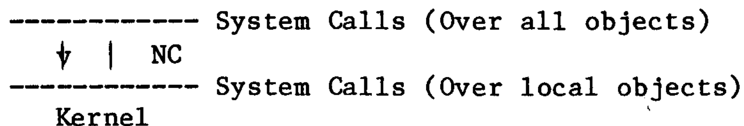


To conclude the UNIX United section, it should be mentioned that the UNIX United architecture and facilities described above have been implemented and extensively tested. The initial UNIX United system consisted of five UNIX systems contained in a single 'master' directory; this system was implemented by L.F. Marshall and myself. The full gen-

erality of UNIX United was achieved allowing commands like those in the examples above (the only restriction being that a system could not act as a `gateway' to another system). Any network errors are translated into the appropriate UNIX error message, hiding the existence of the network completely.

5.3 The Newcastle Connection

The Newcastle Connection is an interpreter extension of the UNIX kernel. It is a recursive extension, making the same set of system calls (and error reporting) apply to both local and remote objects.



Choice of Level

At what level should UNIX United be incorporated into UNIX naming? Recall the three tiers of naming - kernel, utilities and shell - each contributing to file and process naming in UNIX. There is a design decision to be made as to which level of naming should incorporate remote systems, which is partly independent of the level chosen to implement the design.

* UNIX United at the Shell Level

Shell level naming could be modified to include the extended UNIX United naming scheme. Unfortunately, this is difficult; although command names could be re-interpreted (to find references to remote commands), there is no way of knowing which arguments to a command are file names or process names. Given that UNIX United calls for a recursive extension of the file name space, it is neither acceptable nor transparent to add new

syntactical constructs to shell level names.

* UNIX United at the Utility Level

This level has been chosen by some designers of distributed UNIX systems. The 'uucp' (UNIX to UNIX copy) system consists of an extra set of utilities to perform inter-system accesses[Nowitz 1979]. Alternatively, existing utilities can be extended to deal with remote objects as well as local.

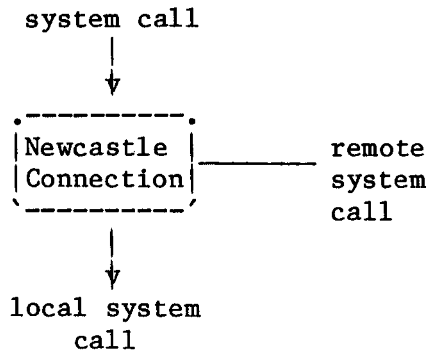
In the first (uucp) approach there are two or more versions of some commands (mail, cp, ...) and the right one must be used, depending on whether local or remote objects are being accessed. Also, other commands are not available over remote systems (e.g. ls(1)). It is also quite likely that the syntax of the two versions of a command will be radically different. The second approach requires modification to an arbitrary amount of existing software.

* UNIX United at the Kernel Level

Because all access to system objects passes through the kernel, it is a good level to introduce the UNIX United naming. If the kernel naming is extended, then both layers above will benefit. Unlike the shell commands, arguments to the kernel commands (i.e. to system calls) are of known type and can thus be processed correctly.

Implementing the Newcastle Connection

Operations on local objects are passed through to the local kernel; operations on remote objects are passed to the Newcastle Connection on the appropriate remote system



For each system call, the Connection has a `ghost' version which performs routing as shown above. Processes run as normal, their system calls are diverted to the `ghost' routine transparently.

At the heart of the Newcastle Connection is a set of routines for checking names. These are used to decide whether a system call relates to a local or remote object and, if remote, how to reach the remote system.

Each test routine returns a token, which is null for local names and non-null for non-local names. The token value is immaterial to the Connection software, it is either ignored (if null) or passed to the network software (if non-null). In fact the value indicates a network address (e.g. ring station and port number), which is used to reach the remote system.

File Names

The routine for checking file names interprets a pathname within the local context to find whether it is local or remote. In the case of remote files, the `tail' of the pathname (a result parameter of `check') will be passed to the remote system. The `tail' is that part of a pathname which cannot be resolved locally because the relevant file or directory is not present. Instead, it is passed to the Newcastle Connection on the appropriate remote system.

When a system call refers to a remote object, the name-checking routine will return a token to identify a remote system. This is packaged along with the arguments to the system call and a number which indicates the particular call being made, and passed to the remote procedure call (RPC) software. A typical `ghost' system call routine looks like this:

```
open(name,mode) {
    token t=check(name,tail);
    if (token==NULL)
        return(open(name,mode));
    else
        return(rmt_open(t,tail,mode));
}
```

The RPC expects two parameters: the message (an unstructured string of characters) and the destination. For example when a remote file is to be opened the name of the file and the required `mode' (read,write, etc.) is passed to the routine `rmt_open' which packages them up and makes the remote procedure call (routine RPC):

```
rmt_open(token,name,mode)
{   string msg="";

    append(msg,OPEN);   /* msg = OPEN */
    append(msg,mode);   /* msg = OPEN mode */
    append(msg,name);   /* msg = OPEN mode name */

    RPC(token,msg);     /* send message */
}
```

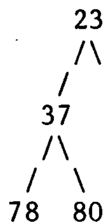
Process Names

The routine for process names consults the mapping table and takes the appropriate action: local access, or remote access to a mapped process name.

Unfortunately, in UNIX, only files have recursive naming; process naming is flat. There is no way of naming processes on another system because such a syntax (e.g. process@system) is alien to UNIX. To get

around this problem in the Newcastle Connection, process names are <u>mapped</u> as they are passed from system to system, preserving their locality. In effect, on any given system there is the illusion that all processes are local, because only one local set of process names is used. Non-local processes are named by using a local name which is transparently mapped into the appropriate remote name (see Chapter 4 for a description of name mapping).

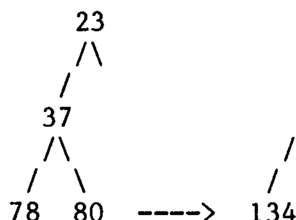
Mappings are set up when remote execution occurs. In UNIX United it was decided that files should be executed by the system where they reside. This is a reasonable default because executable files may not be compatible between systems. When a remote file is named for execution a new entry is added to the mapping table, enabling the resulting process to be named. In this UNIX United process hierarchy:



the files being executed appear to be

Process	File Being Executed
23	/bin/sh
37	/usr/bin/make
78	/etc/bold
80	/../unix2/bin/nroff

when in fact the actual hierarchy (hidden by the Newcastle Connection) is:



Process	File Being Executed (on unix1)	
23	/bin/sh	
37	/usr/bin/make	
78	/etc/bold	
80	/bin/unix_server	(to intercept signals)

Process	File Being Executed (on unix2)
134	/bin/nroff

where the `unix_server' is the local representative of the remote process.

5.4 Some Other Distributed UNIX Systems

The general idea of building a distributed system by extending UNIX over multiple computers is by no means unique to UNIX United. Other authors have recognized the advantages of distributed UNIX but many of the published systems are rather *ad hoc*. The twin ideas emphasised in this chapter, no change to the UNIX design plus no change to the UNIX implementation, are the hallmark of our system and are not matched in any other.

Historically, the first well-known distributed UNIX system came with UNIX Version 7 and involved the `UNIX to UNIX copy' program (uucp(1))[Nowitz 1979]. This enabled files on other systems (connected by serial lines) to be named by using the syntax

systemname!pathname

in the arguments to `uucp(1)' (for copying) and to `uux(1)' (for remote execution). The `systemname' is drawn from the set of systems known to the local `uucp(1)' (or `uux(1)') command (a null systemname refers to the local system). Using these commands a distributed set of UNIX systems can be accessed, although at the price of special commands using special syntax. Despite being cumbersome, uucp has the advantage of

being part of standard UNIX and thus widely available. The implementation of uucp is fairly restricted; for example remote execution is restricted to one site at a time.

The next stage of development in distributed UNIX is to use the existing naming without change. The Network UNIX System[Chesson 1975] and the Purdue Engineering Computer Network[Hwang et al. 1981] provide special commands for inter-machine access but they do use what appear to be standard names for files. The main difference between them is that the former system uses the ARPANET whilst the latter uses hard-wired high-speed connections. (The Purdue system includes an element of load balancing in selecting which processor should execute certain commands.)

Another system based on high speed local communication (using the Datakit switch) divides computing resources into `processing workers' (S-UNIX) and `file system workers' (F-UNIX)[Luderer et al. 1981]. The idea is that a user may use any S-UNIX by `attaching' it to his files stored in an F-UNIX system. This separates the notion of `current UNIX system' from that of `current computer': no files are permanently stored on S-UNIX systems. Each user sees a perfectly ordinary UNIX system although its implementation is in fact shared between the two classes of UNIX systems. Transparent file sharing and file replication is provided by the F-UNIX systems; similarly any process can use any of the available S-UNIX workers.

The LOCUS system provides a symmetric linking of UNIX systems (in contrast to the S-UNIX/F-UNIX dichotomy). LOCUS allows standard commands to access remote files named by standard UNIX naming[Popek et al. 1981]. Network transparency is provided at the expense of a completely re-written system. Not only are the necessary networking features added but also file replication and version numbering; the latter two adding complexity to UNIX. In many ways LOCUS is the same as the UNIX

United/Newcastle Connection system; however being a complete system LOCUS must be installed as a replacement, rather than a supplement to an existing UNIX system.

Finally, the COCANET system[Rowe and Birman 1982] is also very similar to our work. COCANET is implemented by extending the UNIX kernel to include networking. The main difference is that there is no uniform distributed file system; each COCANET system views the others as entries in its '/' directory. The convention is to use

/unix2

to name remote site 'unix2', rather than to use

/../unix2

as in UNIX United. This means that each system has a unique view of the file system with itself as root, rather than it being a component of a larger system.

To conclude this brief survey it must be said that all these systems came to our notice after UNIX United was designed and running. Looking back on the literature it can be seen that some systems (e.g. COCANET, LOCUS) are fairly similar to our design, whilst others, perhaps for pragmatic reasons, are less than transparent. The combination of complete transparency plus no re-writing of UNIX is the unique feature of the UNIX United plus Newcastle Connection system.

Review

Why UNIX? To a large extent the answer to this question should already be apparent. UNIX has provided a suitable base for a distributed system because of its recursive naming of files and the ease of inserting the Newcastle Connection as a new layer. In our original UNIX United

paper[Brownbridge et al. 1982] six principle factors in favour of UNIX were listed:

(1) Recursive file (and device) naming.

(2) Ability to change working and root directories.

(3) Ability to initiate asynchronous processes; this is used within the Newcastle Connection software and also enables real concurrency to be set up when using the Newcastle Connection.

(4) The kernel (system call) interface is fairly simple and strongly isolates the kernel data structures.

(5) UNIX is written in a high level language making it easy to incorporate the Connection into existing programs viâ the standard subroutine library. (System calls have the same syntax as other function calls.)

(6) Exception reporting for system calls is simple and can be used to mask network errors by presenting them in terms of the ordinary failure conditions.

The only post-script which I will add is that UNIX is also suitable for a distributed system because of its wide availability, both in the large number of UNIX systems in use and in the wide range of computers for which UNIX is available.

Why Not UNIX?

There are some simple improvements to UNIX which may find their way into some of he newer (post-Version 7) editions of UNIX.

* A fuller implementation of `links' (i.e. bindings made to a file subsequent to its creation). At present, links can only be made within

one storage device.

* Better inter-process communication (e.g. named pipes).

* A re-think of the file protection system (e.g. to allow `effective user-id' to be set explicitly by a root-owned process.)

Later (in Chapter 7) I will examine more fundamental changes to UNIX, pointing the way to new non-UNIX systems of the future.

* * * * *

UNIX United shows the advantages of recursive naming: systems can be combined to form a coherent distributed system. File naming in UNIX is sufficiently rich for no extra mechanism to be needed in UNIX United; the same system calls are adequate for remote as well as local files.

Although the work on recursive naming (Chapter 4) grew out of the UNIX United work, rather than the other way round, they are presented in the opposite, more logical order in these two chapters. Recursive naming in UNIX ensures that the UNIX United architecture is very general and allows system to be nested within system to any depth.

The Newcastle Connection implementation shows the advantage of a recursive interpreter extension: it can be added to <u>any</u> system which looks like UNIX regardless of the implementation. Neither are changes needed to the UNIX programs using the Connection. No changes were made to the underlying kernel implementation or to the system call interface. As a recursive interpreter extension, the Newcastle Connection adds value to UNIX without changing it by allowing UNIX United to extend over many computers.

Many UNIX United systems have subsequently come into use based on our original software. At Newcastle, further UNIX United work has

included a version for the Three Rivers/ICL PERQ computer using Ethernet and also the development of a Newcastle Connection-based terminal concentrator. Further research work by others has used the Newcastle Connection as a basis for a distributed fault-tolerant system using majority-voting and as a basis for a highly secure UNIX system. Elsewhere, the Newcastle Connection software has been used in a variety of situations in both commercial use and as part of research into distributed systems.

Finally, it should be obvious that the work in this chapter depends crucially on the design of UNIX itself. UNIX United demonstrates that we can go a long way toward a true distributed system within the original UNIX design. This is a tribute to the generality of that original design. Later (in Chapter 7) some suggestions are made on the design of future distributed systems not closely based on UNIX.

Chapter 6

Computation and Recursive Computation

> There are those who believe that God gave FORTRAN and our job is simply to make it go faster
>
> [Anon.]

Computation is the activity which takes place as a computer executes a program. The present chapter will examine the structure of computation - that is the patterns of activity which arise when various computation rules are used.

Most computers use the sequential, instruction-at-a-time 'Von Neumann' computation rule. Programs are a linear array of instructions and data. Control traces a linear path interspersed with jumps and data may be accessed randomly. Such computation can be characterized by two structures, one representing the pattern of control and the other the pattern of access to data. Although in principle random, the patterns are dictated by the programmer and by using the techniques of 'structured programming' some locality is achieved (e.g. iteration over short loops and use of local variables).

Other computation rules give rise to different structures, for instance when there are many threads of control active at once. There are computation rules giving rise to recursive structures, representing a 'divide-and-conquer' approach to execution. It is naturally this

latter possibility which is of particular interest here and will be shown below to have advantages over non-recursive computation.

Computer structure has received considerable attention in the literature[Bell and Newell 1971, Stone 1975, Tanenbaum 1984] but much less has been written about the structure of computation. Computer structure concerns the design of computers whereas computation structure considers the structure of activity when computers execute programs.

The present chapter is based on work for an earlier paper which separates computation structure from the structure of programs and of computers giving a classification scheme for each[Treleaven et al. 1982]. The three sections of the chapter firstly characterize computation as a particular abstraction for the activity of computer systems (Section 6.1), then examine some models of computation (Section 6.2) and finally, explore the notion of recursively structured computation (Section 6.3).

6.1 Abstraction: Computation, Architecture, Hardware

The study of computation examines how instructions are chosen for execution and the range of effect that execution can have. It can be distinguished from architecture (program organization) and hardware (machine organization), which are separate aspects of computer design. It will be shown that the choice of computation, program and machine organizations can be made independently, even though some combinations are more attractive than others. Many different machine organizations may be used to implement the same architecture; many architectures can use the same computation organization. This separation of levels was a key idea in our paper[Treleaven et al. 1982].

Computation Organization

Computation is an abstraction of the activity within a computer when a program is executed. Computation organization describes state changes in programs (and their data). A computation rule guides the course of computation in a given computer. We will be considering the class of computation that arises in a given computer, not the particular computation arising from any single program being executed. Two aspects, sequencing and locality, are of particular interest when examining computation organization.

Sequencing is the choice of next instruction(s) to execute. Computation organization is only concerned with the 'external' behaviour of the computer so a computation rule specifies sequencing in terms of programs. Operations within the computer are of no concern.

Locality of computation is a measure of the 'distance' between elements in the computation structure. That is, the distance between the places in store in which successive elements are held. In Von Neumann computation there are two forms of locality to be considered: locality of control and locality of data reference. Locality of control measures the distance in store between successively-executed instructions; locality of data reference measures the distance in store between successively-accessed data items.

Locality can also be measured by counting how many times each instruction is executed (or how many times each data item is accessed). If the program exhibits strong locality then, say, 10 percent of the instructions in store might represent 90 percent of the instruction executions. (Or similarly 90 percent of data accesses might be to 10 percent of the data).

It may also be useful to consider whether the 'preferred' (most

used) items are themselves situated close together. In many systems this does not matter because the set of preferred items is determined dynamically (e.g the `working set' in a paging system[Denning 1968]). However if the preferred items can be found reliably before a program is executed, they can be located in adjacent parts of the store avoiding the dynamic overhead.

In practice, all but the simplest computers need strong locality of computation to get sufficiently good performance. For instance, the machine-level optimizations of paging pipelining and cacheing are useless unless computation is localized.

The programming language provided by a computer may support explicit control of computation e.g. JUMP instructions. Here the pattern of computation can be controlled directly by program. On the other hand, where there are no control constructs, the pattern of computation is entirely implicit. In data driven and demand driven computation (as described below) there is no distinction between control structure and data structure: computation can be modelled by a single structure. It is then sufficient to examine this single form of locality.

Later (in Section 6.2) control driven, data driven and demand driven computation will be described in more detail.

Program Organization

Usually, program organization is referred to as `computer architecture'. This is a term which unfortunately also includes machine organization, with the two separate issues being confused under one heading. Program organization is typically specified by a `Principles of Operation' manual. For example the `IBM S/370 Principles of Operation' specifies a program organization (i.e. the interface) of huge number of computers implemented by very many different machine organizations.

In fact a noticeable commercial trend has been to build 'ranges' (or 'series') of computers, offering some compatibility of program organization over a wide spectrum of machine organization. The differences between models in a series can be in more than just speed and storage capacity - even the size of the hardware registers and data paths (the ALU width) can vary[Tanenbaum 1984]:

S/370 Model	ALU Width (bits)
370/115	8
370/125	16
370/145	32
370/168	64

Similar differences occurred in the ICL 1900 series: in smaller models, 'registers' are in fact the first few words of the main store, whereas larger models are graced with the more usual implementation of registers.

Machine Organization

Machine organization is the structure of the hardware (or sometimes the software interpreter) which forms the computer. Details of machine organization are hidden 'below' the programming language interface.

A machine organization may be a simple, direct implementation of the machine-language or, as is more usual, the organization will be designed to give good performance for a given class of programs. The problem with general purpose computers, where the language but not the program is known in advance, is to make an educated guess as to the most likely class of programs. As mentioned above, modern Von Neumann computers work fastest when there is strong locality of computation, although this locality is by no means explicit in their model of computation.

6.2 Classification/Analysis: Computation Organization

Three computation organizations will now be examined in greater detail:

* Control Driven (including Von Neumann)
* Data Driven
* Demand Driven

The terms `instruction' and `operator' will be used without distinction to denote the functions available in a computer; similarly `arguments' and `operands' will be used interchangeably.

Control Driven Computation

Computation is said to be <u>control driven</u> if sequencing is controlled explicitly by program. This implies that (at least some) instructions can define the instruction(s) to be executed next.

The simplest form of control driven computation is the Von Neumann model, which has as its main characteristics[Von Neumann 1946]:

(1) A single computing element [processor] with sequential control of computation.

(2) Linearly addressed array of fixed-size storage cells.

(3) Low-level language performing simple operations on unstructured data.

Von Neumann computation is control driven because programs have explicit (sequential) control over the order of execution. This is by means of a `program counter' register containing the address of the next instruction. This register can be manipulated by programs, (e.g. JUMP, or subroutine CALL instructions), thus altering the flow of control. (A less

common form is the four-address instruction, where each instruction contains the address of its successor.)

Parallel control driven computation allows many 'threads' of control to be active at once. Communication between different threads of control can be through the common store. New threads of control are generated explicitly (e.g. by a FORK instruction) and synchronization is also explicit (e.g. by a WAIT instruction).

The essence of control driven computation - explicit choice of next instruction(s) - does not enforce locality of computation: locality is entirely at the programmer's discretion. Neither does it enforce strong locality on operands (data access); it is up to the programmer to ensure that all operand values necessary for an instruction have been calculated before it is executed. For example:

a:=1; c:=a+b; b:=2

The above trivial sequential program serves to illustrate an object 'b' being used as an operand before its value '2' is assigned. Once control reaches an instruction, that instruction is executed regardless of the state of its operands. The problem is not just <u>initialization</u>: throughout the computation an object may have many different values assigned to it. The programmer has to ensure that the <u>current</u> value is as required, each time it is used.

The Von Neumann model has a single linear store, but there are other sequential control driven systems which do not. For example Iliffe's Basic Machine[Iliffe 1972] and the FLEX architecture[Currie et al. 1981] are control driven, but have a (recursively) structured store. Similarly some parallel control driven systems do not rely on a single shared store.

Data Driven Computation

Computation is said to be data driven if sequencing is controlled by the availability of data; instructions are executed when their operand (data) values have been calculated[Dennis 1979].

In data driven computation, sequencing depends on the state of operands, since there is no explicit control. When an instruction is executed it produces a value which may be an operand to other instructions. Any number of those instructions may become executable at that point.

The structure of any data driven computation is a di-graph, ordered by the data dependency relation between instructions. The nodes of the graph are the instructions of a program; the arcs of a graph represent the use of a value produced by one instruction in a another instruction.

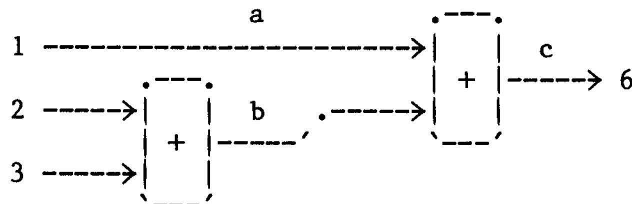
```
Data dependency graph = (I,D)

Where  I = {instructions}
       D = {(i,j) such that: instruction j
             uses a value produced by instruction i}
```

Locality is determined by the data dependency graph. The next instruction(s) to be executed are always those dependent on the current instruction(s) for their data. It might be expected that this strong locality would be exploited in program and machine organizations; however, it is not. (The exception is DDM/1[Davis 1978] which is described below in Chapter 7.)

Data driven computation is innermost (parallel) first (in the sense used by Manna[Manna 1974]). No instruction is executed until all its operand values have been calculated. For example the calculation of 'c' in:

```
a:=1
c:=a+b
b:=2+3
```

is governed solely by the data dependency graph:



First 2+3 must be calculated and then, 1+5 which yields 6. The phrase 'innermost first' signifies that the instructions <u>most deeply nested</u> in the graph are first to be executed.

Demand Driven Computation

Computation is said to be <u>demand driven</u> if sequencing is controlled by the <u>need</u> for values; instructions are executed when the value they calculate is demanded by another instruction.

In demand driven computation, sequencing depends on the need for results - there is no explicit control. When control passes to an instruction, it may cause other (sub)instructions to be executed before the instruction itself is executed. Demand driven computation allows instructions to be <u>non-strict</u>. Non-strict instructions can return values even when some of their operands are unevaluated. One example is the conditional operator 'COND', whose evaluation is defined by:

```
COND(TRUE,a,b)  ::= a
COND(FALSE,a,b) ::= b
```

A 'COND' can be executed as soon as the first operand is a value; the others need not (yet) be values; they may be sub-programs delivering a value. Demanding the result of a COND causes its first argument to be

evaluated.

The structure of demand driven computation reflects the data dependency between instructions in the same way as that of data driven computation. The difference is that demand driven computation is top-down (i.e outermost first) and non-strict, whereas data driven is bottom-up (i.e. innermost first) and strict. Because of the non-strictness of demand-driven computation, not all parts of a program will be executed before termination, some may be `thrown away' by (for example) the COND operator given above:

COND(2>1, 0, f(0)) where f(x)=f(f(x))

COND(TRUE, 0, f(0)) where f(x)=f(f(x))

0

In this example the non-terminating computation of `f(0)' is discarded and result `0' delivered.

Comparison of Data Driven and Demand Driven Computation

Although the locality properties of data driven and demand driven computation are similar, demand driven has significant efficiency advantages.

Data driven computation, being strict, waits until all sub-computations have been terminated. For example, if the function

f(x) = 3

which takes a single argument and returns a constant, is applied to a non-terminating argument (e.g. the function f(y) = f(f(y))), no value is ever returned. D.A. Turner aptly christened such non-terminating arguments `black holes'. Demand driven computation avoids black holes wherever possible by only waiting for values that are actually needed. Data driven computation is the converse: a black hole anywhere will envelop

the whole program!

This discussion of efficiency properties has its parallels in the theory of the Lambda Calculus[Burge 1975] and of Recursive Functions[Manna 1974, Brady 1977].

Concept	Lambda Calculus Equivalent	Recursive Function Equivalent
Program	Lambda-expression	Recursive Function
Data Driven Computation	Applicative Order Evaluation	Innermost [Parallel] Substitution
Demand Driven Computation	Normal Order Evaluation	Outermost [Parallel] Substitution
'Efficiency'	Church-Rosser Theorem	'Safety' of Outermost Substitution

The 'efficiency' property of demand driven computation can be paraphrased as

'Demand driven computation will always find the right answer if one exists.'

The equivalent Church-Rosser theorem states a similar property for normal order evaluation. The equivalent property of recursive functions is 'safety', a property possessed by outermost parallel substitution. Other substitution rules are also 'safe' and have corresponding models of computation e.g. 'eager evaluation' and 'lazy evaluation'[Henderson and Morris 1976, Vuillemin 1974]; these can be regarded as straightforward extensions of the demand driven model.

This discussion would not be complete without mention of another aspect of the Church-Rosser theorem which has been misleadingly quoted

(sometimes by those who should know better!) as

'No matter the evaluation order, the result is the same.'

This is only half the story; termination must be considered. In fact the theorem states that if two evaluators terminate they terminate with the same result. Data and demand driven computation are equivalent in this sense, but then so they are to a 'black hole' evaluator (e.g. function 'g' above), which swallows all programs, for them never to be seen again!

The lesson to be drawn from these parallels is that parts of the theory of computation can be used to provide a sound base for system design.

6.3 Recursive Computation

Recursive computation is the result of combining recursive control with a recursive program[Glushkov et al. 1974, Treleaven and Hopkins 1982, Wilner 1980].

To be precise, recursively structured computation consists of a recursively structured computation rule combined with a recursively structured program. The advantages are those of recursive structure in general: extensibility, location independence and ease of extension and combination.

Recursive Program Structure

Recursive program structure is simply recursive naming of operators and of operands in programs.

Recursive programs are characterized by the following property:

Wherever in the program a operator or operand can appear, a

sub-program (denoting a operator or operand) can also appear.

Hence recursive programs are recursive structures (as described in Chapter 2), where each atom is a fundamental instruction or data item, and each sub-structure is a sub-program. The S-expressions in LISP are recursive programs[McCarthy 1962], as are blocks and expressions in ALGOL 68 (but not statements: loops do not return a value).

Where there is no nesting or a limited depth of nesting, programs are not recursively structured. For example, instructions may have the non-recursive form:

instruction ::= atomic_operator atomic_operand ...

which characterizes 'single-address' Von Neumann computers. In such systems, a sub-program may appear as an argument (by reference) but de-referencing only involves address calculation and fetching, not a sub-computation.

These systems do not provide recursive computation because the program structure is not recursive. Recursive computation can, however, be simulated, by an interpreter extension which takes the appropriate actions automatically. This is what happens when recursively structured LISP programs are executed on a conventional machine. Later (in Section 7.3) a new algorithm is given for managing structures in this situation.

Recursive Control

Recursive control means having a recursive computation rule.

A recursive computation rule can be used to execute any arbitrary sub-program; it does not need a 'whole' program nor does it depend on the program having a particular structure: it is sufficient for programs to be recursively structured.

Both data driven and demand driven computation have recursive control, Von Neumann computation does not. In data driven computation, evaluation is innermost (bottom-up); in demand driven computation it is outermost (top-down), but the pattern traced is the same: both rules follow the data dependency graph.

The differences in termination of data and demand driven computation are computationally important, but not structurally important. The same (recursive) computation structure can be built by either top-down or bottom-up construction. The remaining differences between the two models are:

* Strictness: data driven execution depends on values being available, demand driven does not.

* Demand propagation: demand driven execution first propagates demands, then requested values return by availability; in data driven all instructions are implicitly `demanded'.

Recursive control promotes locality of computation, whatever the exact computation rule used.

Combining Programs

It is important to be able to combine and re-use existing programs. Recursive computation is an important tool for doing this.

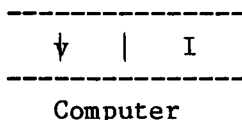
The advantages of recursive computation are as follows:

* Wherever a data value can appear, a program generating a value can appear. Wherever an instruction can appear, a program implementing a (new) instruction can appear.

* There is no distinction between inter-program and intra-program control. The same model of computation applies to composite systems

built from many parts and within a single program.

Recursive computation is useful when a multi-level system is built as a series of interpreter extensions (see Chapter 3). It is much easier to construct the extension `I' because the underlying computer treats the added instructions in the same way as its own instructions.



Then, at least syntactically, there is no difference between instructions from the original set and its extension.

Some Examples of Recursive Computation

(i) Blocks and Expressions in ALGOL 68

These are recursive program structures - wherever a value can appear in a block or expression, a block or expression delivering a value may appear (I prefer the more usual terminology to the ALGOL 68 Report `paranotations').

ALGOL 68 has a recursive computation rule which explores sub-expressions and sub-blocks in a demand driven fashion (other parts of ALGOL 68 are, however, control driven; also, loops do not return any result). Consider an ALGOL 68 example showing an expression containing a value, a sub-expression and a sub-block:

```
x := 1 + (2*3) + (int t:=y+z; t*t )
```

The expression is recursively evaluated to produce `6' and the block recursively evaluated returning value `t*t'.

(ii) LISP

A LISP program is a recursive structure: an S-expression for example:

```
(ADD (QUOTE 1)
    (ADD (MUL (QUOTE 2) (QUOTE 3))
        (LAMBDA t (MULTIPLY t t) (ADD y z))))
```

which represents the same calculation as the ALGOL 68 program above

Computation is recursively structured in a LISP evaluator (disregarding certain non-applicative features). LISP systems can have a form of demand driven computation, typified by the evaluation rule for COND (as shown above). The first argument of the conditional operator is evaluated and its value used to choose which of the other arguments is to be returned.

* * * * *

This chapter has characterized computation as an important issue in computer system design and argued that recursive models of computation offer viable and interesting alternatives to the so-called Von Neumann model. Recursive computation has particular advantages of locality and extensibility making it particularly suitable for large highly-parallel systems. In such systems it is vital to have locality of computation and to be able to combine or extend systems without re-structuring them.

Programs should not be regarded as 'black boxes' which give rise to random patterns of computation when executed. Instead their structure and locality of computation should be taken into account when designing models of computation. If a program is executed for long enough, it is indeed likely that every instruction and every data item will be referenced. On the other hand it is also likely that a small number of

instructions will account for the majority of instruction executions and similarly a small number of data items will account for most of the data accesses.

The view of computation presented here concentrates on the differences between control driven computation on the one hand, and data or demand driven computation on the other. The difference is important because in distributed systems some particular features of control driven computation (e.g. random access to shared objects and arbitrary transfer of control) become acute problems of communication and synchronization. Data or demand driven computation offers an alternative in which many of these problems disappear.

The advantages of recursive computation are similar to those of recursive naming. Recursive computation makes it easier for many processors to cooperate in executing a single program. If computation is not recursive there is a point where it is no longer possible to subdivide work amongst a homogeneous set of computers and a point where sets of computers can no longer be combined into a co-operative system.

In Chapter 7 some suggestions are made as to how the `novel' data driven and demand driven forms of computation might fit into future large-scale distributed systems. At this level where inter-processor communication is time consuming it is most important to have the kind of locality inherent in recursive computation. Also in Chapter 7 are descriptions of some computer systems which use recursive computation.

Chapter 7

Recursive Computer Systems

> Were these things here that we do speak about
> Or have we eaten the insane **root**
> that takes reason prisoner?
>
> [Macbeth I.ii]

The first section of this chapter will examine earlier work on recursive systems. This survey briefly examines the small number of papers using recursion as a theme for computer system design. The first[Glushkov et al. 1974] sets out principles of recursive machines as an alternative to more conventional designs. The second[Davis 1978] adds data driven computation and is closely related to the work of Wilner described in the succeeding section[Wilner 1980] which aims to exploit VLSI technology. Finally some of the work by others at Newcastle is presented to complete the survey[Treleaven and Hopkins 1982].

The second section concerns the implications of the UNIX United/Newcastle Connection scheme for the design of future operating systems. The exact nature of a 'UNIX System' is discussed and some difficulties with UNIX terminology are explained. It is suggested that processes and 'users' could benefit from recursive naming, that a new hybrid plain-file-plus-directory object would aid modular programming; Combined naming (introduced in Section 4.2) and also that data or demand

driven execution could be a part of future systems.

Finally a method is suggested for dealing with cyclic (loop-containing) structures which can arise in the implementation of recursion. This is a reference count algorithm able to manage safely the creation and deletion of some cyclic structures. It has an overhead proportional to the number of elements in a cycle. This is an improvement over the mark-scan technique which has an overhead proportional to the total number of elements and the standard reference count method which cannot deal with any cyclic structures.

7.1 Future Operating Systems

On the Concept of 'A UNIX System'

It will be worthwhile to consider just what is a 'UNIX System'. The possibility of more than one UNIX system per computer makes this an important question. I will attempt an answer based on the ideas used in UNIX United.

A UNIX system can be defined in naming terms as consisting of three elements:

* A set of files with a root directory, forming a tree-like name space

* A set of processes in a flat process name space

* A set of user-ids (and group-ids) in flat name spaces.

The file system forms the store of the UNIX virtual machine and the set of processes form its (multi) processor; user-ids are attributes of files and of processes which are used for the purpose of protection.

A UNIX file system must have a 'root directory' which contains the various standard system files defined in the UNIX documentation (e.g.

the directories /bin, /dev and their contents). Note that this `root' is defined in file system terms, independent of the processes. Here one starts to encounter problems with UNIX terminology. The `root' associated with a process (i.e. the directory named by `/') is not necessarily the `root' of the file system in the sense just described; neither is it necessarily the `root' in the graph-theoretic sense of `a directory unnested in others' (the `topmost' or `base' directory). However in the vast majority of UNIX systems the file-system root is indeed the topmost directory and also every process has that directory as its `/' root.

Only whilst UNIX United was being designed did the nature of these roots become clear. In UNIX United the conventional usage of the roots is altered: our use of `/..' implies that a root directory can be nested within another and secondly, the (little-known) change root system call (chroot(2)) allows the `/' of a process to be moved at will. Hence the three roots are no longer the same. In some systems this is used to partition a single computer into two UNIX systems, for example on a VAX computer where there can be separate PDP-11 compatible and full-blown VAX sub-systems running on one machine. This allows separate development of the two classes of program.

I would argue that for each process, `/' corresponds to `current system' in the same way that the working directory corresponds to the `current workspace'. The standard files described in the UNIX documentation are all named relative to `/', including nearly all the utility programs and their data files: `/bin/sh', `/bin/mail', etc.

The corollary is that `chroot' is too general: it should only be possible to move `/' to directories which contain the requisite system files (or at least, no others in their place). By no means the least reason is that system security could easily be compromised if `/' can be moved arbitrarily. For example an arbitrary file can be substituted for

the password file (/etc/passwd) allowing log-on as any user. It is possible to preserve security, by ensuring that only certain directories can be used as '/'. In this way the file names beginning with '/' cannot be used to compromise security, and the 'change root' command becomes, effectively, a 'change system' command.

An alternative approach to the 'chroot' problem is used in System V. Naming is restricted so that no matter where '/' is, the directory '/..' does not allow access to more files. That is, once '/' is moved to a directory, only the sub-directories of that directory are nameable (which at least localizes the effect of a security breach). The name '/..' is defined to be identical to the name '/' for all processes, no matter which directory they have bound to '/'. In other words the UNIX United approach uses '/..' to unite systems and provides secure translation of names between them; whereas the System V approach uses '/..' to separate systems.

The UNIX United approach to 'chroot' exploits the complete independence of '/' from the working directory. For example '/' can be moved to another, possibly remote, system whilst the working directory remains local. The UNIX United approach requires more sophisticated support (for mapping), but provides secure co-operation between an arbitrarily large set of UNIX systems. The System V approach is simple but precludes multiple co-operating systems.

Each UNIX object (file or process) has an associated 'user' and 'group' identifier. These are used for protection and have a flat name space. It seems reasonable to associate one such space with each UNIX system giving its own set of users and groups. These identifiers can then be mapped between systems, in the same way as process identifiers.

There are further extensive issues of user-id and group-id mapping

in UNIX United concerning nested sub-systems etc. Suffice it to say that there are two contrasting philosophies: (1) each system allows access to named users of other systems by special arrangement; (2) a systematic mapping policy is initiated allowing the mapping to be transitive, symmetric etc.

Subsidiary meanings of `root' in UNIX which are not considered here include: (1) root - super-user - the user name corresponding to user-id zero with special privileges; (2) root - `root device' - the device onto which all other file storage devices are mounted (mount(1)).

Distributed Operating Systems: Future Work

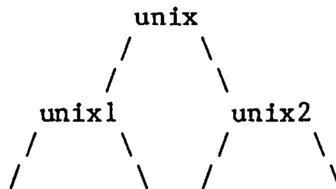
Dennis Ritchie (one of the originators of UNIX) has said that UNIX may well be the end of one thread of operating system development. Indeed some newer systems (e.g. the Smalltalk environment[Goldberg and Robson 1983]) are very dissimilar to the current concept of an `operating system'. The comments which follow take UNIX as a starting point for evolutionary development towards what might be termed `new generation' systems. The ideas are not intended to describe an `update path' for UNIX but rather to indicate scope for developing new systems.

(1) Recursive naming is the essential feature of UNIX which makes UNIX United possible. It would be a great improvement if all objects in UNIX were to have recursive naming, especially processes and `users'. For processes this would add the ability to explicitly name remote processes (e.g. a remote print server process) instead of relying on the implicit Newcastle Connection mapping.

Recursive naming for users could perhaps use block structured naming, thereby imposing a discipline on user-id mapping between systems. This mapping is done on an arbitrary `need to know' basis at present. The UNIX systems in a UNIX United naming tree can be viewed as sub-

directories contained in the outer `network' directory. User names for purely local users are declared local to each system; user names declared in the outer block name the users who have access to both systems.

Consider a UNIX United file system:



where the scope of user names could be represented as:

```
unix:begin
        user a,b;

    unix1:begin
            user c,d;

        end;

    unix2:begin
            user c,d;


        end
    end
```

The <u>scope</u> of a user name can define the user's ability to own files and processes. This notion extends and regiments the ideas for name mapping introduced in Section 4.3 and and used in Section 5.3. No change would be needed to UNIX, this scheme is simply a <u>discipline</u> on the way user mappings are set up between systems.

(2) All names in UNIX are dynamically resolved. In a single UNIX system where all processes have `/' bound to the same directory, this is not a significant problem. Static resolving can be simulated (when necessary) by using root-relative pathnames. An example of a program requiring

static resolving is a print spooler which has the name of the actual printer (a UNIX `file') built into it; the spooler might be invoked from anywhere in the file system but must still use the same printer. In UNIX United (or any other UNIX system where `chroot(2)' is used) a root-relative pathname is not unique; it is then not possible to construct the spooler as required. It would therefore be useful to add static resolving to UNIX.

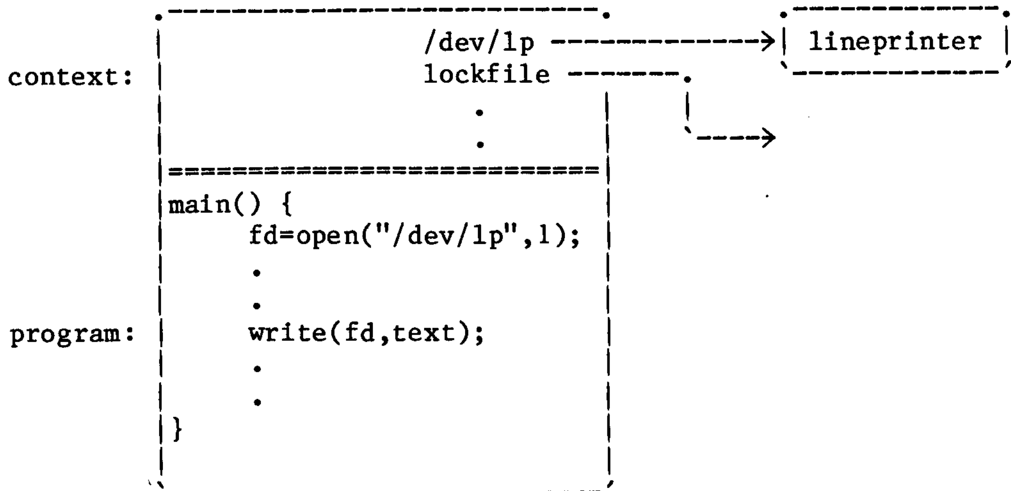
With static resolving, the names within a program refer to the same object regardless of where and when the program is run. A preliminary study indicates that there is no simple way of statically resolving filenames. Many UNIX programs are written in `C' or the `shell' language. Both these languages are `weakly typed' in the sense that any dynamically created string of characters can be used to name a file. The idea of adding a special filename type to `C', the shell and every other language on UNIX (and its run time system) is not appealing.

The solution I suggest is to introduce a new hybrid class of file which is a single object consisting of two components:

(1) A program (an array of bytes)

(2) A set of bindings.

Compilers could then create this object in the sure knowledge that whenever the program is used, the bindings (to its static objects) will be unaltered. Notice that the above object corresponds exactly to a `closure' as used in many programming language implementations. Consider a lineprinter spooler represented as a closure object:



Given that this new class of objects could be added to UNIX, the two main existing file types - plain file and directory - are just special cases and need no longer be implemented separately. A plain file is a closure with no bindings; a directory is a closure with no program.

This addition of a closure object now makes UNIX a suitable environment for programming using `modular' or `abstract data type' techniques. A closure object can be used to represent a module or data type with internal state held between uses e.g. a Simula class or a Modula Module[Birtwhistle et al. 1973, Wirth 1983].

(3) Pathname naming for UNIX files could be replaced by Combined naming (see Section 4.2). Recall that Combined naming is a synthesis of block structured and pathname naming in which each element of a pathname is in turn treated as a block structured name. The obvious advantage is that pathnames need no longer be `exact': they need not name every context used in resolving them. Also it is simple to provide `attribute inheritance' as in Smalltalk-80: if some name cannot be resolved locally then its definition is `inherited' from an enclosing object.

(4) the final suggestion is to remove the distinction between <u>processes</u>

and files. This involves replacing the fork-join style of execution by either data driven or demand driven computation. At the shell level, a pipeline

a | b | c

can already be viewed as being data driven: process `b' is driven by the arrival of data from process `a' and `c' is driven by `b' - or as demand driven: process `c' demands data from `b' which demands from `a'. This represents a move towards novel forms of computation where the distinction between program and process is unimportant.

Note

A simple priority scheduling rule added to UNIX could implement something closely resembling data (or demand) driven computation by giving higher priority toward the left (or right) hand end of the pipeline. Execution would be purely data (or demand) driven unless any pipe in the line became full.

More importantly this suggests a useful method of resource management for data driven or demand driven computers: by inserting something like a `pipe' between each producer and consumer and delaying the producer when the pipe becomes full, over-production would no longer swamp the system. In other words data or demand driven evaluation proceeds at full speed, occasionally (automatically) checked by pipes becoming full. Certainly this seems a preferable to adding resource management annotations to high level language programs as some have suggested[Burton 1984].

End of Note

7.2 Earlier Work on Recursive Systems

'Recursive Machines and Computing Technology'

The Russian work[Glushkov et al. 1974] presents 'recursive computers' as an alternative to the so-called Von Neumann model. The Von Neumann principles (see Section 6.2) are replaced by the following recursive principles:

P1 Recursive machines contain limitless numbers of levels of machine language.

P2 All program elements for which operands are available are to be executed (although elements are only inspected for execution under certain conditions).

P3 Memory structure is flexible.

P4 There is no limit to the number of machine elements.

P5 The machines have a flexible re-programmable structure.

The emphasis is on structured organization (in particular, recursively structured organization) in order to combat complexity.

A basic language for recursive machines is proposed from which higher level objects and languages can be built. The basic objects include 'lists' which are a form of <u>context</u>; these can be formed into a recursive structure thus making recursive naming possible. Similarly recursive computation is embodied in the system, although it is not clear whether it is data driven or demand driven.

DDM/1

The DDM/1 project[Davis 1978] grew out of work research at the Burroughs Corporation laboratories and was later developed at the University of

Utah. Interestingly, Burroughs is also the origin of the work by Wilner described below (with perhaps R.S. Barton being a common influence on the two projects).

The DDM/1 design adopts the recursive principles 1-5 (above) and adds a sixth:

P6 Modules of recursively structured machines should function in a fully distributed asynchronous manner.

Where 'fully distributed systems' are defined to have two characteristics:

C1 At no time can a module of a fully distributed system determine the total system state.

C2 A fully distributed system is incapable of enforcing simultaneity in its distributed modules (modules function wholly on the basis of their local time).

The aim of the additional principle (P6) is to clarify the control mechanism in the recursive machine (essentially by clarifying principle P2).

In DDM/1, machine language programs are Data Driven Nets (DDNs) which have essentially the same structure as the 'data dependency graph' introduced earlier (in Section 6.2). No mention is made of naming mechanisms, although DDNs are implemented as delimited strings and hence some form of context relative naming by selectors seems likely. A single DDM/1 element was constructed successfully although no performance figures or operational experience are given.

Wilner's Recursive Machine

The work of Wilner on recursive machines is documented in a series of reports (written at the Xerox Palo Alto Research Centre) available to the research community but not formally published. The only accessible description I can cite is the report of Wilner's presentation at a VLSI workshop in Newcastle[Wilner 1980].

An important feature of the work is how it demonstrates recursive structure as a way of exploiting VLSI technology. The importance of 'physical recursion' for VLSI is emphasized in addition to recursive program structure. 'Physical recursion' simply means that any one chip containing a single processing element may easily be replaced by another containing many such elements cooperating as one. This form of scale independence could be very important as the life-time of a design is long compared to the time between leaps in chip capacity. It also means that any one single chip can be replaced by a set of similar chips acting in concert to give better performance (although at greater hardware cost).

Wilner's design uses an object-oriented[Goldberg and Robson 1983] computation model with message passing in contrast to the data driven model of DDM/1. Messages are routed by structure selectors which are essentially pathnames through the program structure. Parallelism is available through the basic operators, for instance the APL-like 'insert'. These make it possible for many parallel sub-computations to be generated implicitly from a source program.

Recursive Control Flow

The Recursive Control Flow (RCF) architecture can in some ways be seen as a development of Wilner's work. The basic idea is to combine control flow computation with recursive program and machine organization. One major aim is to provide a de-centralized model of computation suitable for VLSI implementation using many similar chips rather than a set of specialized chips.

RCF programs are represented as nested strings with pathname-like selectors. In programs there is an implicit left to right flow of control between items at a given level of nesting. In addition explicit control flow operators are provided including GOTO and FORK. An aim of the design is to support as wide a range of programming styles as possible.

The primitive operators allow data driven and demand driven computation to be simulated. For example a special 'unknown' value can be placed in an instruction causing it to be delayed until the value becomes 'known'. This can be used to mimic data driven computation. There is a three-argument version of the multiply instruction which contains a destination address for the result; in addition a two-argument multiply is available which has update-in-place behaviour (in the absence of a result address).

* * * * *

The notion of recursive structure has been used in many systems, but usually in only a partial or ad hoc fashion. The papers described above are just those which make it an explicit issue. There are strong links between the papers, starting from the explicit description of recursive systems[Glushkov et al. 1974] which was then adapted for data

driven[Davis 1978] object-oriented[Wilner 1980] computation and finally extended to include to control driven computation by the Newcastle group[Treleaven and Hopkins 1982].

The papers surveyed reinforce the advantages of recursive structure as listed in Chapter 2:

(1) At every level the structure is the same: for example in Recursive Computers[Glushkov et al. 1974] there are limitless levels of machine language (principle P1 above).

(2) Combining recursive structures produces another recursive structure: this is the basis of 'physical recursion', where a combination of computing elements can be made to appear as a single system.

(3) Absence of a distinguished 'root': this is the basis of the 'fully distributed system' as defined by Davis[Davis 1978].

7.3 Object Management in Recursive Systems

One problem with recursive structure occurs when it has to be simulated on top of a flat structure. For example a significant part of the overhead in LISP interpreters is caused by providing recursively structured objects on top of a conventional flat store. Various techniques are available under the heading of 'garbage collection', to assist in such cases[Cohen 1981, Knuth 1972].

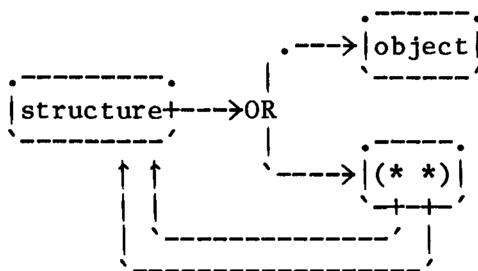
Garbage collection is the process of reclaiming objects no longer required. It is only appropriate to garbage collect <u>re-useable</u> objects, e.g. disc storage space, and not resources that can only be used once. Unless there is an unlimited supply of objects it is always necessary to perform garbage collection.

The main garbage collection schemes are (i) mark-scan and (ii) reference counts. Each has its advantages although in practice the great majority of garbage collectors use mark-scan. This is at least in part because reference counting cannot deal with cyclic structures.

Cyclic structures are those which contain closed loops of pointers from some object back to itself. For example the definition of recursive structure:

```
structure ::= object
            | (structure structure)
```

can itself be represented by a cyclic structure:



Note that it is not essential to use a cyclic representation but it is often the most convenient or efficient.

The aim here is to present a new algorithm for garbage collection in the presence of cycles. It arose from work on graph reduction machine design but is also relevant to extensions the to UNIX described above, or indeed any systems where garbage collection is needed. The new algorithm uses reference counts and is an extension of the usual reference counting algorithm.

A Brief Survey of Object Management Techniques

Mark-scan garbage collection need only be invoked when there are no objects free, otherwise it imposes no penalty. The first phase (mark) causes every object actually in use to be `marked' (usually by setting a

bit), starting from the 'root' and recursively traversing pointers. The 'scan' phase examines the mark on every object, un-marked objects are free and are collected together for re-use.

The advantages of mark-scan are that, given sufficient resources, no garbage collection overhead is incurred and that the space overhead is minimal: one bit per object. A disadvantage of mark-scan is that the structure must not be changed during garbage collection, so normal work must be suspended. A second disadvantage is that <u>all</u> objects must be scanned (no matter how many are free), so the 'cost' of garbage collection is proportional to the total number of objects. In a distributed system this can be serious because work on <u>all</u> processors must halt for a global search to take place when <u>any</u> <u>one</u> processor runs out of objects. There are versions of mark-scan which operate in parallel with normal processing but the garbage collection is still <u>global</u> in the sense that the entire system is always searched[Dijkstra et al. 1978, Kung and Song 1977].

Reference counting consists of associating with each object a count of the number of references (pointers) to it; when the count reaches zero the object is free and can be re-used. The reference count has to be updated each time the structure is changed (e.g. when adding or deleting a pointer).

The advantage of reference counting is that the overhead is spread between all structure manipulations: the work done is just a constant factor of overhead. Also, reference counting is localized, only accessing structurally adjacent objects. The disadvantage of reference counting is that it cannot guarantee to deal correctly with cyclic (i.e. self-referencing) structures. The new algorithm presented below attempts to solve this problem whilst retaining most of the advantages of reference counting over mark-scan garbage collection.

The System Model

For the purposes of exposition an abstract system model (a directed graph) will be used which hides certain implementation-dependent details. The aim is to introduce the new algorithm and at the same time demonstrate its correctness.

The notation used is simple but rather limited by the set of characters available for printing:

A ::= B	Axiomatic Definition: A is defined to be B
A x B	Cartesian Product of A and B
A ++ B	Union of set A and set B
A - B	Set Difference (asymmetric)
a ∈ A	Set Membership: true when a is a member of set A
{s | P}	Set Definition: the set of s such that P is true
|| S ||	Cardinality: the number of elements in set S
A & B	Boolean: A and B
A̲ A:B:C	Predicate: for all A for which B holds, C holds

Let the system consist of a set of objects (Ob) and a set of pointers (P) between them:

Ob ::= some finite non-empty set

P ::= some sub-set of (Ob x Ob)

When an ordered pair (r,s) is in P it is said that \`r points to s' (or alternatively \`s is pointed to by r'). For a given object the set of objects which it points to are called its \`sons':

$$sons(r) ::= \{s \mid (r,s) \in P\}$$

In practice the system may be implemented by having each object contain the addresses of its sons.

Those objects which can be reached by traversing a series of pointers from a given object will be called the 'reach' of the object:

$$reach(r) ::= sons(r) +\!\!+ \{reach(s) \mid s \in sons(r)\}$$

The reach of r is the union of the sons of r and the sets of objects in the reach of each of the sons of r. When some object s is in reach(r) it is said that 's is reachable from r'.

The purpose of object management is to keep track of the objects which are in use and those which are free. One fixed object, the Root, is selected as always being in use as the base of the system. Any other objects reachable from Root are defined to also be in use:

$$Inuse ::= \{Root\} +\!\!+ reach(Root)$$

The remaining objects are free for re-use:

$$Free ::= Ob - Inuse$$

Reference Counting Object Management

Reference counting object management operates by associating a number with each object to indicate the number of pointers (i.e. references) to it. This number will be called the reference count:

$$refc(s) ::= \| \{(r,s) \mid r \in Ob;\ (r,s) \in P\} \|$$

Note that the reference count of an object is <u>defined</u> by the above axiom. In what follows it will be sufficient to describe the creation

and deletion of pointers, without explicitly giving the corresponding changes to reference counts.

In practice the reference count may be stored as a part of the object.

Reference counts are correct if they can reliably be used to determine which objects are in use and which are free. For a given set S of objects, correctness is expressed by the predicate P1:

$$P1(S) ::= \underline{A}\ s : s \in S : refc(s)=0 \Leftrightarrow s \in Free$$

Thus the entire set of reference counts is correct when P1(Ob) is true. It is then guaranteed that objects with reference count zero are free.

The Standard Reference Count Algorithm

The standard reference count algorithm will now be described as it forms a part of the new improved algorithm.

The aim is to allow pointers to be manipulated (created,copied or destroyed) whilst maintaining the truth of P1(Ob). For the purposes of exposition, three pointer-changing operations will be described although other sets of operations can be supported. The three operations are to create a new pointer (NEW), to copy an existing pointer (COPY) and to destroy a pointer (DELETE).

NEW(r,s) `Create a new pointer from r (r ∈ Inuse) to some free object, thus bringing it into use'

```
NEW(r) ::=
    search Ob for s such that refc(s)=0;
    P := P ++ {(r,s)}
```

COPY(r,(s,t)) `Copy pointer (s,t) so that r points to t; (r,s,t ∈ Inuse)'

```
COPY(r,(s,t))) ::=
    P := P ++ {(r,t)}
```

DELETE(r,s) `Destroy pointer (r,s); (r,s ∈ Inuse)'

```
DELETE(r,s) ::=
    P := P - {(r,s)}
    if refc(s)=0 & s≠Root then
        for t in sons(s) do DELETE(s,t) od
    fi
```

The above algorithm is of course unsatisfactory as it does not preserve Pl(Ob) when there are cycles in the graph. Consider a graph of two objects and two pointers:

```
Ob := {Root,a}
P := {(Root,a),(a,a)}
```

Pl(Ob) holds, Inuse={Root,a}, Free={}.

Clearly there is a cycle because of the pointer (a,a). When the pointer (Root,a) is deleted the state of the system is defined by `DELETE' above:

```
Ob := {Root,a}
P := {(a,a)}
```

Hence Inuse={Root}, Free={a}. However Pl(Ob) is false because (a ∈ Free) and (refc(a)=1).

Strong and Weak Pointers

Now the new algorithm can be described. It as based on the fact that the standard reference count algorithm given above will work when there are no cycles. (The proof is left as an exercise for the reader).

In the new algorithm, the pointers (P) are partitioned into two distinct subsets which will be called Strong and Weak:

Strong is a subset of P
Weak is a subset of P

Strong ++ Weak ::= P
Strong Intersect Weak ::= {}

The aim is that the strong pointers will be acyclic and span every object in use; the weak pointers are the remaining, non-strong pointers. Provided the strong pointers are acyclic, they can be managed correctly by the standard algorithm. The strong and weak pointers are set up in a way which allow easy detection of the situations where standard reference counting would fail.

The strong-sons and strong-reach of an object can be defined by replacing P by Strong in the definitions of sons and reach above. The strong-sons of an object are those pointed to by a strong pointer from r. Similarly the strong-reach of an object r consists of those objects which can be reached from r by traversing only strong pointers:

$$Ssons(r) ::= \{s \mid (r,s) \in Strong\}$$

$$Sreach(r) ::= Ssons(r) ++ \{Sreach(s) \mid s \in Ssons(r)\}$$

The strong pointers and weak pointers to an object (r) contribute to the corresponding strong and weak reference counts (Srefc(r), Wrefc(r)):

$$Srefc(s) ::= || \{(r,s) \mid r \in Ob; (r,s) \in Strong\} ||$$

$$Wrefc(s) ::= || \{(r,s) \mid r \in Ob; (r,s) \in Weak\} ||$$

In practice these reference counts may be stored as part of each object.

As a consequence of the above definition it is always true that the reference count of an object is the sum of its strong and weak reference counts:

```
refc(r) ::= Srefc(r) + Wrefc(r)
```

The algorithm will rely on two properties of strong pointers:

(i) Strong pointers are acyclic, no object can be reached from itself by traversing only strong pointers. For a given set S of objects:

```
strong_acyclic(S) ::=
        A s : s ∈ S : not(s ∈ Sreach(s))
```

Use of strong pointers alone will never cause the standard algorithm to fail.

(ii) Strong pointers span the objects in use, each object in use can be reached from Root by traversing strong pointers only. For a given set S of objects:

```
strong_span(S) ::= A s : s ∈ S :
                     s ∈ reach(Root)
                       <=> s ∈ Sreach(Root)
```

All objects reachable from Root have some strong pointers to them.

This can be summarised by the predicate P2, (noting that for correctness P1 must also hold). For a given set S of objects:

```
P2(S) ::=  P1(S) & strong_acyclic(S) & strong_span(S)
```

The purpose of the new algorithm is to ensure

```
P2(Ob)
```

holds after any pointer manipulation, given that it held before.

The new implementation of the three operations NEW, COPY, DELETE can now be described.

NEW(r) `Create a new pointer from r to some free object thus bringing it into use; (r ∈ Inuse)'

```
NEW(r) ::=
    choose s from Ob such that refc(s)=0;
    Strong := Strong ++ {(r,s)}
```

P2(Ob) is preserved by NEW(r). Given P2(Ob) holds before the new pointer is created, s is initially free and points to no other objects, so the new pointer (r,s) cannot create a cycle; because (r,s) is strong and r is in use, s becomes a member of both reach(Root) and Sreach(Root).

COPY(r,(s,t)) `Copy pointer (s,t) so that r points to t; (r,s,t ∈ Inuse)'

```
COPY(r,(s,t)) ::=
    Weak := Weak ++ {(r,t)}
```

P2(Ob) is also preserved by COPY. There is no change to Inuse and (r,t) cannot create a strong cycle as it is weak.

DELETE(r,s) `Destroy pointer (r,s)'

When deleting a pointer there are four cases to be considered depending whether the pointer (r,s) is strong or weak and what other pointers there are (to s). The first three cases are straightforward and action taken is exactly as in the standard algorithm. The fourth case corresponds to situations where the standard algorithm will fail; in these cases a local search is made of objects reachable from s to decide whether s has become free and if appropriate, re-establish a correct partitioning of pointers into strong and weak. If s has become free then the pointers from it to other objects are deleted also.

(i) DELETE(r,s) when (r,s) is weak

```
DELETE(r,s) ::=
    Weak := Weak - {(r,s)}
```

No change to Inuse; all objects still have a strong pointer to them.

(ii) DELETE(r,s) when (r,s) ∈ Strong and Srefc(s)>1

```
DELETE(r,s) ::=
    Strong := Strong - {(r,s)}
```

No change to Inuse; all objects still have a strong pointer to them.

(iii) DELETE(r,s) when (r,s) ∈ Strong & Srefc(s)=1 & Wrefc(s)=0

```
DELETE(r,s) ::=
    Strong := Strong - {(r,s)}
    if s≠Root then
        for t in sons(s) do DELETE(s,t) od
    fi
```

If there are no pointers to a object it cannot be reachable from Root; so if it is not Root, it must be free. To re-establish P2(Ob), recursively delete sons to ensure there are no pointers from free objects to objects in use and that all free objects have a reference count of zero.

(iv) DELETE(r,s) when (r,s) ∈ Strong & Srefc(s)=1 & Wrefc(s)>0

In this case it cannot be decided immediately whether s is free or not. This is precisely the situation which standard reference counting will fail. The new algorithm performs what is essentially a localized mark-scan garbage collection rooted at s.

The following two axioms are the key to the algorithm:

(I) Deleting pointer (r,s) can affect the reference counts of <u>at</u> <u>most</u> s and reach(s). Any object not reachable from s cannot be affected.

(II) The objects which become free when any pointer (r,s) is deleted are those reachable from s but not in use:

Newfree ::= reach(s) - Inuse

::= reach(s) - ({Root} ++ reach(Root))

The method is first to mark the objects which might be affected by the deletion (Axiom I) by colouring them black. (Remembering that by Axiom II, Root should always remain white), then to scan the black objects searching for one pointed at by a non-black object. Such an object must be reachable from Root so it can be coloured white, as can all objects reachable from it. Once this has been done, all the original black objects which were reachable from Root have become white. The remaining black objects are (by Axiom II) exactly the members of Newfree, and can be freed in the usual way.

Black is a subset of P
White is a subset of P

Black ++ White ::= P
Black Intersect White ::= {}

The aim is that whites are correct and blacks are those which need to be considered.

Black := {s} ++ reach(s)

(assume all other objects are white)

Initially, P2(White) holds because whites are unaffected by the pointer deletion; some weaker condition P3 holds for the black objects. The objects coloured black are either newly freed or are reachable from a white object. For a given set S of objects:

P3(S) ::= (S = Newfree ++ reachW(S))

Where reachW(S) ::= S Intersect {reach(w) | w ∈ White}

After pointer (r,s) is deleted the predicate Q will be true:

Q ::= P2(White) & P3(Black)

The truth of P3(Black) can be shown to be true by recognizing that the Root is white so the objects reachable from white objects are exactly the objects in use.

The set WB of black objects pointed to by a white object is selected and strengths are adjusted so that at least one pointer from black to white is strong. This can never create a cycle so it is now safe to make these objects white and (P2(White) & P3(Black)) will still hold. Now the objects reachable from WB are scanned changing them to white and at the same time ensuring Q by sometimes altering the 'strength' of pointers as they are traversed.

Finally P2 is established over all the original blacks that were reachable from whites and P3 still holds for the remaining blacks. As reachW(Black) is now empty the remaining black objects are exactly the members of Newfree, the set of objects which have become free. Each pointer in a black object is now deleted leaving the black objects with a reference count of zero. Because they are free, and have zero reference counts then P2(Black) is now true. Given

(P2(White) & P2(Black))

and that

(Black++White::=Ob)

it follows that P2(Ob) is now true, as required.

All this can be achieved by the following program for case (iv) of DELETE when (r,s) is strong, Srefc(s)=1 and Wrefc(s)>0:

```
DELETE(r,s) ::=
    Strong := Strong - {(r,s)};
    if s≠Root then
        colour(s):=Black;
        for t in sons(s) do mark(s,t) od;
        for t in sons(s) do scan(s,t) od;
        if colour(s)=black then
            for t in sons(s) do kill(s,t) od
        fi
    fi
```

Where the procedures mark, scan and kill perform the main work of the algorithm:

```
mark(r,s) ::=
    if s≠root then
        if colour(s)=white then
            colour(s):=black;
            Brefc(s):=1;
            for t in sons(s) do mark(s,t) od
        else
            Brefc(s):=Brefc(s)+1
        fi
    fi
```

It is initially assumed that all objects are white (e.g. they are given that colour by NEW). The purpose of Brefc(s) is to count the number of pointers to s from black objects. When marking is complete, any objects for which

$$refc(r) > Brefc(r)$$

must be pointed to by a white object.

```
scan(r,s) ::=
    if colour(s)=black then
        if Brefc(s)<refc(s) then
            `make sure there is at least one strong
                pointer from a white object to s'
            colour(s):=white;
            for t in sons(s) do partition(s,t) od
        else
            for t in sons(s) do scan(s,t) od
        fi
    fi
```

The black objects with pointers from white objects (reachW(Black)) are

used as `sub-roots' from which to establish more white objects, at the same time ensuring P2(White):

```
partition(r,s) ::=
    if colour(s)=white then
        Weak:=Weak+{(r,s)}
    else
        colour(s):=white;
        Strong:=Strong+{(r,s)};
        for t in sons(s) do partition(s,t) od
    fi
```

The partitioning procedure ensures that the pointers from white objects have the correct strengths. The procedure can be derived directly from the definition P2. Finally, any remaining black objects must be visited to ensure their reference counts become zero as required:

```
kill(r,s) ::=
    if colour(s)=black then
        P:=P-{(r,s)};
        for t in sons(s) do kill(s,t)
    fi
```

It is guaranteed that the newly free objects are spanned by the object on which deletions are performed.

* * * * *

Another version of the algorithm (without proof outline) is presented as an appendix. That version is designed for use in combinator graph reduction machines, or indeed any application where it is always known when cycles are being created. Combinator machines have the property that arbitrary pointer manipulations are not allowed; only certain `combinator' instructions manipulate the structure with a consequent simplification of the object management scheme. In particular it is known that cycles are only created by the Y instruction.

The key advantage of the new reference counting algorithm is its strong locality. Even in a very large structure, possibly spanning many

computers, it is only necessary (at most) to examine the objects reachable from some object to determine whether it is free. Such locality might also be especially useful in Virtual Memory systems, where alternative garbage collection methods require every page in the Virtual Memory to be examined, no matter how few objects are actually freed.

The new algorithm can be compared to other algorithms by Bobrow and by Hughes[Bobrow 1980, Hughes 1983]. In the former scheme, objects are divided into zones, with intra-zonal pointers being treated differently from inter-zonal pointers. Intra-zonal pointers may form cycles and are not reference counted; inter-zonal pointers may not form cycles and are reference counted. Objects do not become free until the reference count of their whole zone becomes zero. The user of the algorithm is relied upon to create the zones and ensure that inter-zonal pointers are acyclic. When such pointers are manipulated, the user must be able to find the correct per-zone reference count in order to update it.

Hughes' scheme can be regarded as an automated version of the one of Bobrow just outlined. Instead of making the user define the zones they are dynamically computed; instead of making the user find the per-zone reference count, each object contains a pointer to it. Zones are defined by the maximal cycles in the structure and are being detected using an algorithm due to Tarjan[Tarjan 1972]. As pointers are manipulated it is occasionally necessary to re-compute the zones. Although an improvement on the Bobrow algorithm, the use of Tarjan's cycle detection algorithm is a problem, it appears to require that every resource be visited (or at least, every pointer traversed).

The new algorithm that I have introduced above can be viewed as a further development of these two. Not only are cycles 'recognized' automatically, but also there is only a localized search to determine whether an object is free.

There are of course many other schemes for object management but nearly all either rely on some form of global search (or global synchronization), or are not sufficiently general.

An Essential Implementation Trick

In the above description the alert reader will have noticed the operation

'make sure there is at least one strong
pointer from a white object to s'

The details of this operation will not be given. (There is a possible need to count black strong pointers and white strong pointers separately.) However a useful implementation trick will be described. It enables the strength of pointers to an object to be changed without knowing which objects contain those pointers, for example, the operation:

'make all the weak pointers to s into strong pointers'

(which is used in the version of the algorithm in the Appendix). This could have a naive implementation:

'visit every object, if it points to S change strong pointers to weak'

But the whole point of reference counting has been to avoid this kind of global search.

Instead there is a implementation trick which exactly suits our purpose. Each <u>pointer</u> and each <u>object</u> can have a <u>bit</u> associated with it. When a pointer and a pointed-to object have the same bit-value the pointer is <u>strong</u>; when a pointer and a pointed-to object have different bit-values the pointer is <u>weak</u>.

All operations utilizing the `strength' of pointers can now be defined in terms of single-bit changes:

`make all the weak pointers to s into strong pointers'

is implemented as

s.bit := NOT(s.bit)

(where `s.bit' is the strength bit associated with s) and

`make pointer p into a weak pointer'

is similarly implemented as

p.bit := NOT(p.bit)

(with corresponding changes to the appropriate reference counts).

The net overhead in addition to the standard reference count algorithm is one bit per pointer and one bit per object; in addition some extra reference counts are needed for weak pointers when they occur. As weak pointers only occur when cycles are present, this is in total a very low space overhead. (There is also the cost of the temporary black reference counts in the final case of DELETE.)

Note that the strength/weakness of a pointer can only be determined by examining both the pointer and pointed-at object. As strength or weakness need only be known by the garbage collector when it is traversing the structure, no problems are introduced by the trick.

Chapter 8

Conclusion

> This leads us to the question, which must have been rising insistently in the reader's mind: What is the use of all this elaboration? At this point our friend the practical man, must surely step in and insist in sweeping away all these silly cobwebs of the brain. The answer is that what the mathematician is seeking is Generality.
>
> [Whitehead 1911]

8.1 Summary

In this thesis I have attempted to show that recursive structure is a design principle of general applicability in computer systems. In addition I have illustrated the principles of recursively structured naming (and its use in UNIX United) and recursively structured computation. Good structure is essential if computer systems are not to become complex and unwieldy.

A model is an abstraction of a system. By using a model it is possible to concentrate on the structure (form) of a system independent of the content of its parts. A model concentrates on some aspect(s) of interest at some level of detail in the system. Hierarchical structures are good for modelling systems because they simplify by encapsulation. A recursive structure is a uniform hierarchy: anywhere an atomic element can appear a complete sub-structure may also appear. Combining recursive structures produces another recursive structure of the same type;

recursive structures have no distinguished `root' element and so may be arbitrarily nested to any depth.

In order to separate the static and dynamic parts, a computer system can be modelled as a program plus a computer which executes the program. Multi-level systems can be built by combining computers or by stacking programs `on top' of each other to form multi-level interpreters. A recursive interpreter extension is a form of multi-level interpreter where the new level retains the appearance of the old whilst `adding value' to the old interpreter. The added value may take the form of the ability to access a wider range of objects (i.e a larger name space) or it may be less tangible, in the form of increased reliability or availability.

Naming is the use of names to stand for objects. Naming is a particularly important aspect of computer system design. The naming mechanisms define the range of objects that can be manipulated. There is some form of naming at each level in a multi-level system. Typically a name at one level is an address (or route) at another (lower) level. The important idea emphasized was that no matter which level is being considered the basic naming issues are the same. The particular naming mechanisms chosen at any level reflect the constraints of that level but there is an overall `theory' of naming which is applicable at every level. Some common naming mechanisms were surveyed, each being summarized by a simple rule for `resolving' names. Recursive naming was shown to have particular advantages for building systems because it allows them to be combined (or decomposed) to arbitrary depths. In addition a new naming scheme combining the advantages of two common naming methods was given.

The UNIX United/Newcastle Connection scheme demonstrates the use of recursive naming in the design of a distributed computer system based on

the UNIX Operating System. File naming in UNIX is recursive, so the UNIX United design simply extends this naming to encompass files stored on many systems. This design was implemented (by the Newcastle Connection) and provided a useful distributed system which has been adopted widely elsewhere. Unfortunately other UNIX system objects do not have recursive naming so it is not easy to incorporate them into a distributed system. Instead names for these objects are `mapped' between systems to provide each system with the illusion of being the only one.

Recursive computation represents a divide-and-conquer approach to program execution. The commonest model of computation used today is the so-called Von Neumann model, a form of control driven computation. It is characterized by a single thread of control, a linear array of storage cells and a low-level machine language. In data driven computation instructions are executed when the data they require becomes available. In demand driven computation instructions are executed when the data they produce is needed by another instruction. A survey of control driven, data driven and demand driven computation was presented, demonstrating the advantages of recursive computation in terms of locality of reference and execution.

Future computer systems may be a development of the UNIX system. Some suggestions were made as to beneficial changes which could be made to UNIX. However it seems likely that the next widely successful class of systems will be considerably different from the system of today. It was suggested that some of the ideas for recursive naming might be incorporated into such future systems and that they might also involve a recursive model of computation. In conjunction with these suggestions, some of the other work on recursive computer systems was briefly examined.

Finally a reference counting object management scheme was described

which is able to deal correctly with cyclic structures without involving global overhead. One development of the scheme provides an incremental object manager for a `graph reduction' machine (which supports functional programming) and this is described in an Appendix.

8.2 Future Research

The design of recursively structured systems offers considerable scope for further investigation. Some particular areas of the thesis which could benefit from further research will now be considered.

Developing the work described here, it should be possible to build a UNIX-based recursively structured system. This could incorporate recursive naming of `processes' and `users' perhaps also using `Combined Naming' to ease the inconvenience of pathnames. The resulting system would be close to UNIX but a significant improvement. I feel that this would be an interesting exercise but of little practical significance. The advantages of the new system are unlikely to out-weigh the disadvantage of moving away from a standard system.

A new system adopting some of the more radical changes suggested above (e.g. closure objects, some element of data driven or demand driven computation) would be more ambitious and interesting. It could involve a synthesis of ideas from programming language and operating system design. It should be clear that such a system should be designed to be a component of a recursively structured system. If this is done then there are fewer problems when systems are to be combined or partitioned.

Formal models of computer system structure are an area suitable for further research. The one aspect largely ignored in the treatment of system structure above was the separation of structure into two hierarchies: the `details' hierarchy which describes which parts are

components of others and the 'uses' relation which describes which parts are used by name in others. This could lead to a simplification and generalization of programming language 'module' constructs.

Naming was presented from an essentially operational point of view. It would be an improvement to give a more mathematical treatment. Naming could be viewed as the relation between names and objects. In a relational model of naming, the definitions given above for name resolvers could be used to define a relation over names and objects. A context could be modelled as a binary relation (i.e. a subset of all the pairs of names and objects); a context structure could then be some higher order relation. Such a model would be a nice generalization and might result in some simplification and the discovery of further naming schemes.

Further development is being undertaken (by others) on the Newcastle Connection software. The original experimental version is being adapted for use over wide area networks and multiple networks. This involves a substantial implementation effort but retains the UNIX United system design. Another aspect being investigated is the design of a standard network interface for the Newcastle Connection to make it easier to adapt the software to different networks.

It is worth noting that the Newcastle Connection cannot be fully transparent. The reason is that, despite what was said above, the UNIX kernel interface is not completely specified. For example, one of the system calls (ioctl(2)) provides a device-dependent extension to the kernel in order to use the built-in features of devices attached to a system. Each kind of device can make different uses the parameters of 'ioctl(2)', to the extent that value parameters of one device's interface can be result parameters when another device is accessed. In addition the set of devices available is system-dependent. In future

systems it would be useful if some kind of standard device interface were provided, rather in the way that a standard interface to terminals is provided in the more recent versions of UNIX.

The area of the thesis with most immediate scope for further work is the resource management algorithm described in Chapter 7. The algorithm given can be improved in cost, especially when more is known about the way in which it will be used. This is demonstrated by the version of it given in the Appendix which manages structures created using combinators. Further work is needed to show the correctness of this latter algorithm. Also it would be useful for distributed systems to determine the conditions for non-interference of concurrent invocations of the algorithms.

* * * * *

To conclude, it has been shown that structure can play an important part in the design of computer systems. Recursive structure has been defined and shown to be a good way of arranging the structure of naming and of computation. The UNIX United/Newcastle Connection system demonstrates how recursive naming can simplify the design and construction of a practical distributed system. A solution has been given to the problem of managing resources in a structured system ('garbage collection'); this new reference counting algorithm has the same (constant) cost as the standard algorithm unless there are cycles present when there is an extra overhead related to the size of the cycle.

This work contributes to the overall aim of managing the complexity of computer systems and towards the design of better systems in the future, based on the principles of recursive structure.

Appendix

Object Management in Combinator Graph Reduction

Here, another version of the reference count algorithm is given. This version is more efficient but can only be used when information is available about the creation of cycles. That is, when a pointer is copied, it must be specified whether it needs to be weak.

In this case the set of weak pointers can be made minimal by only making pointers weak when absolutely necessary. The weak pointers are minimal in the sense that making any one weak pointer strong would create a cycle of strong pointers.

For example, in a combinator graph reduction machine[Clarke et al. 1978, Turner 1979] it is known that only one instruction, the Y combinator, will create a cycle. Execution of Y can be implemented by creating a weak pointer, all other pointers can be strong (except subsequent copies of weak pointers).

The minimality of weak pointers means that the mark and scan phases of the algorithm can be combined into a single traverse, visiting each object reachable from s at most once in determining whether s is free. If s has not become free, the same traverse will have ensured the correctness of pointers from s and reach(s).

The new 'minimality-dependent' algorithm is the same as before except for two alterations:

(1) COPY is given an extra, boolean, parameter to indicate whether to create a weak or a strong pointer (the correctness of the parameter value being assumed).

(2) The crucial case of DELETE, when the last strong pointer to an objects s with (Wrefc(s)>0), is handled as follows:

DELETE(r,s) `where (r,s) ∈ Strong, Srefc(s)=1, Wrefc(s)>0'

```
DELETE(r,s) ::=
    Strong:=Strong-{(r,s)};
    `make all the pointers to s strong';
    for t in sons(s) do suicide(s,(s,t)) od;
    if Srefc(s)=0 then
        for t in sons(s) do kill(s,t) od
    fi
```

where `make all pointers to s strong' can be accomplished by altering a bit at s as described earlier and `kill' is the same as used in the previous version of DELETE. `suicide' performs a recursive traversal of objects reachable from s:

```
suicide(me,(r,s)) ::=
    if (r,s) ∈ Strong then
        if (Srefc(s)>1) or (s=me) then
            Weak:=Weak+{(r,s)}
        else
            for t in sons(s) do suicide(me,(s,t)) od
        fi
    fi
```

The operation `make all the pointers to s strong', creates N cycles of strong pointers where N=refc(s) (because at that point every pointer to s is weak and weaks are minimal). The recursive traversal attempts to `weaken' just sufficient strong pointers to `undo' these incorrect cycles without weakening the last strong pointer to any object (except s which is carried along as parameter `me'). On termination, if all pointers to s have been weakened then s must be free; otherwise s is not free and correctness (i.e. P2(Ob)) has been restored.

I have made use of the above algorithm successfully in an emulation of Turner's combinator graph reduction machine.

References

[ALGOL 60 1963]
P. Naur (ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, pp.1-17 (January 1963).

[Alexander 1964]
C. Alexander, Notes on the Synthesis of Form, Harvard University Press, Cambridge, Massachusetts (1964).

[Anderson and Lee 1981]
T. Anderson and P.A. Lee, Fault Tolerance: Principles and Practice, Prentice-Hall (1981).

[Apperley and Spence 1983]
M.D. Apperley and R. Spence, "Hierarchical Dialogue Structures in Interactive Computer Systems," Software Practice and Experience Vol. 13(9), pp.777-790 (September 1983).

[Backus 1972]
J. Backus, "Reduction Languages and Variable free programming.," Report RJ 1010, IBM T.J. Watson Research Center, Yorktown Heights, New York (April 1972).

[Banahan and Rutter 1982]
M.F. Banahan and A. Rutter, UNIX - The Book, Sigma Technical Press, Wilmslow, Cheshire (1982).

[Barron 1968]
D.W. Barron, Recursive Techniques in Programming, MacDonald / American Elsevier, London / New York (1968). (Computer Monographs: 3)

[Barron 1977]
D.W. Barron, An Introduction to the Study of Programming Languages, Cambridge University Press, Cambridge, England (1977). (Cambridge Computer Science Texts 7).

[Bell and Newell 1971]
C.G. Bell and A. Newell (eds.), Computer Structures: Readings and Examples, McGraw-Hill (1971).

[Birtwhistle et al. 1973]
G.M. Birtwhistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard, Simula begin, Academic Press (1973).

[Bobrow 1980]
D.G. Bobrow, "Managing Reentrant Structures Using Reference Counts," ACM Trans. on Programming Languages and Systems Vol. 2(3), pp.269-273 (July 1980).

[Bochmann 1983]
G. von Bochmann, Concepts for Distributed System Design, Springer-Verlag (1983).

[Brady 1977]
J.M. Brady, The Theory of Computer Science: A Programming Approach, Chapman and Hall (1977).

[Brooks 1975]
F.P. Brooks, Jnr., The Mythical Man-Month, Addison-Wesley (1975).

[Brownbridge et al. 1982]
D.R. Brownbridge, L.F. Marshall, and B. Randell, "The Newcastle Connection - or UNIXes of the World Unite," Software Practice and Experience Vol. 12(12), pp.1147-1162 (December 1982).

[Burge 1975]
W.H. Burge, Recursive Programming Techniques, Addison-Wesley (1975).

[Burton 1984]
F.W. Burton, "Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs," ACM Trans. on Programming Languages and Systems Vol. 6(2), pp.159-174 (April 1984).

[Buxton and Randell 1969]
J.N. Buxton and B. Randell (eds.), Software Engineering, Proc. NATO Conference, Rome, 1969.

[Chesson 1975]
G.L. Chesson, "The Network UNIX System," Proc. 5th Symp. Operating Systems Principles., pp.60-66 (1975). (In: ACM Operating Systems Review 9(5)).

[Clarke et al. 1978]
T.J. Clarke, P.J.S. Gladstone, C.D. Maclean, and A.C. Norman, "SKIM - The S,K,I Reduction Machine," Proc. LISP 1980 Conf., Stanford, California, pp.128-135.

[Cohen 1981]
J. Cohen, "Garbage Collection of Linked Data Structures," ACM Computing Surveys Vol. 13(3), pp.341-367 (September 1981).

[Currie et al. 1981]
I.F. Currie, P.W. Edwards, and J.M. Foster, "Flex Firmware," Report 81009, Royal Signals and Radar Establishment, Malvern, England (Sept. 1981).

[Dahl et al. 1972]
O-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press (1972).

[Date 1975]
C.J. Date, Introduction to Database Systems, Addison-Wesley, Reading, Massachusetts (1975).

[Davis 1978]
A.L. Davis, "The Architecture and System Method of DDM/1: A Recursively Structured Data Driven Machine," Proc. 5th Int. Symp. Computer Architecture, pp.210-215 (April 1978). (ACM SIGARCH Newsletter 6(7) 1978)

[Denning 1968]
P.J. Denning, "The Working Set Model for Program Behaviour," Comm. ACM Vol. 11(5), pp.323-333 (May 1968).

[Dennis 1979]
J.B. Dennis, "The Varieties of Data Flow Computers," Proc. 1st Int. Conf. Distributed Computer Systems, Huntsville, Alabama, pp.430-439, IEEE (1979).

[Dijkstra 1968b]
E.W. Dijkstra, "The Structure of the 'THE' Multiprogramming System," Comm. ACM Vol. **11**(5), pp.683-696 (May 1968).

[Dijkstra 1982]
E.W. Dijkstra, "EWD 477: On the Role of Scientific Thought," pp. 60-66 in Selected Writings on Computing: A Personal Perspective, Springer-Verlag (1982). (Texts and Monographs in Computer Science.)

[Dijkstra et al. 1978]
E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On The Fly Garbage Collection," Comm. ACM Vol. **21**(11), pp.966-975 (November 1978).

[Elson 1973]
M. Elson, Concepts of Programming Languages, SRA - Science Research Associates (1973).

[Fraser 1971]
A.G. Fraser, "On the Meaning of Names in Programming Systems," Comm. ACM Vol. **14**(6), pp.409-416 (June 1971).

[Glushkov et al. 1974]
V.M. Glushkov, M.B. Ignatyev, V.A. Myasnikov, and V.A. Torgashev, "Recursive Machines and Computing Technology," Information Processing 74, pp.65-70, North-Holland (1974).

[Goldberg and Robson 1983]
A. Goldberg and D. Robson, Smalltalk-80 The Language and its Implementation, Addison-Wesley (1983).

[Gries 1981]
D. Gries, The Science of Programming, Springer-Verlag (1981). (Texts and Monographs in Computer Science.)

[Henderson 1980]
P. Henderson, Functional Programming, Application and Implementation, Prentice-Hall (1980). (Prentice-Hall International Series in Computer Science).

[Henderson and Gimson 1981]
P. Henderson and R.B. Gimson, "Modularisation of Large Programs," Software Practice and Experience Vol. **11**, pp.497-520 (1981).

[Henderson and Morris 1976]
P. Henderson and J.H. Morris, "A Lazy Evaluator," Proc. 3rd ACM Symp. Principles of Programming Languages, pp.95-103 (January 1976).

[Hoare 1980]
C.A.R. Hoare, "Programming in an Engineering Profession," Technical Monograph, Programming Research Group, Oxford University Computing Laboratory, Oxford, England (1980).

[Hughes 1983]
R.J.M. Hughes, "Reference Counting with Circular Structures in Virtual Memory Applicative Systems," Oxford University Programming Research Group (1983).

[Hwang et al. 1981]
K. Hwang, W.J. Croft, G.H. Goble, B.W. Wah, F.A. Briggs, W.R. Simmons, and C.L. Coates, "A UNIX Based Local Computer Network with Load Balancing," _Computer_ Vol. 15(4), pp.55-66 (1982).

[Iliffe 1972]
J.K. Iliffe, _Basic Machine Principles_, American Elsevier, New York (1972). (2nd edition).

[Iverson 1980]
K.E. Iverson, "Notation as a Tool for Thought," _Comm. ACM_ Vol. 23(8), pp.444-465 (August 1980).

[JIPDEC 1982]
Anon., _Outline of Research and Development Plans for Fifth Generation Computer Systems_, Institute of New Generation Computer Technology (JIPDEC), Tokyo, Japan (1982).

[Jensen and Wirth 1978]
K. Jensen and N. Wirth, _PASCAL User Manual and Report_, Springer-Verlag (1978). (Second Corrected Reprint of the Second Edition).

[Jones 1981]
S.B. Jones, "The Performance Evaluation of Interpreter Based Computer Systems," Ph.D. Thesis, University of Newcastle upon Tyne (June 1981).

[Kernighan and McIlroy 1978]
B.W. Kernighan and M.D. McIlroy, _UNIX Programmer's Manual_, Bell Laboratories (1978). (Seventh Edition).

[Kernighan and Pike 1984]
B.W. Kernighan and R. Pike, _The UNIX Programming Environment_, Prentice-Hall, Englewood Cliffs, New Jersey (1984).

[Kilburn 1962]
T. Kilburn, "One Level Storage System," _IRE Trans. on Electronic Computers_ Vol. **EC-11**(2), pp.223-235 (April 1962).

[Knuth 1972]
D.E. Knuth, _The Art of Computer Programming, Vol. 1: Fundamental Algorithms_, Addison-Wesley, Reading, Massachusetts (1972).

[Kung and Song 1977]
H.T. Kung and S.W. Song, "An Efficient Parallel Garbage Collection System," Dept. of Computer Science Report, Carnegie-Mellon University (August 1977).

[LaPrie 1983]
J.G. LaPrie, "On Impairings, Means and Measures of Computing Systems Dependability: Basic Concepts and Taxonomy," Research Report 83.050 CNS/LAAS, Centre National de la Recherche Scientifique / Laboratoire d'Infomatique et d'Analyse des Systems, Toulouse (France) (July 1983).

[Lampson et al. 1981]
B.W. Lampson, M. Paul, and H.J. Siegert (eds.), Distributed Systems - Architecture and Implementation, Springer-Verlag (1981). (Lecture Notes in Computer Science, Vol. 105).

[Ledgard and Marcotty 1981]
H. Ledgard and M. Marcotty, The Programming Language Landscape, SRA - Science Research Associates (1981).

[Liskov et al. 1970]
B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," Comm. ACM Vol. 20(8), pp.564-576 (August 1970).

[Luderer et al. 1981]
G.W.R. Luderer, H. Che, J.P. Haggerty, P.A. Kirslis, and W.T. Marshall, "A Distributed Unix System Based on a Virtual Circuit Switch," Proc. 8th Symp. Operating System Principles, Pacific Grove, California., pp.160-168, ACM (December 1981). Also in: ACM Special Interest Group on Operating Systems - Operating Systems Review, Vol. 15(5) (December 1981).

[Malitz 1979]
J. Malitz, Introduction to Mathematical Logic, Springer-Verlag (1979).

[Manna 1974]
Z. Manna, Mathematical Theory of Computation, McGraw-Hill (1974).

[McCarthy 1962]
J. McCarthy et al., The Lisp 1.5 Programmers Manual, MIT Press, Cambridge, Massachusetts (1962).

[Mead and Conway 1980]
C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley (1980).

[Metcalfe and Boggs 1976]
R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Comm. ACM Vol. 19(7), pp.395-403 (July 1976). (Also in Comm. ACM 26(1) `Special Anniversary Issue', January 1983, pp.90-94).

[Mitchell et al. 1978]
J.G. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual," Palo Alto Research Center Report: CSL-78-1, Xerox Corporation (February 1978).

[Nowitz 1979]
D.A. Nowitz, "Uucp Implementation Description," in UNIX Programmer`s Manual, Seventh Edition, Vol. 2 (January 1979). (Section 37.)

[OED 1941]
The Little Oxford Dictionary of Current English, Oxford (3rd ed. 1941).

[Organick 1972]
E.I. Organick, The MULTICS System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts (1972).

[Parnas 1972]
D.L. Parnas, "On Criteria to be Used for Decomposing Systems into Modules," Comm. ACM Vol. **15**(12), p.1053 et seq. (December 1972).

[Parnas 1974]
D.L. Parnas, "On a 'Buzzword': Hierarchical Structure," Proc. IFIP 1974, pp.336-339. (Also: pp. 335-342 in Programming Methodology Ed. D. Gries, Springer-Verlag 1978).

[Popek et al. 1981]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," Operating Systems Review Vol. **15**(5), pp.169-177, ACM (December 1981). (Proc. ACM 8th Conf. Operating System Principles, Assilomar California.)

[Pouzin 1976]
L. Pouzin, "Names and Objects in Heterogeneous Computer Networks," Proc. ECI (European Cooperation for Informatics), Amsterdam, The Netherlands (August 1976).

[Rowe and Birman 1982]
L.A. Rowe and K.P. Birman, "A Local Network Based on the UNIX Operating System.," IEEE Trans. on Software Engineering Vol. **SE-8**(2), pp.137-146 (March 1982).

[Rushby and Randell 1983]
J.M. Rushby and B. Randell, "A Distributed Secure System," Computer Vol. **16**(7), IEEE (July 1983). (Also: Technical Report 182, Computing Laboratory, University of Newcastle upon Tyne).

[Saltzer 1978]
J.H. Saltzer, "Naming and Binding of Objects," in Operating Systems, An Advanced Course, ed. R. Bayer, R.M. Graham, G. Seegmuller, Springer-Verlag (1978). (Lecture Notes in Computer Science, Vol. **60**, Chapter 3A).

[Saltzer 1982]
J.H. Saltzer, "On the Naming and Binding of Computer Network Destinations," in Local Computer Networks, ed. P.C. Ravasio, G. Hopkins and N. Naffah, North-Holland (1982).

[Scarrot 1982]
G.G. Scarrot, "Some Consequences of Recursion in Human Affairs," IEE Proc 'A' Vol. **129**(1), pp.66-75 (1982).

[Shoch 1978]
J. Shoch, "Inter-Network Naming Addressing and Routing," Proc. IEEE COMPCON, Fall 1978, pp.72-79 (1978).

[Shrivastava and Panzieri 1982]
S.K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," IEEE Trans. Computers Vol. **C-31**(7), pp.692-697 (July 1982). (Technical Report 171, Computing Laboratory, University of Newcastle upon Tyne).

[Simon 1969]
H.A. Simon, "The Architecture of Complexity," in The Sciences of the Artificial, MIT Press, Cambridge, Massachusetts (1969). (Also: Proc. American Philosophical Society **106**(6), 1962).

[Stone 1975]
H.S. Stone, Introduction to Machine Architecture, SRA - Science Research Associates (1975).

[Stoy 1977]
J.E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, Massachusetts (1977).

[Su 1983]
Z-S. Su, "Identification in Computer Networks," Proc. 8th Data Communications Symp., N. Falmouth, Massachusetts, pp.51-55, IEEE Computer Society Press (October 1983). (ACM SIGCOMM Computer Communication Review **13**(4)).

[Sunshine 1982]
C.A. Sunshine, "Addressing Problems in Multi-Network Systems," Proc. IEEE INFOCOM 82, Las Vegas, Nevada, pp.12-18 (March-April 1982).

[System V 1983]
UNIX System User's Manual, System V, Western Electric Co. Inc. (January 1983). (Reference 301-905 Issue 1).

[Tanenbaum 1984]
A.S. Tanenbaum, Structured Computer Organization, Prentice-Hall International Editions, Englewood Cliffs, New Jersey (1984).

[Tarjan 1972]
R. Tarjan, "Depth-First Search and Linear Graph Algorithms," SIAM J. Computing Vol. **1**(2), pp.146-160 (June 1972).

[Thompson and Ritchie 1978]
D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Bell Sys. Tech. J. Vol. **57**(6), pp.1905-1929 (1978). (Also in Comm. ACM **26**(1) `Special Anniversary Issue', January 1983, pp.84-89).

[Treleaven and Hopkins 1982]
P.C. Treleaven and R.P. Hopkins, "A Recursive Architecture for VLSI," Proc. 9th Symp. Computer Architecture, pp.229-238, IEEE Computer Society Press (April 1982). (ACM SIGARCH Newsletter, **10**(3), April 1982).

[Treleaven et al. 1982]
P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, "Data Driven and Demand Driven Computer Architecture," ACM Computing Surveys Vol. **14**(1), pp.93-143 (February 1982). (Also in Japanese translation in BIT **12** (1983); and in part in Supercomputers: Design and Applications K. Hwang and R. Khun (eds.) IEEE Computer Society Press 1984).

[Turner 1979]
D.A. Turner, "A New Implementation Technique for Applicative Languages," Software Practice and Experience Vol. **9**, pp.31-49 (1979).

[Vuillemin 1974]
J.E. Vuillemin, "Correct and Optimal Implementations of Recursion in a Simple Programming Language," Journal of Computer and System Sciences Vol. **9**, pp.332-354 (1974).

[Watson 1981]
R.W. Watson, "Identifiers (Naming) in Distributed Systems," pp. 191-210 in Distributed Systems - Architecture and Implementation, (Munich, March 1980), ed. B.W. Lampson, M. Paul, H.J. Siegert, Springer-Verlag (1981). (Lecture Notes in Computer Science, Vol. **105**, Chapter 9.)

[Whitehead 1911]
A.N. Whitehead, An Introduction to Mathematics, Thornton Butterworth (1911).

[Wilner 1980]
W. Wilner, Recursive Machines, In `VLSI: Machine Architecture and Very High Level Language', ed. P.C. Treleaven, ACM Computer Architecture News **8**(7) December 1980 pp. 27-38 (Technical Report 156 University of Newcastle upon Tyne).

[Wirth 1983]
N. Wirth, Programming in Modula-2, Springer-Verlag (Second Edition, 1983).

[Woods and Wheen 1983]
J.V. Woods and J.T. Wheen, "MU6P: An Advanced Microprocessor Architecture," Computer J. Vol. **26**(3) (1983).

[Wulf et al. 1976]
W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard programs," IEEE Trans. on Software Engineering Vol. **SE-2**(4), pp.253-265 (1976).

[Xerox 1981]
Xerox, "Internet Transport Protocols (Xerox System Integration Standard)," XSIS 028112, Xerox Corporation, Stamford, Connecticut (December 1981).