Abstract: This report documents the progress made to date on the Science Research Council - sponsored project on the Design of Highly Reliable Computing Systems. The background to the project is discussed, and brief descriptions given of certain computer architecture and program structure features that are being included in an initial experimental system implementation. These features provide a hierarchical structuring for software error detection and system error recovery.

Research on Computing System Reliability at the University of Newcastle upon Tyne, 1972/73.

by

B. Randell

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 57

c 1974 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
RANDELL, Brian

Research on computing system reliability at the University of Newcastle upon Tyne, 1972/73. [By] B. Randell

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory 1974.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series No. 57.)
CONTENTS

1. Introduction
2. Background to the project
3. Project Hardware
   3.1 The Magnabus
4. Detection of Program Errors
   4.1 The Emulated Machine Language
5. Addressing and Protection
   5.1 Recursive Virtual Machines
6. System Error Recovery
   6.1 Recovery Blocks
   6.2 The Recursive Cache
   6.3 Further work on error recovery
7. The Initial Experimental System
8. Project Organisation and Schedule
9. Conclusions
10. Acknowledgements
11. References

Appendix 1 System Reliability Memoranda

Appendix 2 Initial Configuration of Project Computing Facilities
1. Introduction

The formal starting date of the Science Research Council-sponsored research project on the design of highly reliable computing systems, which is directed by the author, was January 1st 1972. Prior to this a preliminary grant had been used to assist us in the preparation of a report [23] which incorporated a survey of the reliability needs, and existing techniques, at a number of large-scale computing installations within the U.K. (Some of the material in this report was also included in an invited survey lecture given at the IFIP 71 Congress in Ljubljana [24].)

Grants received so far from the Science Research Council for the project cover salaries and recurrent expenses for a four year period and the costs of the initial configuration of the project's computing facilities. These grants total £147,604. The project staff consist of four senior research associates, Dr. T. Anderson, Mr. R. Kerr, Dr. A.J. Mascall and Mr. R.M. Simpson, one electronic designer (part time), Mr. K. Heron, and one secretary, Miss J.A. Lennox. (Several academic staff, in particular Dr. J. Eve, Dr. H.C. Lauer, Dr. C.R. Snow and Mr. D. Wyeth and a varying number of Science Research Council-sponsored post-graduate students are also closely associated with the project.) Financial provision has also been made for a number of fellowships, totalling four years duration, to be held by visitors to the project - the two senior visiting fellowships awarded so far have been held by Professor J.J. Horning of the Computing Systems Research Group, University of Toronto, and Mr. P.M. Melliar-Smith, who until recently was with General Electric Computers Limited. In addition a separate grant of £11,228 has recently been received from the Science Research Council for an independent, but nevertheless closely related project, of two years initial duration, to investigate techniques for the reliable operation of data base systems. This investigation is to be carried out by Mr. Melliar-Smith.

The objectives of the research project, as stated in the original application, are "to develop, and to give a realistic demonstration of the utility of, computer architecture and programming techniques which will enable a system to have a very high probability of continuing to give trustworthy service in the presence
of hardware faults and/or software errors, and during their repair.
A major aim will be to develop techniques which are of general utility,
rather than limited to specialised environments, and to explore possible
trade-offs between reliability and performance."

The approach that was sketched out originally, which involves
the implementation of an emulator for the computer architecture that
we develop, and the programming of a rather skeletal operating system
and some simulated application programs to be executed using this
emulator, is being followed. It is proving to be a very valuable line
of attack on a set of problems whose difficulty we perhaps realised more
clearly now than when we set out. We believe we have indeed made
significant progress, particularly in the area of computer architecture
and program structure.

The initial hardware configuration for the project, a twin-
processor PDP-11/45 system, is due to arrive within a few months, and
the detailed design and implementation of a preliminary experimental
system is well under way. The present report attempts to document the
work done so far on the project, and to give an overview of the ideas
that form the basis of the experimental system.

Section 2 of this report discusses how we are concentrating our
efforts on the problems of detecting residual program errors at run time,
and of error recovery following either hardware or software errors. It
describes how our work on the development of new computer architectures
and program structures is based on our belief in the practical value of
recursively-defined hierarchical structures, and of explicit programmer-
supplied redundancy.

Section 3 explains the factors that went into the choice of hard-
ware for the project, and our plans for developing a special hardware
device, which we call a Magnabus, for centralised control of the way in
which our set of PDP-11 processors, peripheral devices, and memory modules
are configured together.
Our work on the run-time detection of residual program errors is discussed in Section 4. A description is given of those aspects of the computer architecture that we have devised which relate to the processing of individual programs compiled from a block-structured typed language - this architecture is in some ways a descendant of that of the B5000 and the B6700. The other main topic covered is our ideas on using the problems of providing formal or informal proofs of correctness of a program as guidance as to the incorporation of useful redundancy into the program.

Section 5 documents the work that we have done on addressing and protection, which has led to the idea of a 'recursive virtual machine'. This provides an environment for the hierarchical structuring of an operating system and the application programs whose execution it is controlling.

The 'recovery block' scheme that we have developed for structuring programs so as to make the extra code that is incorporated for purposes of error detection and recovery explicit is described in Section 6. Because of possible patent claims, only a brief discussion is given of some of the characteristics of the associated 'recursive cache', a hardware scheme for automating the restarting of a program after an error.

In Section 7 we give a brief description of the way in which our first experimental implementation is being planned in order to provide us with a first test of our ideas as the emulated 'machine language, and the recursive virtual machine and recursive cache concepts.

Finally, in Section 8, we give some brief comments on the project's organisation, and revised schedule.

2. Background to the project

There is surely no need to justify research on the subject of computer system reliability, which we define as the correctness and continuity of operation of a system. However, it is worth explaining, and attempting to justify, the particular choice of topics that we
have made from this very large subject area, and the particular attitudes which underlie the work that we are doing.

Viewing a computing system as being composed of a set of interacting hardware and software "components", the reliability of the overall computing system can be thought of as being based on
(i) the reliability of the individual components
(ii) the ability of the system to cope with less than absolute reliability on the part of one or more of its components.

It is this latter topic that we are concerned with, a topic which has recently been termed "fault-tolerant computing" [25]. Thus on the software side, we have no (direct) interest in topics such as 'program correctness proofs', or 'debugging aids'. These have the praiseworthy aim of ensuring that software components have error-free designs; our perhaps cynical assumption is that for some time to come it will be worthwhile continuing the practice of attempting to obtain useful work from complex software systems even in the knowledge that there is no "guarantee" that they are free from programming errors.

Unreliability in major hardware components (such as processors, devices, etc.) can be due not only to design errors, but also to operational errors caused by, say, physical deterioration, although the boundary between such failures and design errors is sometimes rather blurred. Hardware designers have developed many techniques for error detection and, in many cases, automatic error correction. Our basic assumption, which has received the agreement of several hardware designers, is that the hardware should have the responsibility for all hardware error detection; those errors which are not automatically corrected by the hardware will be signalled, and thus become the responsibility of the software, which will not otherwise concern itself with hardware errors.

We are thus concentrating our attention on the problems of software error detection, and system error recovery (i.e. following either hardware or software errors). Indeed we have so far tended to emphasise the problems of system recovery after software errors. This has been in the expectation, which so far seems to have been reasonably well-founded, that techniques which are adequate for software error
recovery will provide much of what is needed for hardware error recovery problems.

Having thus delineated the scope of the project with respect to hardware and software, it is necessary to justify the inclusion of the design of computer architecture, i.e. the hardware/software interface, in the work of the project. With a few notable exceptions, existing machine architectures still are more heavily influenced by microscopic considerations (e.g. word/byte access, arithmetic, etc.) than by the structural and logical characteristics of the tasks they are set. It is expected that architectures influenced by more macroscopic considerations such as block structure, control structure, composite data manipulations, etc., will have the effects of eliminating some of the error-prone processes in conventional programming, and assisting the detection, diagnosis, and the determination of the extent of errors, and recovery from them. In particular, tests which should be performed frequently, but which almost invariably result in the same decision being made, are prime candidates for being implemented in hardware, rather than software, so that they can be made in parallel with other actions. The development of computer architecture has been marked by the identification of many such tests, e.g. arithmetic overflow, interrupts, and array bounds checking. As is described in Section 6, we have found various other such tests concerned with error detection and recovery, which in our view merit hardware implementation.

However, the detailed design of such hardware facilities, and the accurate determination of performance and reliability characteristics, lies outside the scope of the project. Therefore, in order to defend ourselves against a possible charge of 'defining problems away', we have tried to avoid postulating computer architectures whose complexity significantly exceeds that of various current computers, such as the B6700 [21] or the MU5 [16]. Within the project our chosen computer architecture is being implemented by an interpreter (emulator) on a PDP-11/45 - this should suffice for first assessments of (scaled) performance/reliability trade offs. A more realistic implementation, via microprogramming on the laboratory's B1700, or in hardware, is a possibility for later.
Another major characteristic of the project is our emphasis on hierarchical structures. Complexity is one of the major causes of unreliability, and we view hierarchical structure as one of the most successful means of mastering complexity. (This point has been well argued by Simon [25].) Thus the program structure that we have devised for the specification of error recovery actions is hierarchical. As a result, in contrast to the situation with less-structured systems, the occurrence of further errors during the error recovery process itself does not cause special difficulties.

The emphasis on hierarchical structure has also caused us to abandon our original intention of avoiding effort on the design of addressing and protection mechanisms by the appropriation of some existing design based on the idea of capabilities [9]. Instead we have produced a scheme whose recursive nature facilitates the implementation of hierarchical system software structures more in keeping with the other aspects of the project.

As important to us as the concept of 'hierarchy' is the notion of 'redundancy' - we view all error detection as being based on the provision of useful redundancy whose consistency can be checked. When one is concerned merely with operational failures of hardware components, multiple instances of each different type of component can provide the required redundancy. However design errors, such as those that occur in software, require a different kind of redundancy, namely redundancy of specification. Indeed one view of current research on automated proofs of correctness of programs [11] is that they consist of compile-time checking of the consistency of a program and its accompanying assertions, which are a programmer-supplied redundancy. We however are concentrating on checks that are performed during the running of the actual system. We thus aim to detect errors that actually occur, rather than use sophisticated theorem-proving techniques which aim to identify potential errors which some particular, and perhaps unlikely, set of input data would cause to occur.

9.
Many existing systems incorporate extra coding whose purpose is to provide error detection and recovery facilities [23]. However such extra text usually looks like a coding mistake or an accidental inefficiency, requiring extensive commentary if its real purpose is to be understood by a reader. A major aim of our investigation has been to find suitable means of making this 'redundant' program text, and the activity it engenders, readily distinguishable from the text and the activity of the main system. This will assist readers of the text; more importantly, one can imagine facilities being provided either at compile-time or at run-time, which take advantage of the identifiability of such text and activity, for example in order to arrange the automatic exercising of the error detection and recovery mechanisms.

This implies, correctly, that we are including the design of programming language features within the domain of our project. However every effort has been made to minimise the amount of programming language design that is undertaken, so as to avoid at least one trap which it is all too easy for a project such as ours to fall into. Instead, the project is basing its efforts very closely on the Project SUE system language [7], for which compilers for both S/360 and PDP-11 have been implemented at Toronto, and made available to us. The arguments that led to the choice of the SUE system language are documented in project memoranda SRM/25, 26 and 30. (Appendix 1 provides a listing of memoranda issued to date.) The SUE system language is closely related to PASCAL [26] but incorporates certain modest, but worthwhile, improvements - SRM/35, which is a summary of a lengthier critique of PASCAL [15], and SRM/39 document these language matters.

3. Project Hardware

Ideally, the selection of the quantity and type of hardware to be obtained for the use of a research project such as ours would be left until virtually all the preliminary design work had been completed. In practice, even in the best of circumstances likely delays make it necessary for the choice of hardware to be one of the first problems to be tackled.
An important aspect of the planned programme of research was, and remains, the giving of a realistic demonstration of the techniques that are developed. Therefore the final stage in the original four-year schedule for the project called for a working demonstration system, incorporating multiple instances of each hardware resource class (processors, devices, etc.) which would be interfaced to a separate system which in effect had the taste of simulating the external world. This separate system would therefore generate the workload, receive the results, inject simulated hardware and software errors into the demonstration system, and monitor its behaviour and performance. The proposal to use simulated rather than actual workloads was based on the realisation that a system whose developers are primarily interested in its abilities to cope with errors, and who will therefore, where necessary, be testing its abilities to respond to artificially induced errors, will hardly be attractive to real life users. However this meant that considerable economies and short cuts could be taken in the hardware configuration, and in the software facilities provided.

Before the start of the project a detailed survey of possible hardware systems was undertaken. This was documented in SRM/4, 9, 13 and 14. Since the primary use for the processors was for implementing an emulator, we favoured systems with a low-level and flexible instruction set. The possibility of implementing the emulator on a microprogrammable processor was discarded because at that time, at any rate, the unattractiveness of the design of the various processor's micro-instruction sets, and the comparative difficulty of making modifications to microprograms, outweighed the possible processing speed advantages. The second important criterion was flexibility of interconnection of major hardware components. We have yet to decide what forms of interconnection are preferable—shared main memory or backing store, channel-to-channel connection, etc. This led us to the idea of a very flexible means of configuring systems based on a device for interconnecting linear bussing schemes, such as the Unibus of the DEC PDP-11 range. The method involves a centralised switch, the Magnabus, which is briefly described below.
For these reasons we proposed that a PDP-11 system with multiple processors be purchased for the project. The original project schedule called for some such hardware to be available six months or so following the start of the project. In the event, the Science Research Council asked for further evidence to back up our request. This was produced and even included a fairly detailed specification (documented in SRM/19) for a device, based on equipment from another manufacturer, which attempted to match the facilities that could be provided by a Magnabus. As a result we received, just a few months after the start of the project, S.R.C. approval for an initial configuration consisting of a twin-processor PDP-11/45 system. (The configuration is detailed in Appendix 2.) Unfortunately, it was not until April 1973 that, following the receipt of the approval of the Department of Trade and Industry to this choice, an order could be placed with Digital Equipment Ltd., by the Central Computer Agency, on our behalf. In the meantime worsening delivery delays have resulted in the likely installation date being April 1974, some nineteen months later than called for in our original schedule.

The unexpected effort that had to be diverted during the early stages of the project into the comparative evaluations, and negotiations over the choice of hardware, has of course taken its toll. The effects of the delay have been aggravated by the fact that its extent was not predictable.

However in retrospect, we now view the delay as having had some beneficial effects. It has enabled us to gain a much greater understanding of the problems that we are attempting to tackle, and to develop our ideas on computer architecture much more fully, prior to getting enmeshed in implementation problems. The interest, and likely value to us, of our planned first set of experiments on the PDP-11/45's is therefore considerably increased.
One effect of the delay has been to increase the extent to which the laboratory's main computing system, the IBM 360/67, is used for the preparation of software for the PDP-11's. Right from the start we had planned to have some interconnection between the project machines and the Model 67. These plans have evolved, as circumstances have changed, and are documented in SRM/65. The connection will be made via a DEC DP-11 DA synchronous communications interface, a communications cable terminated by two modem simulators with a maximum capacity of 9600 baud, and a line set on the IBM 3705 controller. The supporting software for the IBM end of the link already exists, since the PDP-11/45's will appear like HASP work stations to the Model 67. The software for the PDP-11/45's is being provided to us by the Edinburgh Regional Computing Centre.

3.1 The Magnabus

The PDP-11 Unibus is a linear bussing scheme which connects a single processor to the various memory modules and peripheral device controllers. It provides a single linear address space, for peripheral device controllers as well as memory locations. Each device controller or storage unit that is connected to the Unibus contains an indication of the set of addresses by which it is to be known to the processor. Physically the bus contains 56 bi-directional lines, 16 of which are for data transfer, and 18 for address specification, with the remaining lines being used for control, synchronisation, processor interruption, etc. There is a bus arbitration control unit at one end of the Unibus, and an electrical termination at the other. The processor, as opposed to the device controllers and memory modules, is connected to the Unibus through the arbitration unit, which is in fact built into the processor.

We require, for the later stages of our project, a means of connecting several processors together into one system. Our proposed means of doing this involves a centralised switch, which we have termed the Magnabus, which allows the controlled interconnection of several Unibusses. The Magnabus that we have specified would be a very flexible and powerful mechanism. We are at present attempting
to define a small series of intermediate stages so that our hardware designers can first implement a comparatively simple interconnection mechanism, and progress gradually towards the full facilities called for by the complete specification. However we will here confine ourselves to giving a very brief outline of the specification and likely form of the complete Magnabus.

The Magnabus has two separate aspects. In its dynamic aspect its purpose is to allow several processors to have access to shared memory and devices on a word-by-word basis. In this sense the Magnabus can be regarded almost as a multiplexor. In its static aspect its purpose is to allow the reconfiguration of processors, devices, and memory modules into one or several computer systems. Because the static function of the Magnabus switch is to include the ability to partition the set of components into separate possibly untested systems, it must not be under the control of any of the switched components. Rather it should be regarded as a "manual" switch with relatively long periods (i.e. half-an-hour or more) between reconfigurations.

In addition to connecting components into systems, we also require our switch to facilitate the renaming of these objects. For example, if all of the memory were divided between two separate systems, each system might want its address space to begin with location zero and increase linearly. The switch must map the logical memory names from the two systems into the absolute memory addresses recognised by the hardware components. As with the setting of the switch itself, this mapping must be static and not under programmed control of any component in order to protect the hardware configuration from faults. Thus this mapping is quite distinct from the program-controlled mapping functions, such as provided by paging and segmentation facilities, that might be used in the system or systems that are configured with the switch.

The Magnabus is regarded as providing facilities for the project, rather than being itself a part of the main programme of research into questions of reliability. We have therefore chosen to keep the Magnabus quite separate from our investigations into possible future computer
Figure 1.
Directly connected network.

Figure 2.
Distributed cross-bar network.
architectures by insisting that it should not affect the way in which PDP-11 processors are ordinarily programmed. Thus we are deliberately eschewing the "opportunity" that the interposition of a hardware unit of our own design between a PDP-11 processor and its memory modules and peripheral devices would give us to introduce, for example, our own special address mapping facilities. Instead the Magnabus is required to be as neutral as possible with respect to the PDP-11 system design, and our aim is that it should not require any changes to be made to existing PDP-11 hardware or software.

Centralised configuration control (see for example, Blakeney et al [5]) has great advantages for our project in facilitating the switching at different times of the day, between experiments which involve the full complement of processors, storage and devices, and periods when the independent use of subsets of the total complement would suffice. It should thus increase the overall utility and productivity of the computing equipment. The centralisation of the dynamic aspect of the switch is a separate matter, which merits discussion.

Other possibilities include direct cable interconnections between all appropriate processors, storage units and device controllers, such as in the Modular 1, or distributed cross-bar networks such as found in various systems, including the 360/67. (These are illustrated in figures 1 and 2 respectively, which use the FMS notation of Bell and Newell [3]). It is the existence of the Unibus, which is itself a sophisticated switching system, and arguments involving the relative costs of cables, connectors and integrated circuits, that has caused us to prefer a centralised switch, illustrated in figure 3, which works by selective interconnection of Unibusess.
Both the horizontal and vertical links shown in figure 3 are Unibus cables, and have electrical and logical characteristics identical to that of a normal complete Unibus, so that standard hardware components can be connected to them. As with the normal Unibus, components can be either active, passive, or both. An active device is one which can request control of the bus, place an address on the address lines (thereby broadcasting it to all devices), and either transmit or receive data or exercise a control function. A passive device is one which cannot request control of the bus; it can only recognise addresses and either accept or return data. In the PDP-11 system, a central processor is a strictly active device, while a memory is strictly passive. The controllers for peripheral devices (i.e. Kf and Ks) are both active and passive: they are active in the sense that once they are operating they can move data to and from memory directly or transmit interrupts on the interrupt line; they are passive in the sense that their control registers are treated just like memory cells.
Figure 4
A more detailed representation of the Magnabus
The Magnabus logically is a switch which allows the \( n \) horizontal busses to access the \( m \) vertical busses. It is a cross-point switch providing up to \( \min(n, m) \) possible simultaneous conversations. Once it has been set up in accordance with a particular configuration requirement only certain interconnections between horizontal and vertical busses will be permitted. The actual routing of information through the switch will depend on the addresses that are issued by active devices, and the addresses that have previously been set up in the switch as part of the configuration specification. The configuration specification will be set up by a 'Magnabus controller' which could be entirely manual, but is more likely to involve a small processor - this would speed reconfiguration, and reduce the risk of errors.

The Magnabus is shown in more detail in figure 4. It is built from 'switch slices', which incorporate 'switch elements'. The switch element is the part of the Magnabus which connects a particular vertical bus (at whose end a processor is situated) to a particular horizontal bus. The switch slice incorporates all the switch elements capable of connecting the vertical busses, and hence processors, to a particular horizontal bus. In addition, each slice contains the arbitration logic to determine, in the event of simultaneous requests for devices on the associated horizontal bus, which vertical bus wins, i.e. which switch element \( S \) will be enabled. The switch slice is also responsible for routing bus requests and responses for processor interrupts. Associated with each switch slice is the indication of the set of addresses which it is to recognise, and the set of addresses to which these are to be translated.

Thus when an appropriate address appears on the address lines of a vertical bus, the corresponding switch slice will connect this bus to the required horizontal bus, and pass on the translated form of the address. This translated address will then presumably be recognised by the appropriate device on the horizontal bus. Similarly, requests will be routed, with address translation, from horizontal busses to vertical busses where they will be recognised either by the processor, a private device, or another switch slice. In this latter case, the request will thus be routed from a device on one horizontal bus to a
device on another horizontal bus.

This then is a very brief summary of our ideas on the Magnabus. Clearly many details have been left out, in particular those concerned with performance and reliability considerations, interrupt handling, and the complications arising from the certain details of the way in which some devices use the Unibus. (More detailed discussions can be found in SRM/16, 18 and 56.) These complications are such that there are likely to be limitations as to what components can share a single horizontal bus—at present we envisage that such a bus could be used for either one or more memory units, one or more I/O device controllers, or a single disc controller.

4. Detection of Program Errors

The problem of run-time detection of residual program errors has been studied on two levels, so to speak—the programming language and the machine language level. In view of the project's interest in hierarchies and redundancy, it was natural that we should assume that programs (whether part of the system software, or separate application programs) should be written in block-structured, typed languages. As mentioned earlier, we have chosen to base our work on the SUE system language.

The SUE system language, in common with various similar languages, is designed to facilitate extensive compile-time type checking. This is of course a very laudable aim, particularly for present-day computer architectures. We have taken the perhaps extreme view that type-checking should also be performed with vigour at run time, particularly at assignment. Various arguments can be put forward for this, all based on the view that the safest time to check the validity of an operation is immediately before it is performed. (By this means, worries about whether a compiler was used, whether it had any errors in it, or whether the machine mal-functioned during the compilation, and whether the object program had been corrupted, are all allayed.) The aim is for the language of the machine we shall emulate to resemble the SUE source language in its structure and redundancy; the main role of the compiler
is then that of taking advantage of language definition rules (such as concerning the precedence of operators, and the irrelevance of program lay-out and the choice of symbols for identifiers) that allow binding to be used in order to increase the efficiency of the run time system. In fact the EML (Emulated Machine Language), which is briefly described in Section 4.1 below, can be seen to have some similarities to the Algol-oriented B6700 machine language, but with a restricted (goto-less) control structure, and reinforced checking of operand types and accessing.

There has in fact been a small amount of work on programming language design, mainly concerned with control structures such as exits from cycles and loops (SRM/48) and on restrictions which would have the effect of preventing the writing of non-terminating programs. The theoretical consequences of requiring that bounds be given for the number of repetitions of each loop and recursive activations of each procedure were studied, and the conclusion reached that they were outweighed by the practical advantages to be gained. (A detailed account of this study is given in SRM/63)

This work was in fact an outgrowth of earlier work in which the concept of a 'safe' algorithm was developed (see SRM/23). The work involved several successive attempts to design an algorithm for a particular function and to provide an accompanying 'correctness proof'. This provided an unintended demonstration that proofs can be as prone to errors as programs. Indeed they may be even more error-prone — proofs may be informal, while programs must of necessity be precise; moreover mathematical arguments may induce a false sense of security. These problems led to the idea of an algorithm's 'safeness', as opposed to 'correctness'.

Suppose we wish to write a program conforming to some specification P. We can instead endeavour to design a program which when executed either conforms to P, or gives an indication that for some reason it was unable to do so. It may then be possible to prove correctness for the new specification 'P or set (error)'. This proof is said to be of safeness with respect to P, the program itself being termed safe. A
proof that the program in fact never gives an error indication (i.e.,
actually conforms to P) is now an argument for the program's usefulness,
but is more conveniently referred to as a proof of the correctness
of a safe program. The advantage which derives from this distinction
is that it may be much easier to supply a proof of a program's safeness
than of its correctness. Even when a proof of correctness can be given,
a safeness proof should be much simpler, and in consequence more likely
to be valid. In a sense a safe program is insured against the risk of
its proof of correctness being invalid.

In fact the points at which an attempted correctness proof is
intractable, or difficult, or even just obscure, can be used as an
indication of where extra redundancy, in the form of run-time checks,
are worth inserting. This should result in the provision of a modified
algorithm with an accompanying much-simplified proof of safeness.
This is very important, since the provision of arbitrary redundancy
in a complicated program may even decrease its reliability (as well as
its performance).

The most extensive test so far of this idea of 'proof-directed
redundancy' has been the initial design of a simple filing system
(documented in SRM/38). A previous design (see SRM/22) had been made
in which the aim was to retain redundant book-keeping information
and to choose the sequence in which actions were carried out so care-
fully that it would always be possible to recovery from a system
crash. (This design turned out to have several features in common
with the file system for the PRIME project [2].) An attempt was
made to implement this design in order to quell doubts raised as to
the adequacy of the informal logical arguments that it was based on.
This attempt was abandoned, it having been realised that 'mere pro-
gramming errors' were turning out to be a much greater cause for concern
regarding the system's reliability.

The second design was very much influenced by the work on safe
algorithms. The aim was to design a file system such that there was
a high degree of confidence that the data structures on backing store
would remain intact, whilst conceding that there might well be software errors within the file system that would cause it to stop, or rather to report the failure. This was done by isolating the actions which could write to backing store to a few very simple procedures, and making extensive use of run-time checks, guided by the task of providing informal proofs of safeness. The system was fully implemented, and although various (quite accidental) programming errors were encountered, the run-time checks were completely successful in maintaining the integrity of the stored data structures.

In summary therefore, we are relying on two very different forms of redundancy for the detection of programming errors - the redundancy that is inherent in any block-structured typed language, and which programmers are already well accustomed to, and the redundancy given by programmer-provided assertions, which are treated as run-time checks. The choice and placement of these assertions is regarded as an important part of the programmer's task, but one which we think can be very usefully (and "respectably") guided by the results of efforts to provide correctness proofs, whether of a formal or informal nature. We have in fact developed a method of structuring programs, and some associated hardware mechanisms, which facilitate the design of assertions, and make the distinction between their text and that of the main program explicit, but discussion of these matters will be deferred to Section 6, on error recovery.

4.1 The Emulated Machine Language

The global problems of addressing and protection amongst the set of programs which make up the operating system and the different applications which at any given time exist in the computing system have been tackled separately, and without undue commitment to the use of a particular programming language - their discussion is deferred to Section 5. In this section we attempt a very brief overview of the way in which translated individual SUE system language programs will be represented. (More detailed descriptions can be found in SRM/31, 49, 53 and 67.)
The Emulated Machine Language is intended specifically, although not necessarily exclusively, for translated SUE programs—the absence of a branch instruction is the most obvious "missing" construct which would tend to restrict its use over a wide range of languages. Because there is no possibility of arbitrary branching and because all control constructs may be nested but not interleaved the structure of an EML program has the form of a tree. This tree is a simple transformation of the parse tree of the source program with each node corresponding to an EML statement, primary, or operation, and containing a 'fragment'—a fragment is either (i) single instruction or (ii) a group of instructions and/or references to other fragments (nodes). The various instructions include basic arithmetic and logical operations, storage access instructions, instructions for beginning and ending blocks and procedures, instructions representing declarations, and sequence control instructions.

A general impression of the Emulator Machine Language can be obtained from figures 5 and 6. Figure 5 gives a simple program which, to avoid unnecessary description, has been written in a version of the SUE system language that has had its syntax altered to look more like Algol. In the EML version of the example, given in figure 6, certain fields in instructions which are necessary for the Emulator but not for understanding the program are marked with an asterisk.

```
begin
  type numbers = (1 to 10000);
  declare
    integer i;
    array (1 to 12) of numbers (A);
  i := 1;
  cycle
    A(i) := 0;
    i := i+1;
  if i = 11; then: exit else: end
end
```

Figure 5

The program corresponding to the EML program of figure 6.
During execution the Emulator will maintain four segments for each separate process (or 'virtual machine' - see Section 5.1).

(i) the (read-only) code segment, containing the EML program being interpreted

(ii) the data segment, containing the data stack and the cache (see Section 6.2)

(iii) the control segment, containing the block/procedure display, dump areas for registers and other process state-word details

(iv) a segment containing areas that have been or are to be allocated to descendant processes.

Variables are divided into three classes. Scalars occupy a single cell on the Data Stack; repeaters, which are arrays of scalars or repeaters, and occupy as many contiguous cells on the Data Stack as there are elements; and structures, which are like Algol W records. A scalar cell has four fields: machine tag, null bit, variable type and value. The machine tag is a coarse classification of the scalar and is used by the Emulator for type checking during expression evaluation. The variable type is a fine classification of the scalar consisting of an index into a Type Table (see below), and is used for type checking at assignment. The null bit indicates that a Data Stack cell has been reserved but not initialised. The value field is self-explanatory.

The Type Table contains entries for all the implicit types supplied as primitives of the language, and for all those types constructed by the programmer from the primitives. Each entry contains information describing the machine tag associated with variables of a given type, the range of values they may take on, the space they will require on the stack, and the symbolic name of the type.

The Type Table, and the tree-structured form of programs, are the main features that distinguish the basic Emulated Machine Language from a more conventional stack-oriented machine language. The actual amount of redundancy in the representation of EML programs and the amount of run-time checking to be performed are still a subject of debate - the aim is that only statements which make sense in the SUE
Figure 6

An example of an EML program.
system language should appear in the EML, all other bit configurations must be non-executable or result in program faults. These questions however are largely orthogonal to the other architectural features of the emulated machine to be described in the next two sections.

5. Addressing and Protection

In recent years addressing and protection provisions have rightly come to be regarded as among the most important aspects of the design of computer architecture. This is certainly the case with respect to questions of system reliability. The ability of the protection mechanism to prevent inappropriate usage of the various objects, such as files, data areas, devices, processes, etc., which make up a system once again involves the checking of the consistency of useful redundancy. Without the provision of explicit information, keyed for example to passwords or job numbers, no checking would be possible. The reliability of this information, and of the operation of the protection mechanism itself, is therefore crucial. The ability of a system to recover after a detected error depends on an ability to identify what parts of the system can still with justification be relied on. To a certain extent a programme of explicit checks can be undertaken in order to estimate the extent of the damage, but if reliance can be placed on a protection mechanism one has a priori knowledge of which parts of the system cannot have been affected.

In some quarters, attempts have been made to separate the issues of protection and addressing, and to be content with an unstructured, one-level, view of the set of objects constituting a system [14, 17]. In our view a hierarchical model is much preferable – for example this enables one to represent objects formed by the combinations, regrouping and renaming of other objects, and so deal quite simply with what could otherwise be the difficult questions of creation and deletion of objects. With such a model it becomes clear that addressing and protection are very closely related indeed, with much protection being given by controlled addressability. (A more extensive discussion of these matters is given in SRM/7 and 24.)
The hierarchical addressing/protection regime that is being developed in association with the project has its origins in an investigation of the necessity and desirability of having a distinction between privileged and non-privileged modes of operation of a processor (documented in SRM/15 and 32). Now a privileged state is provided to prevent faulty or malicious programs (which are assumed to consist only of non-privileged code) from gaining control of the system or its vital resources. This imposes a two-level structure on a system, with all privileged functions being concentrated in one level—a restriction which we do not find acceptable. Instead the scheme that we are incorporating into the architecture of our emulated machine is based on the notion of a 'recursive virtual machine'.

5.1 Recursive Virtual Machines

The virtual memory concept has been extended in systems such as CP/67 [19], which provides what are called 'virtual machines' to software and application programmers. These virtual machines resemble the existing hardware exactly, so that it is possible to support several different operating systems for that hardware on the same real machine simultaneously. They also provide the ability to test a new copy of the system as a user job under itself and, recursively, to support facilities similar to those supported by the given system. However the underlying computer architecture is the source of considerable performance problems, so that although in theory operating systems could be nested within each other indefinitely, in practice the depth of nesting must be severely limited, typically to no more than two levels.

The ability to run a copy of an operating system under itself is not an important goal on its own, but it incorporates some of the properties of machine architecture which we believe to be most appropriate for building well-structured systems. We are interested in the possibility of constructing an operating system as a structured hierarchy of virtual machines for implementing structured hierarchies of logical resources. Such machines can then be viewed as the units of system programming, protection, and modularity—indeed we take the
step of identifying the concepts of virtual machine and process.

Our recursive virtual machine scheme has similarities to that of Goldberg [13], but was developed independently. Virtual machines can be created recursively within other virtual machines without sacrificing efficiency, the privileged state is eliminated, no programs need be bound to absolute memory or resource addresses, interrupts are associated with the level and context in which they must be handled, all protection is achieved through the virtual memory mechanism, and any primitive machine function is potentially available to any virtual machine at any recursive level without the need for interpretation. Unlike the CP/67 system, there is little need to return to previous levels to have sensitive instructions or operands interpreted. Thus, we expect programs to run at nearly the same speed in a virtual machine as on bare hardware.

<table>
<thead>
<tr>
<th>Functions of privileged state</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control access to I/O and similar devices</td>
<td>Make device addresses part of addressable, hence virtual, memory; require devices to operate with virtual rather than real addresses.</td>
</tr>
<tr>
<td>Prevent programs from altering address mapping tables and control registers</td>
<td>Allow any program to create a virtual machine as a subset of its own space. Perform all other changes as part of hardware context switch.</td>
</tr>
<tr>
<td>Control allocation of central processors</td>
<td>Allow any process to control allocation amongst its descendants, but make reallocation as a result of interrupts a hardware function.</td>
</tr>
<tr>
<td>Control masking, acceptance, and routing of interrupts</td>
<td>Associate specific interrupts with specific processes; use synchronizing primitives such as P and V [10].</td>
</tr>
<tr>
<td>Prevent programs from becoming more privileged</td>
<td>Abolish privileged state; allow transfer of control to more privileged virtual machine only by (addressable) procedure call.</td>
</tr>
</tbody>
</table>
A fairly full description of the scheme has recently been published [18], so only brief details will be given here. Table 1, taken from the published description, summarises the forms of protection that privileged state provides on conventional machines, and indicates the alternative forms provided by the recursive virtual machine scheme.

Each virtual machine has an associated virtual memory, made up of a collection of segments. The basic scheme is independent of many details of the system architecture, such as the form the instructions and data take. It is irrelevant, for example, whether the machine is a stack machine or a general register machine. The virtual memory is assumed to encompass the state information of a process that can be inspected or manipulated by that process itself - e.g. the programmable registers such as program counters, accumulators, index registers, condition codes, local timers - even though it may sometimes reside in fast hardware registers.

Associated with each virtual memory is a mapping (in tabular form) which specifies, among other things, which segments are identified with which segment names. This mapping is called the segment table for the virtual memory - the locations of entries within the table provide the set of non-negative integers which are used as segment names.

A virtual memory is either:

1. identically equal to the hardware memory, in which case each of its segments is an identifiable hardware segment; or

2. a subset of another virtual memory, in which case each of its segments is a subset of a segment in that containing virtual memory.

Thus the notion of virtual memory may be made recursive. In the first case, the segment table of a virtual memory is implicit in the hardware memory accessing circuitry. In the second case, its segment table specifies the name and part of the containing segment in the containing virtual memory. An entry of the segment table in this
case is illustrated in Figure 7 and is called a descriptor. It specifies how big the segment is (the size), any type and control information to describe the segment, the location of the first bit of the segment (expressed as an offset from the beginning of the containing segment), and the name of the containing segment (expressed in terms of the containing virtual memory). The ith descriptor in a segment table describes the segment which is associated with the ith segment name of the virtual memory.

The set of virtual memories in a system at any instant form a hierarchy based on the containment relation. Each segment table is a data structure in the containing virtual memory, and, since it contains only names and quantities known there, it can be freely operated upon by the process of that containing virtual memory. Figure 8 illustrates an example of a hierarchy of virtual memories.

Within a virtual memory, programs identify each memory location by a virtual memory address, which consists of a segment name, and a position within the segment, expressed as an offset from the beginning of that segment. The hardware processor translates the virtual address into a physical address with the aid of a small array of registers called the display, and a single integer register named level. These registers are internal to the processor and are not accessible to any programs, not even the most central part of an operating system. Instead they are automatically updated by the hardware as a side effect of obeying the special instructions which are provided for such purposes as starting or returning from a process.
Figure 8.

A Hierarchy of Virtual Memories

32.
The register level contains at all times the degree of nesting of the process (i.e., its virtual memory) currently being executed. The hardware memory is regarded as level zero. The elements of display contain the physical addresses of the segment tables of the hierarchy of processes containing the current one — see figure 9. (This display is not to be confused with the block/procedure display mentioned in Section 4.1.)

In any practical implementation, the hardware address algorithm must include checks against violating the segment length, exceeding the maximum permitted segment name of a virtual memory, accessing a segment which is not defined or not present, or attempting operations not permitted by the type and control information for that segment. These checks must be applied at each level when necessary. For this purpose, each processor has, in addition to its display registers, another array of integer registers named virtual memory size. Each entry in this array contains the maximum possible number of segments in the virtual memory described by the segment table named in the corresponding display register.

In principle the hardware address algorithm involves n traversals of a loop, each involving a reference to a display and a virtual memory size register and a segment table, for a level n virtual machine to access one of its memory locations. In practice the processor would maintain a small fast look-aside memory containing the physical locations and characteristics of the most recently used segments of the current virtual memory. This should make it possible for the processor to operate at nearly the same speed at all levels in a virtual memory hierarchy.

Fuller details of the basic addressing algorithm are given in Lauer and Wyeth [18], which also describes how processes can be created and deleted, a parameter mechanism for communication between a process and its descendants, the handling of input/output devices and the interrupts that they generate, and a paging scheme.
In summary, the recursive virtual machine system provides a protection mechanism based on a recursive context-dependent addressing scheme. Each process operates in a virtual memory defined by its immediate supervisor; and any process can create descendant processes within itself, freely allocate its own resources to those processes, and freely control any asynchronous devices allocated to it. No matter how devious a process might be, it cannot exceed the bounds placed on it by its supervisor. It can offer any bit pattern whatsoever as a potential address of a resource, either directly or indirectly, but only those bit patterns which are valid in the context of its virtual address space will be honoured. It can execute any operation in the machine order code with the same effect in its virtual memory as would have occurred had it been operating on a bare hardware system with the same configuration.

The mechanism ensures that no process in the system need be aware of the fine structure of its descendants nor whether or not they have descendants. This is because every request made to a process is framed in terms of its own virtual address space by the parameter passing mechanism, no matter how deep in the hierarchy it originated. A process need only be concerned with allocating its resources to its immediate descendants, for they can allocate them further.

These kinds of protection, namely the hierarchical protection of objects with nested lifetimes, should be distinguished from another very important kind of protection. The latter is the kind used to protect objects which could potentially be in the scope of a large number of processes and where a finer degree of protection than one that is keyed to the lifetimes of objects and processes is required — this is where the notion of capability is most valuable, and most troublesome.

Capabilities are essentially non-forgeable pointers and, we believe, would fit quite naturally into our hierarchical system in much the same way that references to records have been fitted into decent block-structured languages, starting with Algol W [27].
Within a hierarchical structure the natural point of declaration of a capability (or any other object), which is to be common to several parts of the structure, is within the scope which encloses all such parts of the structure. Thus in different parts of the structure different sets of capabilities could be provided, but no capability could be used as a means of allowing access to an object whose scope exceeded that of the virtual machine in which the particular capability class was introduced. By this means areas of comparative lack of structuring are circumscribed, and fitted into such an overall hierarchical structure as can be provided — an approach which we much prefer to the alternative of making no attempt to retain any hierarchical structure [8]. However development of this particular aspect of the system architecture can we feel be postponed to allow more pressing problems of error detection and recovery to take precedence.

Much work remains to be done on the idea of a recursive virtual machine and on its incorporation together with the other plans we have into a single coherent computer architecture. The task of producing an actual implementation of an initial version of the emulator, as required for the experimental system outlined in Section 7, is proving a very useful stimulus to this work.

6. System Error Recovery

Our first attempt at tackling the problem of error recovery involved an analysis of various existing techniques for checkpoint, back-up and restart, as applied both to complete operating systems, and to subsystems such as spooling systems, general-purpose filing systems and application-oriented data base systems. Various programming language facilities, mainly developed in artificial intelligence contexts, for implementing backtracking were also examined as part of the study. (The survey is documented in SRM/37.) One common aspect to the systems examined was the comparative lack of structuring. It seemed clear to us that the facilities for checkpointing, etc., a hierarchically structured system could with profit differ considerably from those analysed. This started an at times confused debate on system structure and its relation to program structure, error responses across levels of a system (a subject that Parnas has also discussed [22]),
and the notion of forward and reverse progress at a given level of a system. (This debate gave rise to SRM/41, 43, 44 and 46.) Out of this debate eventually came the idea of 'recovery blocks', which now form the basis of our plans for structuring the error recovery provisions of a system, and which spurred the development of what we have termed the 'recursive cache'.

6.1 Recovery Blocks

A well-structured program is constructed from identifiable operations, many of which are themselves constructed from further smaller operations. Our scheme is based on the selection of a set of these operations upon which to base the recovery operations. These will be referred to as Recovery Blocks. Recovery blocks can be nested like Algol blocks, but a recovery block must have a more complex internal structure than an Algol block so as to support error recovery. Each recovery block contains a primary block, an acceptance test, and zero or more alternate blocks.

The primary block corresponds exactly to the block of the Algol-like program, and is entered to perform the desired operation.

The acceptance test is executed on exit from the primary block to confirm that the primary block has performed correctly.

If the primary block is detected to be in error, the alternate block is entered and is required to perform the desired operation in a different way or to perform some alternative action acceptable to the program as a whole. The acceptance test is then repeated.

The diagram of a recovery block structure given in figure 10 is intended to be illustrative, rather than syntactically representative. The double vertical lines define the scopes of recovery blocks, while the

37,
single vertical lines define the scopes of primary and alternate blocks. Figure 11 shows how the primary and alternate blocks can contain, nested within themselves, further recovery blocks.

The acceptance test is a section of program which is invoked on exit from a primary or alternate block. Its function is to provide a binary decision as to whether the operation required of the recovery block has been performed acceptably to the program which encloses or invokes the recovery block and it is essentially a form of assertion [12]. (The run-time checks included in 'safe algorithms', as discussed in Section 4, could be given in the form of acceptance tests.) There is no requirement that the test be, in any formal sense, a check on the absolute 'correctness' of the operation – it is for the designer to decide on the appropriate level of rigour of the test. For each recovery block there is a single acceptance test, invoked on exit from the primary and also on exit from the alternate should the alternate be required. Thus in figure 11, the acceptance test BT is invoked on completion of primary block BP, so as to check the acceptability of the results of BP.

The acceptance test does not lie within the primary block, but is syntactically a part of the next enclosing block. Thus the acceptance test cannot access the local variables of the primary or alternate blocks; indeed there is no reason why the local declarations of these blocks should be the same. The function of the acceptance test is to ensure that the operation performed by the recovery block is to the satisfaction of the program which invoked it. The acceptance test is therefore performed by reference to the variables accessible to that program, rather than to local variables which can have no effect or significance after exit.

The acceptance test can reference any variable within the current scope immediately enclosing the recovery block. The primary block may modify some of these variables. For convenience and increased rigour, the acceptance test is enabled to access such variables either for their modified value or for their original unmodified value.
Figure 10

A diagrammatic representation of a recovery block structure.
A more complex recovery block structure, showing nested recovery blocks.
There are four possible causes for the rejection of a primary or alternate block. These are:

a) explicit rejection by the acceptance test,
b) failure to terminate, detected by a timeout,
c) detection of an error within the block by one of the implicit error detection mechanisms (e.g. protection violation, divide by zero, etc).
d) explicit or implicit rejection within an inner recovery block which exhausts the recovery capability at that level.

If the primary of a recovery block is rejected then an alternate is automatically invoked. When this alternate terminates, its results are submitted to the same acceptance test, and should the acceptance test be satisfied, the program proceeds using the results generated by the alternate. If the acceptance test is again not satisfied then a further alternate is tried. Thus, in figure 11, if the results of primary block BP are rejected by acceptance test BT, then alternate block BQ is invoked. If the results from BQ are still unacceptable to BT, then BR must be invoked.

An alternate block might perform the desired operation in a different way, presumably less efficiently, but perhaps by a simpler and less error-prone algorithm. However the more likely case is for an alternate block to be designed to provide an operation which, though less desirable, is still acceptable to the program as a whole. Thus the recovery block structure might be used merely as a means of structuring the rather ad hoc error recovery facilities that many existing systems incorporate.

Should all the alternate blocks have been obeyed, and all have failed to satisfy the acceptance test, then the entire recovery block must be regarded as having failed. This causes rejection of the enclosing block which invoked the recovery block, and the alternate to that enclosing block must be attempted instead. Thus, in figure 11, if the results from alternate block BR are still unacceptable to acceptance test BT, then recovery block B as a whole, and therefore primary block AP, must be regarded as having failed. The next program to be attempted is alternate block AQ.
If an error occurs during the execution of the program of an acceptance test, this is regarded as an error occurring within the next enclosing recovery block. The alternate invoked is therefore that for the enclosing block rather than that for the block whose acceptability is being tested. Thus if an error is detected whilst executing the program of acceptance test BT, the alternate block AQ must be entered.

6.2 The Recursive Cache

The recovery block scheme presupposes an underlying mechanism, whose overheads are tolerable, such that

(1) Primary blocks can be written in exactly the same manner as in a conventional system with no restrictions on the operations they use, the procedures they call, or the global variables they modify.

(2) There is no requirement for the explicit preservation of restart information.

(3) Alternate blocks can be written with the same freedom as primary blocks. Their design is not affected by the prior unsuccessful execution of the primary block.

(4) On rejection of a primary block by its acceptance test, the state of the program must be restored so that the alternate block can be presented with exactly the same environment as was its primary when it was entered.

The mechanism we have devised for these purposes was developed under the name 'recursive cache', a term which was introduced at an early stage in its development and which is retained mainly for reasons of continuity. The recursive cache mechanism is at present under consideration by the National Research Development Corporation, acting as agents of the Science Research Council, in order to decide whether patent protection should be sought. We will therefore not describe the actual mechanism here, but merely discuss one basic property that it possesses. (Mechanisations of the recursive cache concept are documented in SRM/57 and 59.)

42.
The requirement that an alternate block be presented with exactly the same environment as was its primary could be mechanised by appropriate core image check points, (or even by retaining enough information to allow programs to be executed backwards, instruction by instruction[1]), but the associated overheads would defeat the purpose of the concept. The heavy overhead of recording a conventional check point on entry to each recovery block is caused by the recording of the entire context of the recovery block, a substantial quantity of information. But in many cases only a few of the non-local variables will be modified by the recovery block.

The recursive cache mechanism ensures that just such non-local variables are saved, by recognising when an assignment to a non-local variable is the first one to have occurred within a given primary or alternate block.

Thus no unnecessary information is saved. The amount of extra space needed for saved information will of course depend on the design of each program and its recovery block structure. Experiments with our initial system implementation will enable us to assess the space requirements of the cache; however we have already been encouraged by the results of some brief experiments involving the interpretive execution of a number of Algol W programs. Even regarding each block as a recovery block it was found that the amount of space that would be needed for the cache was in every case considerably smaller than that needed for the ordinary data of the program.

6.3 Further work on error recovery

The recovery block scheme and the associated recursive cache mechanism as discussed above concern single sequential processes in isolation which generate their results entirely by assignment to storage locations. The cache mechanism automates the restarting of such programs following the detection of an error simply by doing the required assignment reversals.
However, many programming operations generate results other than by assignments, or need to preserve some results even when the processing of the operation is abandoned and an alternate is attempted. Typical examples are operations which involve file access and input-output interfaces, accounting routines, diagnostic traces, and interactive user interfaces. It is characteristic of such programs that error recovery is more complex than automatic reversal of assignments, and an opportunity must be provided for the program designer to specify the appropriate recovery action.

We have extended the recursive cache mechanism so as to allow operations requiring special recovery action to be structured into special procedures, known as recoverable procedures. (These were introduced in SRM/58.) A recoverable procedure is not itself a recovery block with an acceptance test and alternates, but its body should consist of one or more recovery blocks. Such a procedure may declare own variables, whose values are not automatically reset by the recursive cache mechanism in the event of error, but can be restored by program within the recoverable procedure. It may be appropriate to allow several recoverable procedures to be associated and to share a set of own variables. Such a structure would follow naturally from the classes of Simula [4] or the type mechanism that has been developed at Newcastle by Campbell and Habermann [6].

Just as the recovery of simple variables involves the saving of a value on first assignment within a recovery block, multiple further assignments without special action, and the restoration of the prior value in the event of error, so the recoverable procedure must provide three entry points, the save, the normal and the reverse entry points. The save entry point preserves such recovery information as the programmer specifies before performing the required action, and the recursive cache mechanism will enter here the first time that the procedure is called within the recovery block. Subsequent calls of the procedure within the recovery block will enter at the normal entry point. (The assumption is that the information saved on the occasion of the first call of the procedure suffices to allow the system to be reinstated to the
satisfaction of the programmer, even if there are many further calls). It is important to note that the program which calls the recoverable procedure is not aware of the save entry point, which is invoked automatically by the cache mechanism. The reverse entry point is also invoked automatically by the cache mechanism, in the event of an error in the block that invoked the procedure, so as to provide an opportunity to restore the recoverable procedure in accordance with the information saved on entry through the save entry point.

An example of a recoverable procedure might be a file access interface which maintains own variables indicating the current position of each file. The save operation would simply record the values of these variables on entry to each recovery block, while the reverse operation would use the saved values to reposition the files.

We have also made some progress in the development of techniques for restarting a set of interacting processes after an error has been detected in one of them. In fact the ordinary recursive cache mechanism suffices to cope with a number of simple situations of parallel processing. However work is still in progress on this problem, and it is a little premature to attempt a detailed survey of our results to date. (The work is documented in SRM/60, 68 and 69.) The problem of detecting errors of synchronisation amongst a set of interacting processes has not been tackled directly. However we have hopes that the 'path expression' technique of expressing the synchronisation requirements of processes using operations that involve shared objects that has been developed by Campbell and Habermann [6] will help us with this problem.

Clearly, the recursive cache scheme is only in the early stages of development, and its value has yet to be demonstrated. However, we feel it is a very promising technique for providing a well-structured context for explicitly considered error recovery operations. These can be constructed at various levels within the program so that if a local recovery operation should fail a more global recovery action can be substituted. The structure makes explicit the nature of the checks that are being applied in a system, and also the nature of the action to be taken in the event of an error, the number of such actions being
strictly limited. We intend to take advantage of the explicit structure by making provision for the automatic exercising of all alternate blocks.

One final aspect of the recursive cache scheme is worth mentioning. The aim has been that it should not impose any unnecessary constraint on the programming or the architecture of the computer, and the mechanism has been designed with this in mind.

Each primary and alternate block is programmed in exactly the same manner as a conventional program, and could be written in any desired programming language and with any desired programming style or methodology. Extensions to an existing language would be required only to define the recovery block structure, and to permit access to prior values of variables during acceptance tests.

It is natural to equate the recovery blocks with a subset of the blocks of an Algol-like program, but it is not necessary that all the Algol blocks be recovery blocks, or even that the programming language provide an Algol-like block structure. The only requirements are that the recovery blocks should be explicitly defined, that they should be dynamically nested, and that entry to and exit from recovery blocks should be explicit. Clearly, well-structured programming techniques are to be encouraged, but the recursive cache concept does not depend on them. It is as applicable to programming in assembler on a S/360 as to Algol on a B6700. Thus the recursive cache concept is substantially orthogonal to the programming language and methodology.

7. The Initial Experimental System

Paper design studies have been undertaken of two fairly simple operating systems – a batch-processing system incorporating input/output spooling, and a multi-access system. The purpose was to generate ideas on the susceptibility of operating system designs to unreliability, to gain experience of the SUE system language and of the problems of analyzing and documenting system design decisions, and to provide feedback to the designers of the emulated system's architecture. (The work is reported in parts 1 and 2 of SRM/40 and SRM/47.) However we are now undertaking the detailed implementation of a first experimental system, whose major components will be an
Emulator, and a 'Fault Tolerant' Operating System. The standard facilities provided by the operating system will in fact be more modest than those envisaged in the earlier design studies, at least initially - instead it will incorporate special provisions for coping with errors in user programs.

The experiment is intended to serve three purposes:

(i) To ensure that our novel architectural features have been specified and mechanised sufficiently to form an adequate basis for future experiments of greater complexity. In particular the recursive virtual machine and recursive cache concepts are to be demonstrated in operation.

(ii) To investigate the use of the recursive cache concept for error recovery. Miniature but realistic user programs are to be written containing acceptance tests and alternate blocks. These programs will perform input-output operations, but will not communicate with each other. (An example of the kind of program that is to be produced is a simple sort-merge.)

(iii) To investigate the operating system problems involved in providing error recovery aided by the recursive cache, for multiple non-interacting user processes. In particular the problems of recovering the input-output system when a user process must be restarted will be investigated.

In order to concentrate on these aims, we are avoiding implementing our own self-contained operating system, which is what would be needed if the Emulator was coded to run on a bare PDP-11. Rather we are making use of the manufacturer-supplied Disk Operating System, and will run the Emulator under its control. The Fault-Tolerant Operating System will appear to be running entirely on the emulated machine, but many of its facilities, and in particular those concerned with actual input/output devices, will be obtained by indirect calls on the relevant DOS services. Figure 12 is intended to illustrate the overall structure of the initial experimental system.
Because of the hardware installation schedule and the superior facilities available to us with the Michigan Terminal System (MTS) on the 360/64, all program preparation will be performed on the Model 67 rather than the PDP-11's. The Emulator is being programmed in the ordinary SUE system language, and will be compiled to PDP-11 code on the Model 67 using the standard SUE compiler. The operating system and the application programs will be written in a dialect of SUE, with the latter making use of the additional language features which provide the recovery block structure, and will be compiled to Emulator code by means of extensive modifications to one of the standard SUE compilers.

Initially at least the Model 67 will be used even to the extent of preparing core images of the Emulator, the Fault-Tolerant Operating System and a set of fully-loaded application programs, for transmission direct to a PDP-11 processor. On the PDP-11, the Fault-Tolerant Operating System will provide, in addition to facilities concerning error detection and recovery, just a multi-programming scheduler (for a fixed number of user processes, operating in fixed size memory partitions), support for an operator's console, and interfaces to the DOS device routines. Subsequently the Fault-Tolerant Operating System may be

![Diagram of the first experimental system.](image-url)
extended - the approach is to elaborate a system which already embodies appropriate reliability techniques, rather than to attempt to add reliability to an already complex system.

8. Project Organisation and Schedule

The size and character of our project is such that an informal organisation in which ideas are freely exchanged and developed suffices. Nevertheless a certain degree of formality is required for the assessment of progress, the taking of decisions, and the apportionment of effort. We are attempting to achieve the necessary degree of co-ordination by progress meetings, which are held irregularly but not infrequently, whose discussions are minuted and distributed to all associated with the project. Our Systems Research Memoranda form a medium through which project members present their thoughts and provide a record of the technical development of the project. Occasional literature surveys also appear in this memorandum series and form the basis of the catalogue of relevant literature that the project maintains.

The delays that occurred in the early stages of the project were considerable, and mean that the original schedule has had to be reassessed. The project starting date was chosen on the assumption that there would be a sufficient interval following acceptance of the project proposal for us to advertise for and select staff prior to the chosen starting date. The actual interval was in fact so brief that staffing of the project did not start until February 1972, and was not complete until August 1972. Of course even more significant have been the delays that have arisen in the procurement of computer hardware, which were detailed in Section 3.

As described earlier, the original schedule called for the completion of a demonstration system which would incorporate multiple instances of each different type of major hardware component, and have the ability to cope satisfactorily with simulated hardware and software errors injected into it by a monitoring computer. Such a system is considerably more ambitious than the planned first experimental system which was outlined in Section 7. We feel that its design should await
the outcome of the first experiments, which will provide us with the first reasonably adequate test of the potential of the ideas that we have had so far on program structuring and computer architecture. Thus a detailed revision of the project schedule will be undertaken towards the end of the initial set of experiments, probably during the summer of 1974. By this time we should have a clearer idea of how much of the first implementation of the Emulator and the Fault-Tolerant Operating System should be retained for use in the full scale implementation. However it is already clear that we will not be able to recoup very much of the delay that the project has been subjected to, and will wish to have the duration of the project extended significantly.

9. Conclusions

The difficulties of evaluating a computing system design, particularly of any degree of novelty, are immense. Indeed, there have been systems which have generally been regarded as failures, despite the technical excellence of much of their design. (The reverse is just as true!) Even now we still lack adequate criteria for assessing the performance, let alone reliability, of systems which differ in other than entirely trivial functional aspects. Moreover it is usually very difficult indeed to isolate the effects on the overall behaviour of a given system of each of the many different design decisions that went into its making.

Our hope is that we can take proper advantage of our freedom from certain external pressures, such as a need to provide a service to actual users, or to maintain compatibility with some earlier system, and gain increased insight into the problems of designing a system that has an extensive ability to detect and recover from errors. For example, we are not committed to retaining any part of the early versions of the system design and implementation. Moreover we are deliberately concentrating on aspects of the system concerned most directly with reliability, at the expense of many other problems which would require solutions for a comprehensive and balanced system design. In particular we will content ourselves with a system whose actual performance merely suffices to enable us to estimate what the reliability/performance trade-offs should be in a fully realistic implementation.

50.
Clearly we are not presuming that anyone else will wish to accept in its entirety the system design that we evolve. Therefore we are trying to avoid any undue interdependence between the various aspects of our design. Thus, for example, although we believe that the emulator language, the recursive virtual machine, and the recursive cache all complement each other, we have deliberately kept their basic designs as separate as practicable. Hopefully this will aid us in the (undoubtedly simplistic and largely qualitative) evaluation exercises that we plan to conduct with the experimental system implementations.

At the present stage of the project many problems still remain to be tackled, or even properly identified. Even the progress that we believe we have made has yet to be, so to speak, "validated". However, we hope that the necessarily brief details given in this report of the work done so far has demonstrated the potential of our approach to fault-tolerant computing, and in particular the appropriateness of our concentration on the development of recursively defined hierarchical structures, at both the programming language and machine language levels of computer systems.

10. Acknowledgements

A glance at the list of project memoranda given in Appendix 1 will demonstrate that the work surveyed in this report is the result of the collaborative efforts of a sizeable group of project personnel, academic staff, and post-graduate students – indeed much of the text of this report is extracted from their writings. It would be very difficult to assign individual credit and responsibility accurately for the various aspects of the project, which originated in discussions between Jim Eve and the author, but several contributions merit mention.

The work on programming languages and on the use of correctness proofs as a guide to the use of program redundancy was principally carried out by Tom Anderson. The Emulator has been the responsibility
of Robert Simpson, with assistance from Hugh Lauer and Ron Kerr. Hugh Lauer, aided by Dick Snow and David Wyeth, is the source of the recursive virtual machine concept. The work on operating systems has been carried out mainly by Tony Mascall and Ron Kerr, with assistance from Nico Habermann, who was the holder of an IBM-sponsored visiting professorship at the laboratory. The recursive cache concept was developed by Jim Horning, Mike Melliar-Smith, Hugh Lauer and the author. The initial idea of the Magnabus arose from discussions between Gordon Bell, of Carnegie-Mellon University, Hugh Lauer and the author, and has since been developed considerably by Keith Heron. The author's thanks are due to those named above, and to the various other members of, or visitors to, the Computing Laboratory at Newcastle who have done so much to make the project not only exciting, but also very enjoyable.

As has been mentioned earlier, the project has received valuable technical assistance from several other institutions, in particular the Computing Systems Research Group at the University of Toronto, and the Edinburgh Regional Computing Centre, to whom our thanks.

Finally, of course, it is a pleasure to acknowledge the assistance and support that the project has received from the Science Research Council.

11. References


53.


APPENDIX 1

System Reliability Memoranda


12. (Not issued).


32. On a Recursive, Virtual Machine Architecture.

33. A Run-Time Mechanism for Handling External Variables and Protection.

34. Reports and Papers relating to Highly Reliable Computing Systems (Second List).

35. Summary of Critical Comments on the Programming Language PASCAL.


37. Checkpoint, backup and restart in a "reliable" system.

38. An Approach to the Problem of Writing Highly Reliable File System Software.

39. The project SUE System Language in Relation to Habermann's Comments on PASCAL.

40. Designing and Documenting Operating Systems (Part I)

41. A Taxonomy of Error Responses.
59. The Recursive Cache.


60. Recovery from Error among a set of communicating processes.


61. Further considerations for the First Reliable Computing Experiments.


63. Determinable Termination.


64. Visit to E.R.C.C. on Wednesday 10th October.


67. Detailed Description of the Computational Facilities of the Emulator.


68. A Structured Message-Passing System.


P.M. Melliar-Smith, November 1973.


K. Heron, December 1973.

71. Estimates for the size of the Recursive Cache.

APPENDIX 2

Initial Configuration of Project Computing Facilities

2 PDP-11/45's (includes two teletypes)
4 MM11-S 8K Modules, 850 ns core
1 LP11 Line-printer
2 TU57 Dual DEC-tape transports
2 TC11 DEC-tape controllers
2 RKO5 Disc drives
1 RK11 Disc drive controller
2 MR11-DB Bootstrap loaders
2 DL11 Asynchronous Serial Line Interfaces
1 DP11-DA Synchronous interface
1 Elliott Paper tape Reader

Cabinets, power supplies, cables, etc.