AN ERROR RECOVERY SCHEME FOR CONCURRENT PROCESSES

by

Lindsay Forsyth Marshall

Ph.D. Thesis

Computing Laboratory

University of Newcastle upon Tyne

August 1980

AN ERROR RECOVERY SCHEME FOR CONCURRENT PROCESSES

Lindsay Forsyth Marshall

Ph.D. Thesis

University of Newcastle upon Tyne, August 1980

Abstract

With the more widespread use of multi-processors and distributed computing systems, programmers need a simple, reliable interface to them. This thesis describes language constructs, and mechanisms for their support, that can be used in the implementation of fault-tolerant concurrent processes. The basic language structure is the Atomic Action, supported by a modified recovery cache mechanism. This combines the collection of recovery data with the locking of resources and allows recovery blocks to be integrated with Atomic Actions. Synchronisation between actions is discussed, as well as a means of detecting and breaking deadlocks, based on the use of a "blocking graph".

Reliable communication and cooperation between actions is considered, and several constructs are investigated. The limitations of Shared Atomic Actions are identified, and, further, the use of a form of reliable "secretary" is shown to lead to unneccessary recovery activity. These problems are resolved by structures based on a classification of resources by the way they are used in programs.

Also contained in the thesis are descriptions of trial implementations of some of the mechanisms described, and a discussion of existing concurrent programming techniques.

Aknowledgements

I would like to thank my supervisor Professor B. Randell for his help and encouragement. His criticisms of my work and his suggestions for sources of information have been invaluable in shaping this thesis.

My thanks are also due to the research staff at the Computing Laboratory at the University of Newcastle upon Tyne, especially Drs. T. Anderson, S.K. Shrivastava and P. Treleaven, with whom I had many fruitful discussions.

Finally, I must thank Miss C.J. Smith who typed this thesis, and the British Ship Research Association who have kindly allowed me to use their facilities for its production.

The work described in this thesis was supported by a grant from the Science Research Council of Great Britain.

Contents

1.0 Introduction

1.1 Objectives

With the rising cost of software development and the increasing use of distributed processing (especially the advent of cheap multi-microprocessors) there is considerable pressure to make concurrent programming techniques more reliable and more accessible to the average computer user. The problems involved in concurrent programming are well understood, but the design of language interfaces and the mechanisms to support them has lagged behind the theoretical work in this field. The interface usually provided to the programmer consists of a set of library procedures, calls on which allow him to spawn tasks and share the use of resources with these tasks. An example of this kind of interface is that provided on the UNIX system which is described in (Rit 78). However, as the structured programming techniques advocated by Dahl, Dijkstra and Hoare (Dah 72) become more widely used there is a need for parallel programming facilities to become a more integral part of programming languages (for example see the "STEELMAN" language requirements (DoD 78)). The result of this has been the development of languages like concurrent PASCAL (BrH 75) and MODULA (Wir 77), both specifically designed to provide a simple, structured means of specifying concurrent programs. Unfortunately, though the process structuring primitives provided by these languages are excellent, their facilities for controlling the use of shared resources and for error detection and recovery are severely limited. It is the aim of this thesis to show how to support a simple language interface, permitting controlled access to all types of shared resources and providing comprehensive error recovery facilities.

Existing work in this field has tended to fall into one of two areas - either concerned with the fairly simple interactions between processes at the lowest levels of operating systems or with the highly complex interactions occuring in data base systems. The former area has produced several language structures for resource control some of which will be discussed in chapter two, but in general the question of recovery from errors when shared resources are being used has not been addressed. Almost the reverse is true of the latter area where the development of language interfaces has been secondary to that of mechanisms to ensure the integrity of data bases and of sophisticated locking schemes to make access to data more efficient. There is obviously a need, therefore to bring together ideas from the two fields so that a more general purpose set of facilities can be built up. This thesis attempts to show one way of acheiving this synthesis by combining the main features of these two areas into a system that provides the user with the ability to write <u>FAULT-TOLERANT</u> software.

A fault-tolerant program is one which will produce acceptable results even when it has passed through erroneous states during its execution. It will normally contain code that is designed to cope with these incorrect states by attempting to ensure that errors are not propagated to later stages of execution. Where programs may interact through the use of shared resources or by direct communication it is often very difficult for the programmer to know the extent of the damage caused by a fault and in these cases the underlying system must play a large part in the collection and maintenance of recovery information. The main part of the work below is a description of a mechanism , based on the Recovery Cache (Ran 75) (see section 3.4) , which allows this to be done very simply and which gives the user control over the way in which error recovery takes place. However before embarking on this description we must define some fundamental concepts which are used throughout this thesis.

1.2 Definition of basic concepts

1.2.1 Atomicity -

An activity will be described as ATOMIC if the operation that it performs appears to be indivisible and does not interfere with any other concurrently executing activity. Dijkstra (Dij 72) and Brinch Hansen (BrH 75) have categorised such operations as being MUTUALLY EXCLUSIVE, however we shall use this term to describe only the subset of atomic operations which do not allow the possibility of several activities performing them concurrently. An example of an operation which is not mutually exclusive is the reading of a variable without modifying its value, which can be carried out safely by any number of processes. Brinch Hansen has said that without mutually exclusive operations discussion of concurrent computation would be meaningless, and Lipton (Lip 75) has shown the desirability of atomic operations in that their presence simplifies the task of proving parallel programs. Eswaran et al (Esw 76) use the notion of atomicity to define a TRANSACTION which is a sequence of atomic operations grouped together forming a unit of consistency. That is, the system state will be consistent before and after the transaction has been executed. For the purposes of this thesis we shall assume that a PROCESS consists of a sequence of one or more, possibly nested, transactions. The term process will also be used in cases where transactions are nested and we need to refer to the innermost transaction *and* all its ancestors. PROGRAM will be used interchangeably with process, except where there is no concurrency in which case it will be prefixed by the qualifier SEQUENTIAL.

A formal model of atomicity has been described by Best and Randell (Bes 78, Bes 79), which, though still under development, is being used as a basis for studies of parallelism and error recovery. Gray et al (Gra 76) and Davies (Dav 79) also make considerable use of the concept of atomicity in their work in these fields.

1.2.2 Resources -

Throughout this thesis we shall use the term RESOURCE to describe *any* object which a programmer can use through the medium of his programming language. The most usual examples of such objects are the variables held in a computer's memory, but magnetic tape drives, graph plotters and terminals are also resources. In describing the operations that a programmer performs on resources we shall use the categories - READ and WRITE. A read operation is one which examines the state of a resource but does not modify it (for example determining the position of a disc head), whilst a write operation always modifies the state. Thus rewinding a magnetic tape would be regarded as a write operation as it affects the state of the drive by changing the position of the tape. Many of the examples in the text will be couched in terms of variables (storage resources), but by using this classification of operation they can be generalised to all types of resources.

1.2.3 Commitment -

The state of a resource is said to have been COMMITTED when the transaction which put it in that state confirms that the state is correct and relinquishes the ability to perform any recovery involving that resource. Where transactions are nested, commitment is also nested, the ability to recover being maintained until the outermost transaction commits the resource's state. Before this FINAL COMMITMENT the state of a resource is not guaranteed correct and may be subject to change at any time, especially in the event of an error. If other processes make use of the uncommitted results of a program the support system must record this fact to enable any subsequent changes to the resource to be correctly propagated throughout the system. This is especially important where the results are found to be in error and corrective action must be taken.

There are two ways in which this can be done. Firstly, processes which use uncommitted data can be prevented from committing their results until the data in question has itself been committed. We shall describe these processes and the originator of the uncommitted data as being <u>COUPLED</u>. Secondly processes may use uncommitted data but some means of <u>COMPENSATION</u> must be provided to correct any errors that may have arisen. For example, when some goods have been sent out from a depot by mistake, by issuing a recall order a compensation mechanism has been invoked to recover from the error. Processes which interact in this way will be described as <u>UNCOUPLED</u>.

<u>1.2.4</u> <u>Cooperation</u> -

Shrivastava and Banatre (Shr 78a) have identified three types of interaction between activities - <u>INTERFERENCE</u>, <u>COOPERATION</u> and <u>COMPETITION</u>. Interference is always erroneous and occurs when shared resources are modified simultaneously and the use of atomic operations is designed to eliminate such interactions. Cooperation occurs when activities deliberately pass information between each other, and competition (or contention) arises when activities simply wish to share resources but do not wish to pass information. However in this thesis we shall look at the interaction between activities in a slightly different way which will require another terminology. To this end we shall introduce the idea of <u>COOPERATIVENESS</u>, it will, however, be easier to first define its opposite - <u>UNCOOPERATIVENESS</u>.

An activity is said to be uncooperative if it does not communicate directly with other activities and if it does not release the shared resources it has acquired during its existence until it terminates. That is, the activity appears atomic to the rest of the system. On the other hand a cooperative activity is one which may communicate with other activities directly and which may also release some resources during the course of its execution. Such an activity would not seem to be atomic but we shall show that by restricting communication and the type of resources that can be released, a form of atomicity can be attained.

There is also another type of interaction and we shall call activities which uses it CLOSELY-COOPERATIVE. Here a group of activities appear to be one simple uncooperative activity to the rest of the system, but are able to communicate with each other and release resources so that other members of the group can acquire them.

1.3 Summary of thesis

The major part of this thesis consists of a description of a user interface for the implementation of parallel processes which share the use of resources, and the mechanisms needed to support it. The interface provides the programmer with facilities for incorporating fault-tolerance into his programs and allows him to control the way in which the failure of one process affects other processes which have interacted with it. The most important features of this interface have been built into test systems and their implementation is described.

Before this, in chapter two, an overview of existing systems for controlling the use of shared resources is given. Firstly there is a discussion of the kind of facilities that such systems ought to provide and this is followed by a description of some of the techniques that have been developed. Evaluation of these systems is carried out with reference to two areas - the interface they provide to a user, and their reliability in the face of errors.

Chapter three describes a basic user interface for the support of uncooperative processes based on a mechanism which combines the locking of shared resources with the collection of recovery information. A discussion of deadlocks follows and a means of detecting and recovering from them is developed. The problems of synchronisation between processes are then described and a program structure to overcome them is introduced. Implementation of a system of the type described is then discussed with reference to an existing system. Finally the efficiency and cost of such a system are considered.

Chapter four introduces an extension to the basic user interface that allows the specification of closely-cooperative processes. Difficulties that arise when designing a language interface suitable for this are discussed and various solutions are considered. The incorporation of closely-cooperative processes into the prototype system of chapter three is then outlined. An example of the use of closely-cooperative processes is given and in conclusion their overall usefulness is discussed.

Chapter five extends the interface to permit cooperative processes to be supported, developing a new classification of resource types to do this. Examples are given of the use of the facilities that are built up, with special reference to the way in which error recovery is managed. A test system utilising some of the ideas developed in this chapter is described and the problems encountered in its construction are discussed.

Finally in chapter six, a summary and evaluation of the work described are presented, and this is followed by suggestions for directions that future research in this area could take. The sources of the references contained in the text are then given as an appendix.

2.0 Schemes for control and recovery of concurrent activities

2.1 Locking

Whenever two or more activities take place at the same time (or what appears to be the same time), there is always the possibility that they may interfere with each other in such a way that erroneous and inconsistent system states will arise. The classic example of such circumstances is that of concurrent assignments to the same variable, where, if no control is exercised over the interaction, the variable can take on one of several different values, depending on the execution flow of the processes involved. The guaranteed prevention of such events is a necessary condition for the results of an activity to be regarded as correct, and, as the system supporting the activities may not be able to differentiate between correct and incorrect interactions, facilities must be provided to allow the explicit delineation of those areas of an activity where interactions involving a specified shared object are to be prevented. We shall call such a facility a LOCKING SCHEME.

The control a locking scheme provides over activities can be usefully regarded as serving two distinct but related purposes:-

1. Resource allocation - by associating locks with shared resources, activities competing to use them must pass their requests to the manager of the locking scheme, thereby leaving the granting of requests in the hands of the system,

2. Prevention of communication of uncommitted data between activities - that is locking a resource guarantees that operations upon it will appear atomic to those activities not involved in changing its state.

The need for these arises in all fields and it will be worth while to look at the more general aspects of locking, before turning to its application to computer systems.

2.1.1 Classes of locking -

Locking schemes can be classified in several different ways, but we shall look at only two of them here. The first is concerned with when a request for a resource is made, and gives rise to two classes:-

1. STATIC schemes, where resources are locked before the activity wishing to use them starts its operations,

2. DYNAMIC schemes, where requests for resources are issued during the course of an activity, generally immediately before the resources in question are first needed.

The important difference between these classes, apart from the obvious one, is in the way in which system DEADLOCKS are handled - a deadlock being said to occur when two or more activites block one another in such a way that their further progress is prevented. We shall look at this topic in greater depth in section 3.7 below.

The other classification system we shall look at is based on the way in which the "lock" is actually placed on a resource. We shall identify three distinct ways of doing this, but in most cases some combination of the three methods is used to ensure greater security. The three are:-

1. Presence/absence method - when an activity wishes to make use of a shared object it "removes" it from the commonly accessible place allocated for it, to a place private to the activity, thus reserving the object for its own use. There are many examples of this type of lock - a switch (presence or absence of electricity) or in a reference library (presence or absence of a book) - and most have the extra property that the locked object can be relinquished, voluntarily or involuntarily, to some other activity, without it being returned for normal competition. This type of "lending" forms the basis for the preemption schemes used in many computer systems. The major weakness of this scheme is that an object may be "stolen" and so never be returned to be contended for, and this can only be prevented by combining this method with one or both of 2) and 3).

2. Record method - here the status of the object in question is noted down in some way, and this record is consulted to

determine whether a request to lock the object can be granted. This scheme is often used where the object contended for is not amenable to a class 1) solution (for example where it is not present at the place where the request must be made) and has the advantage that the identity of the activity possessing the object can be recorded allowing it to be traced should that become necessary.

3. Token method - in this method an activity is given a token - for example a theatre ticket for a specified seat - which shows that it has locked some object. Note that such things as library tickets, passwords and keys are NOT examples of this kind of token, but are members of the more general category of "CAPABILITIES" (Den 66) which are one step removed from locking schemes, controlling whether or not an activity has the right to perform certain operations, the ability to make requests for resources being one of them.

All these types of locks occur in computer systems, though the presence/absence kind only exist in the hardware, the other two methods often being used to simulate it at the software level.

2.1.2 Granularity of locking -

Whenever structured objects are contended for it is very important to specify how much or how little of the structure must be locked by a user to enable him to operate on one of its parts. We shall call this amount the GRANULARITY OF LOCKING for the structure. This concept was introduced by Gray et al (Gra 76) for use in the discussion of access control for databases, but may be usefully extended to include the types of structure available in programming languages.

The granularity of a structure determines the amount of concurrency that can occur when several activities attempt to use it and so can drastically affect the efficiency with which a system operates. Generally, if the lock unit is too large concurrency will be reduced, if it is too small the probability of requests overlapping and causing deadlock will increase. So to select the lock unit the system designer must evaluate the trade off between the need for concurrency and the frequency of possible deadlock. Of course there is no theoretical reason why all lock units should be the same size as each other and it may be possible to arrange that parts of structures which are always used together are locked as a whole. However, different programs may wish to impose different groupings of parts on a single structure, and where these overlap deadlocks can arise. Gray (Gra 76) has tackled this problem by introducing a new type of lock which is related to the way in which structures are accessed and we shall describe this in the next section.

Another aspect of the choice of granularity is the amount of storage space required to record information about the locks that have been placed by processes. The smaller the lock unit, the greater the number of locks placed so the system will need a larger amount of storage space for its control information. This larger quantity of information could also mean that checking whether a deadlock will arise if a request is granted will be slower, thereby reducing the overall performance of the system.

We can see therefore that the chosen granularity of locking has a significant effect on the efficiency of the system, and we shall discuss this topic further in section 3.9.

2.1.3 Modes of locking -

So far we have made no mention of the access rights that a process acquires when it places a lock on a resource, these being dependent on the MODE of the lock request. The simplest mode is that of the EXCLUSIVE lock, which when granted, gives the requestor read and write (as defined in section 1.2.2) access to a resource, whilst preventing other processes from using it in any way. Of course the request could be made on behalf of a group of processes resulting in them all having SHARED READ/WRITE access to the resource, and this can be a very useful facility, even though it allows the processes in the group to interact with each other in an uncontrolled manner. We shall discuss the application of this in chapter four.

However there are cases where exclusive locks are too restrictive, such as that where several processes all wish only to read a resource. Here all the processes could use the resource concurrently, but an exclusive locking scheme will only permit one process to use it at a time. To overcome this the <u>SHARED</u> <u>READ</u> (SR) lock has been introduced which gives a process read only access to a resource, several processes being permitted to hold such a lock at the same time. Any process wishing to modify the resource must either wait till all the shared read locks on it have been released (barring any that the process may hold itself) or <u>PREEMPT</u> the other processes, forcing them to release their locks, before placing an exclusive lock on the resource. If the process previously held a shared read lock on the resource it is said to have <u>CONVERTED</u> the lock mode from shared read to exclusive. Unfortunately the process of conversion can lead to a deadlock, where two processes both wish to convert a lock on the same resource, and we shall look at this difficulty in section 3.7 below.

These two lock modes are sufficient in systems where only simple, unstructured resources are contended for, but where resources are more complex in structure, as in a data base system, a greater degree of control is needed over access. The reason for this is that exclusive and shared read locks can only be applied to the specific parts of a structure that a process wishes to use. For if locks had to span the hierarchy of the structure one exclusive lock would prevent any other process from accessing it at all, but then a process cannot safely lock a part of a structure without first determining whether any sub-structure contained within the part it wishes to lock has been locked by another process, a very time consuming procedure if a large structure is involved. A solution to this problem has been given by Gray et al (Gra 76) who have introduced the concept of INTENTION mode locks. They have suggested the use of three specific locks:-

1. Intention shared read (ISR) - a process placing such a lock is declaring that it will only place shared read or intention shared read locks on lower parts of the structure, and, if conversion is allowed, will only attempt to convert ISR locks to shared read,

2. Intention exclusive (IX) which has the same properties as ISR but with respect to exclusive access,

3. Shared Read and Intention exclusive (SIX) - which allows a process to have shared read access to a part of a structure, but also lets it lock smaller parts in IX or X mode, thus allowing concurrency with other processes who wish to examine parts it does not wish to alter.

Fig. 2.1a shows a possible partial ordering of all the modes we have discussed, where NL represents the unlocked state and the operator '>' is used to mean 'is more binding than'. The table in fig. 2.1b shows how the system must treat multiple lock requests for the same resource - Y indicates that the request is compatible with the current lock state of the resource and can be granted, N means that this combination of locks is not possible. The intention locks provide an excellent way of controlling access to hierarchical structures, and Gray gives several examples of their use in cases which are usually exceedingly difficult to cater for. However in the rest of this thesis we shall not concern ourselves with the use of intention locks as the extra complexity they introduce would obscure the working of the mechanisms developed. Nonetheless the possibility that they may be required must always be borne in mind.

```
          > IX
X > SIX          > IS > NL
          >  S
```

(a)

		REQUEST MODE					
		NL	IS	IX	S	SIX	X
	NL	Y	Y	Y	Y	Y	Y
	IS	Y	Y	Y	Y	Y	N
RESOURCE	IX	Y	Y	Y	N	N	N
STATE	S	Y	Y	N	Y	N	N
	SIX	Y	Y	N	N	N	N
	X	Y	N	N	N	N	N

(b)

Fig. 2.1

2.1.4 Requirements of programmers -

Before looking at some of the existing programming language facilities for controlling the use of shared resources, let us try and set out the characteristics that such an interface should possess. We shall assume that the user expresses his programs in a structured high-level language such as PASCAL (Wir 71).

The interface has to provide the user with four essential functions:-

1. The ability to issue a request to lock a resource in the required mode,

2. To wait until the request is granted,

3. Allow the resource to be manipulated without restrictions, other than those set by the definition of the resource and the mode in which it has been locked,

4. To release the lock making the resource available to other users.

Two additional functions could also be provided, namely the ability to convert the mode of a lock, and the option not to wait if a resource is not immediately available when a request is made for it. The interface must also be reliable - it should be impossible to use a shared resource without first having locked it and should an error occur after the lock has been placed the support system must be able to release it in a consistent state if the error handling in the failing process does not.

Of course though these facilities must all exist they need not be used explicitly by a programmer for them to be invoked - the system may provide them automatically. This will free the programmer from worrying about resource allocation and release, and allow the system to conceal the fact that some resources are shared though they do not appear to be. Most systems supporting shared resources have already hidden any waiting for resources from the user, but automating acquisition and release presents some problems, though their solutions increase the reliability of the system considerably. Let us look first at the process of acquisition.

If the whole text of a program is available a compiler can determine the points at which resources must be acquired and where they can be safely released. The analysis will be fairly difficult because conditional use of resources means that resouce acquisition must take place at a point in the program which will be executed by all paths or else the code must be repeated where necessary (this may mean that a process must be allowed to request a resource it already possesses). However with the rise of modular programming techniques separate compilation of modules has become the norm, so this approach is not really usable. The only alternative is to perform resource acquisition at run time, that is whenever a shared resource is used the system must check to see if it has been locked by the process, and if not must lock it. This will of course be relatively inefficient, but it guarantees that a resource cannot be used without being locked, and that resources which are used in paths that are not executed will not be locked. One disadvantage is that where more than one lock mode is implemented mode conversion must be supported. For it will often be the case that the first use of a resource will only need a weak lock (reading - shared read) but a later use may require a stronger lock (writing - exclusive), and so the lock must be converted. Section 3.7.2 dicusses the problems that can arise where conversion is provided.

Automatic release of resources is considerably more difficult and either requires analysis of the whole program or for all resources to be released at some predefined point, usually the end of a process's execution. The former though providing far more efficient use of resources is not reliable as commitment of a value can take place before it is certain that the value is correct, so the latter is preferable because of the increase in reliability obtained through using it. The fact that resource release is automatic also means that the system should easily be able to implement the other reliability requirement mentioned above - that resources should be released when an error occurs.

From the above we can see that the simplest interface (probably only from the programmer's point of view) would be one where the programmer treats shared resources exactly as he would unshared ones, leaving the support system to determine when locking is appropriate. The fact that a resource is shared can be determined from its address which will point to a part of the address space common to several processes. In effect the memory management hardware of most computer systems performs this operation anyway so existing techniques should be perfectly adequate to do this. The association between a shared resource and an address need only be carried out once, as most modules could access it as an external object, alowing the shared attribute of the resource to be concealed if necessary. Resource release still presents problems, as holding resources till a process terminates may not implement what the user wishes. However we would suggest that by splitting a process into a sequence of transactions and releasing all locks at the end of each transaction the programmer will be able to implement most algorithms.

2.2 Existing systems for controlling concurrency

Many language interfaces have been devised to allow programmers to control the use of shared resources, however we shall only examine six of them. The constructs chosen are representative of all the other methods that have been developed and are the ones most frequently referred to in the literature. Of course the vast majority of today's computer systems do not provide the programmer with languages that have these constructs embedded in them. The acquisition and release of resources are normally achieved either through the use of a Job Control Language or by procedures calls in a process body. However both these methods can be regarded as equivalent to the use of certain of the constructs described below and we shall indicate this relationship in the appropriate sections.

2.2.1 <u>Semaphores</u> -

The semaphore, described by Dijkstra in (Dij 68a) and developed by him for use in the THE operating system, is widely regarded as a paradigm for concurrency control mechanisms, and the correctness of a new construct is often demonstrated by showing how semaphores may be implemented using it. In fact the concept behind semaphores is so fundamental that almost all the actual implementations of other constructs make use of them (if they are not exactly equivalent).

A semaphore is a special non-negative integer variable which has two operations associated with it called P and V. P is the "acquire" operation of the pair and, where S is a semaphore, is equivalent to the operation:-

```
WAIT UNTIL S > 0;
S := S - 1;
```

The V operation performs the release function and can be represented as :-

```
S := S + 1;
```

and of course both these operations must be atomic to eliminate any interactions between processes executing them at the same time. Dijkstra identifies two types of semaphore, namely the <u>BINARY</u> semaphore, whose value is either zero or one, and the <u>GENERAL</u>

semaphore which can take on larger values, however in (Dij 68a) he shows that only binary semaphores are strictly necessary.

There are two ways in which semaphores can be used to control access to resources. The first is to associate a semaphore with every common resource and require the programmer to perform P's and V's on it to acquire and release the resource. The alternative method is to use one semaphore to control several resources, using single P and V operations to acquire and release all of them at one time. Unfortunately, though the use of these methods gives the programmer considerable flexibility and power in the way in which he builds his program they both possess severe disadvantages:-

1. The fact that a resource has a semaphore associated with it does not guarantee that the programmer will actually perform a P on the semaphore before using the resource,

2. Because P and V are explicit operations there is no structure to link them together, which can not only reduce the modularity and clarity of programs using them, but can also lead to the programmer forgetting the V operation altogether, thereby preventing any more P operations from being successful. Another problem is that the programmer is not prevented from performing two P's without a V between them causing his program to deadlock itself, or from issuing two V's without a P between them, which could cause two processes to acquire the resource at the same time.

3. Semaphores only provide exclusive locking and cannot be used to support shared read or intention locks, for firstly only one process can hold a semaphore at a time and secondly it cannot be guaranteed that the process acquiring the resource will not modify it, this would require some extra protection mechanism,

4. In order to avoid deadlocks the programmer must use special methods such as the "banker's algorithm" (Dij 68a) or must carefully analyse the interactions between the processes in the system before implementation to ensure that they cannot occur.

Despite these disadvantages semaphores are often provided to users - sometimes explicitly, as in the language ALGOL 68 (Wij 69), but more usually through the medium of system procedures. These procedures, for example ones to open and close files, can be used to hide the manipulation of semaphores from the user, but still suffer the disadvantage of being unstructured. Other approaches which have been tried, such as the LOCK and UNLOCK statements of Dennis and Van Horn (Den 66), effectively hide semaphore like mechanisms from the user, but still do not provide structure. This need, and that of simplifying the task of deciding how many semaphores are required to implement a given algorithm, has therefore led to the development of other interfaces.

2.2.2 Path expressions -

Campbell and Habermann (Cam 74) have tackled this problem by introducing what they call PATH EXPRESSIONS. These are a means of specifying the valid sequences of operations that can occur, all those not conforming to the path expression being erroneous - for example the functioning of a semaphore can be represented thus:-

```
PATH P;V END;
```

which indicates that a P operation must always occur before a V. More complex relations can be built up in the notation, and fig. 2.2 shows the specification of a type "file" which can either be opened, read or written several times and closed or can be renamed.

```
TYPE FILE IS
CLASS
   PATH (( OPEN ; ( READ , WRITE )* ; CLOSE ) , RENAME ) END;
   ENTRY PROC READ   ; BEGIN...END; -- reads from the file
   ENTRY PROC WRITE  ; BEGIN...END; -- writes to the file
   ENTRY PROC OPEN   ; BEGIN...END; -- opens the file
   ENTRY PROC CLOSE  ; BEGIN...END; -- closes the file
   ENTRY PROC RENAME ; BEGIN...END; -- renames the file
   BEGIN...END; -- initialisation of a new file
```

Fig. 2.2

An instance of the path is associated with each object declared of type file, and a complete path, that is a sequence of operations such as open, read, write, close, would be executed exclusively by the process which initiated it, any other process having to wait until the path terminated before it could initiate one on the same file. There are several different kinds of paths possible and these are discussed in (Lau 75), which also shows how path expressions are related to Petri nets, (Pet 77) and gives a solution to the "Cigarette smoker's" problem using paths. In order to implement paths, Campbell and Habermann have developed an algorithm which can translate them into equivalent sequences of P and V operations on semaphores which can be placed round the operations used in the path in order to control them. They have thus been able to structure the use of semaphores safely, ensuring that P and V operations are not ommitted, overcoming the first two disadvantages of simple semaphores. However, though this frees the user of correct sequences from concern about interactions between processes, if he erroneously issues an operation out of sequence his process will wait indefinitely, possibly blocking other processes at the same time. Such circumstances can be detected by performing complex analyses of the programs involved, as the use of Petri nets shows, but the techniques required have not been sufficiently developed as to make them accessible to the ordinary user, and this may be one of the reasons why Path expressions have not been incorporated into the design of more recent general purpose languages. Another disadvantage is that, again because of the requirements for mutual exclusion, shared read and intention locks cannot be supported. Also, where separate compilation of program modules is supported, the same path can be represented in different ways (either by error or malicious intent), resulting in hard to detect run-time errors.

We can see therefore that though path expressions provide a clear and compact way of controlling the operations on shared resources, they are as error prone, in their own way, as the explicit use of semaphores. Also the use of regular expressions to describe sequences of operations does not reflect the algorithmic approach that most programmers adopt when implementing a new type. It would seem, therefore, that, whilst path expressions will remain an outstandingly useful theoretical tool, applications programmers will require resource control mechanisms which are directly related to the structure of their programs.

2.2.3 <u>Regions</u> -

In order to make the structure of programs using semaphores clearer Dijkstra (Dij 68a) has introduced the notion of a critical region, which is the term he uses to describe a section of program executed in mutual exclusion from other processes. A critical section is entered by performing a P on a semaphore and terminated when the process performs a V on the same semaphore. Brinch Hansen (BrH 72) has incorporated this concept into a programming language by the use of the REGION statement, which allows statement blocks to be marked as critical with respect to certain specified resources. A compiler can then automatically generate the appropriate operations needed to acquire and release those resources, for example:-

```
REGION P,Q DO
BEGIN
  .
  .
  .
  -- use P and Q
  .
  .
END;
```

This is interpreted as meaning that resources P and Q are to be locked before the compound statement is executed and released immediately it terminates. This provides several advantages to the programmer over the previously described methods:-

1. The notation is simple and relates closely to that used when abstract specifications of programs are written (note that the job control statements that must often be supplied around program bodies are a type of REGION statement),

2. The user is forced to structure the use of shared resources more carefully, preventing such occurrences as overlapping critical regions which are possible with semaphores,

3. The compiler can verify that only those shared resources named in the REGION statement are actually used in the statement body, preventing interference, (though by doing this separate compilation of program modules may be made difficult),

4. Though the structure was originally designed for use with

semaphores, any resource locking scheme could be used to support it, allowing the use of the more sophisticated lock structures described above.

There are, however, disadvantages:-

1. Deadlocks can occur if careful analysis of programs using the statement is not carried out, and, as with the other two schemes we have described, no method of breaking a deadlock that has arisen can be provided,

2. The body of a region statement is NOT atomic and resources can be altered without the users knowledge. Consider this example:-

```
REGION A DO            PROCEDURE P;
BEGIN                  BEGIN
  .                      REGION B DO
  .                      BEGIN
  .                        .
  P;                       .
  .                        .
  .                      END;
  P;                   END;
END;
```

Here the programmer may expect the results of the procedure P to be identical after each of the two calls. However, as the shared resource B is acquired and released every time P is called, B can be used by other processes between the calls on P, introducing the posssibility of incorrect interactions between processes,

3. The statement is inherently unreliable, for if an error occurs during its execution the resource release part of the statement implied by its termination will not be executed. This means that the shared resources it acquired will not be returned to the common pool, and, even if a mechanism could be provided to release them, irreversible alterations could have been made to them, possibly rendering the system state inconsistent.

For these reasons the REGION statement is not as useful as it appears at first, and, like path expressions, it seems to be used more as a theoretical tool than a practical part of a programming language, however this cannot be said of our next example.

2.2.4 Monitors -

The MONITOR was first described by Hoare in (Hoa 74) and has been used very successfully by Brinch Hansen in the language Concurrent Pascal (BrH 75) and his Solo operating system (BrH 76), and also by WIRTH in programming language MODULA (Wir 77). A MONITOR is a special kind of CLASS (Bir 73) in which only one entry can ever be active at a given time, the body of each procedure forming a critical region with respect to the variables declared within the MONITOR. These procedure bodies are atomic because they can only access variables local to the MONITOR or passed in as parameters, and, as only one entry is active at a time, the environment seen by a process cannot change whilst it is executing the entry. Synchronisation and co-ordination of process activity is achieved by the use of queues, a process putting itself on a queue when it wishes to wait for some condition to become true. When a process joins a queue it automatically releases the MONITOR, allowing other processes to use it, and will not be reactivated until another process detects that the condition associated with a queue is true and explicitly activates the waiting process, at the same time relinquishing its lock on the monitor.

However though MONITOR's have been used with great success to solve a great variety of problems they suffer from similar disadvantages to the constructs described above:-

1. The use of queues is not only inelegant but can lead to deadlocks, where their attached conditions are ill formed,

2. The mutual exclusion enforced on MONITOR entries prevents any concurrency, even in cases where it would be perfectly acceptable, and so again shared read and intention locks could not be implemented.

3. Should an error occur in a MONITOR entry, the MONITOR variables could be left in an inconsistent state, causing erroneous information to be propagated when other processes use the MONITOR. Also if the support system cannot identify that the error arose in a MONITOR entry, the MONITOR may not

be released, thus preventing other processes from using it,

4. Because of the scope rules applied to MONITOR variables, algorithms which require the use of nested monitor calls can become very complicated, leading to programmer error.

However despite these disadvantages, MONITOR's have been widely used and the methods needed to solve problems using them are becoming familiar to a widening circle of users. This has led to considerable work being done to improve the reliability of MONITOR's and we shall look at this in a later section.

2.2.5 Atomic Actions -

Ultimately all locking schemes are concerned with ensuring that certain sequences of operations are performed atomically, and to this end Lomet has introduced the notion of an ATOMIC ACTION (Lom 76a) as an explicit program structure. This is simply a device that can be used either like a BEGIN ... END block or a procedure, but which indicates that the sequence of statements enclosed by the structure is to be executed atomically. This has several advantages over the techniques we have described above:-

1. Nested ATOMIC ACTION's do not have the problems of nested REGION's as atomicity is guaranteed from the start of the outermost ATOMIC ACTION until its termination;

2. Once a statement block has been declared as atomic the programmer can use shared resources exactly as he would private ones. This allows programs to be written and tested using local resources, and later converted to use shared resources, simply by altering the declarations of the resources in question;

3. Separate compilation of ATOMIC ACTION's is possible firstly because of their modularity and secondly because the atomicity or otherwise of any enclosing modules, defined elsewhere, is not important;

4. As there is no explicit acquisition and release of resources in a program, the support system has full control over the way in which these operations are carried out, this ensures that all shared resources used in an ATOMIC ACTION are locked before use eliminating the possibility of interfering with other processes,

5. Because an ATOMIC ACTION does not necessarily have to be executed in mutual exclusion the support system can provide the user with shared read and intention locks, allowing the use of data bases and other structured resources to be amalgamated efficiently into the system.

The disadvantage of ATOMIC ACTION's is that the way in which the atomicity they provide is to be implemented is not defined. If a technique similar to any of those described above is adopted, ATOMIC ACTION's will suffer form the same kinds of reliability problems that they do, namely deadlocks, permanently locked resources and the propagation of inconsistent information through a system. However other mechanisms can be devised to overcome this and the subsequent chapters of this thesis will show how one of these can be developed. We shall also discuss an extension of the ATOMIC ACTION which Lomet has introduced, known as a SHARED ATOMIC ACTION. This allows several processes to collectively acquire shared read/write access to resources, their combined operations appearing atomic to the rest of the system though they may not be atomic with respect to each other.

2.2.6 Communicating processes -

Up until now all the constructs we have looked at have required there to be a set of resources commonly available to all processes, but recently both Hoare (Hoa 78) and Brinch Hansen (BrH 78) have proposed systems which do not require this to be so. These systems are based on the use of messages passed between processes in synchronisation, Hoare's system making communication similar to input/output by allowing processes to read from and write to each other, Brinch Hansen's giving one process the ability to remotely call a procedure in the body of another process. Both these constructs are related to the idea of coroutines, introduced by Conway (Con 63), and the "call", "detach" and "resume" primitives provided by SIMULA (Bir 73), and similar proposals have been put forward by other authors, for example Kahn and MacQueen (Kah 76).

However many of these tend to be directed towards aiding program proving and are not particularly suitable as "real" programming languages.

Both Hoare and Brinch Hansen's systems control the interactions between processes by specifying that a process executing a send operation (that is an output to another process or a remote call) must wait until the process to which the operation is directed specifically accepts the message, in Hoare's system this is done by the use of GUARDED REGIONS. A guarded region is a non-deterministic structure internal to a process, consisting of several procedures callable by other processes, entry to which is controlled by a GUARD, which is a necessary condition for execution to commence. When one of the guards comes true and a call has been issued for the procedure associated with it, the procedure is executed and when it terminates the calling process is reactivated, if several calls are outstanding on one procedure they must be queued and handled one at a time. The use of guards is due to Dijkstra (Dij 75) and is a subject which we shall be returning to in several later sections.

Another feature common to these systems is the ability to describe arrays of processes, thereby allowing messages to be directed to a process whose identity is determined at run time, however Kieburty and Silberschatz (Kie 79) have suggested that some of the interactions expressible using this facility are not implementable on a system made up of distributed processors, and as the use of such distributed systems is the avowed aim of Brinch

Hansen's system, some doubt must exist as to the practical usefulness of the whole system, for many problems require the use of process arrays to solve them, as demonstrated in both (Hoa 78) and (BrH 79). Another criticism leveled against these systems is that the level of concurrency is unnecessarily reduced by the requirement of synchronisation between processes in situations where message passing is the only aim of the interaction as, for example, in the "pipes" available to the user on the UNIX system (Rit 78), and we shall look at this topic in greater depth in chapter 5 below. As to the reliablity of the constructs, all the problems existing in other systems arise for the queueing required to send messages can lead to exactly the same kinds of deadlocks and related errors that queueing for access to common resources can.

Despite these difficulties this type of technique must be regarded as opening up new possibilities in the design of systems, as it encourages highly modular designs, eliminates the need for shared resources, can be applied to distributed systems (though with the reservations expressed above) and allows algorithms to be expressed in new ways. It remains to be seen how successful people will be in using and implementing systems of this type, but the fact that the much disputed tasking mechanism in ADA (Ich 79) is based on the use of communicating processes must encourage more research to be done in this area.

2.2.7 Summary -

This brief examination of the main language facilities for controlling resource usage in concurrent processes has shown that, though constructs exist which are modular and conform to the principles of structured programming, none of them, as they stand, are proof against the occurrence of software errors. Also, support for the sophisticated locking schemes needed for the efficient use of data bases is very difficult (and in some cases impossible) to provide. The main reason for these problems would seem to be that most of the constructs we have looked at were designed for use in the implementation of operating systems, and so cannot be considered as general purpose structures. Only ATOMIC ACTIONS and communicating processes seem to possess the attributes necessary for such a general purpose interface. However both these techniques suffer from the disadvantage that, as they are relatively new, little use has been made of them, so the problems they give rise to are not fully known.

2.3 Recovery schemes for concurrent processes

Now we must turn our attention to the techniques that have been developed to cope with the reliability problems raised by the use of the methods described above. There are four main areas which must be tackled:-

1. If a process becomes dependent on uncommitted data which is later found to be in error it too must be regarded as being

in error,

2. Deadlocks must either be prevented or detected and broken,

3. Resources locked by a process which fails must be unlocked and returned to common availablity, preferably in a consistent state,

4. Some means must be found of detecting processes in unintended infinite loops, and making them fail explicitly, thus causing the release of the resources they hold (because of 3).

The problem of deadlocks has long been recognised and we shall discuss it futher in sectiion 3.7 below, the rest of this section will look at some specific areas of fault tolerance, which relate directly to the other three areas. A general survey of fault tolerance techniques exists in (Ran 78) which gives many examples of the different approaches used in this field, but we shall only look at those which impinge directly on the programmer by providing a language interface to their facilities. However before dealing with concurrent processes per se we must look at recovery facilities in sequential programs.

2.3.1 Sequential programs -

Up to this time only two language facilities for the control of errors have been provided to the programmer. The first is an example of what is known as FORWARD ERROR RECOVERY and is the provision of the ability to handle exceptions, such as the "ON" conditions of PL/I (IBMa). The facility provided by the use of ON conditions is rather limited and several suggestions have been made for expanding the usefulness of exceptions, for example (Goo 75a), (Goo 75b) and (Par 72), which all describe ways of integrating exception handling into the structure of programming language. The use of such forward error recovery techniques is ideal where the programmer knows the exact nature of the faults that may occur in his program, and can precisely define the operations that must take place to recover from them. However, as is shown in (Mel 76), in cases where unforeseen errors arise, possibly due to residual software faults, forward error recovery does not provide an adequate solution. This is due to the fact that the program may not be able to assess fully the damage to the program state caused by the error, so cannot repair it completely.

For this reason approaches based on BACKWARD ERROR RECOVERY have usually been adopted where unforseen errors must be handled. The characteristic of these techniques being the restoration of the program state to that pertaining at some defined time prior to the error, known as a CHECKPOINT. This restoration is made possible by either storing a complete representation of the program state as it was at the checkpoint or by maintaining a record of all the operations which affect the system state, that have been performed since the checkpoint. To provide recovery in the latter case the system must undo all the operations it has recorded, whilst in the former recovery is implemented by making the current program state identical to the stored state. The most widely used backward error recovery mechanism is the RECOVERY CACHE, devised at the University of Newcastle upon Tyne, which provides a means of building up an incremental checkpoint of a program state. Often associated with the recovery cache is a program structure knows as a RECOVERY BLOCK, and both of these are fully described in section 3.4 below.

The main disadvantage of backward error recovery is that it is expensive to implement, requiring processor time and storage to maintain the recovery information needed. However as it provides a means of handling a very wide range of errors (all except those involving the recovery mechanism itself) the expense is generally regarded as worthwhile. Implementations of backward error recovery in sequential programs have been described in (Shr 78b), (And 76) and (Cri 79), the latter combining the use of exception handlers and the recovery cache. Certainly, as we shall see from the succeeding sections, the use of backward error recovery, especially the recovery cache, has been the foundation of most of the recent research done on the field of fault-tolerant software, and this thesis also bases its proposals on this type of mechanism.

2.3.2 Database systems -

Much of the impetus for research into recovery in systems supporting parallelism has come from the need to preserve integrity in large data bases. Verhofstad (Ver 78) has identified seven classes of fault tolerant recovery techniques used in data base systems, these are:-

1. Salvation programs - programs run to restore the data base to a consistent state after an error,

2. Incremental Dumping - periodically archiving data to provide checkpoints for updated files,

3. Audit Trials - recording the order of operations performed on a data base, so that these may be "undone" in the correct sequence in order to reach a consistent state, namely the start of the sequence of operations,

4. Differential Files - here file updates are not made to the main file, but take place in other files which are periodically merged with it and emptied.

5. Backup/current verion - where all the files in the data base are periodically archived to provide a checkpoint should the current information be damaged.

6. Multiple copies - several copies of every file are held, which all contain the same data, except during an update operation,

7. Careful Replacement - updates only take place in COPIES of the data, which are merged with the original, only when the values are committed.

The common feature of these techniques, except the audit trail method, is that they are primarily concerned with the integrity of the data base, and regard the effects of errors on the programs running in the system as secondary. From the programmer's point of view these systems often appear highly fault intolerant, as facilities for error handling are minimal, the usual solution being for all programs affected by state restoration to be rerun. This will usually include several programs which were not in error and which were not dependent on erroneous data, and, even if the jobs are resubmitted without involving the programmer, a considerable lengthening of the turn-around will be experienced by those users affected.

However systems that use audit trail methods can BACK OUT an individual process, rather than BACKING UP all the processes in the system providing a much better user interface. The programmer can then be given facilities to control the way in which recovery takes place in his programs, very much as for sequential programs (note that the recovery cache mechanism can be viewed as an optimised audit trail as well as an incremental check point). An example of this is provided by System R (Ast 76) which gives the programmer the operations SAVE, (to identify the start of an audit trail), RELEASE (to commit an audit trail) and RESTORE (to wind back to a named point in an audit trail). This type of interface is very often unstructured because of the COBOL-like languages used in data base systems, but they must be regarded as a major step forward in making error handling by fault tolerant methods available and familiar to a much wider spectrum of users.

2.3.3 Ports -

Outside the sphere of databases much of the work on error recovery in concurrent systems has been theoretical and very little attention has been given to the needs of the programmer. However Shrivastava and Banatre (Shr 78a) have described a program structure known as a PORT which allows competition and cooperation between processes whilst preserving recoverability. A PORT is a specialized type of SIMULA class with features to aid recovery, and is used to control access to resources. It consists of some variables, some entry procedures, a reverse procedure and an initialisation statement consisting of two parts known as a PRELUDE and a POSTLUDE separated by means of a SIMULA INNER statement. The prelude of a PORT is concerned with resource acquisition and the postlude with resource release, and they are invoked by means of the USING statement. This associates a block of statements with an instance of the PORT, and causes the block to be executed when the INNER statement of the PORT initialisation code is reached, ensuring that the block is bracketed by the prelude and postlude. The PORT assumes the presence of a recovery cache and if an error occurs whilst a USING statement is executing, state restoration will take place, and the postlude of the PORT will automatically be executed, guaranteeing that resources will be released by failing processes. However if an error occurs after a USING statement has terminated the effects of the operations performed in it must be undone, and this is where the special reverse procedure comes in. The use of a PORT is recorded in the recovery cache and when state restoration takes place the prelude and postlude are executed to reacquire and release the resources used, the reverse procedure being executed when the INNER statement is encountered. This procedure is only accessible to the recovery mechanism and is to provide compensation for the effects of the previous use of the PORT, that is, because the USING statement cannot be undone by the system, the user must provide a piece of program which attempts to do this. In most cases this will involve sending a special message to other processes or the system operator to tell them to ignore certain data, but it can also consist of constructing messages whose effects will be the exact opposite of the earlier message, thus undoing its effects. Another facility provided to the user is an errorflag which allows him to determine, during the postlude and prelude, whether the process is recovering or not, and so these parts of the program can be tailored to the recovery process also.

```
TYPE MANAGER IS
MONITOR
   PROCEDURE ENTRY SEND ( I : INTEGER ; LAST : BOOLEAN );
   -- queue value for receiver, last is true if it is final value
   BEGIN...END ;
   PROCEDURE ENTRY RECEIVE(I : OUT INTEGER ; LAST : OUT BOOLEAN )
   -- get a value form the queue
   BEGIN...END;

   BEGIN...END;  -- initialisation code

TYPE SENDER IS
PORT( MAN : MANAGER ; I : INTEGER ; LAST : BOOLEAN )
   VALUE : INTEGER ; COMPENSATE : BOOLEAN ;
   REVERSE PROCEDURE;
   BEGIN END; -- called as INNER when recovering

   BEGIN
      -- first of all the prelude
      IF ERRORFLAG THEN -- system is recovering
      BEGIN
         COMPENSATE:=TRUE; -- indicate this to postlude
         MAN.SEND(VALUE,LAST) -- send compensating value to receiver
      END
      ELSE -- its a normal exchange
      BEGIN
         VALUE:=-I; -- record compensating value for this exchange
         COMPENSATE:=FALSE; -- for postlude
         MAN.SEND(I,LAST) -- send the value
      END;
      INNER;  -- perform the user's code
      -- now the postlude
      IF ERRORFLAG AND NOT COMPENSATE THEN
         MAN.SEND(VALUE,LAST) -- compensate for error in INNER
    END;

LINK : MANAGER ;
SEND : SENDER ;  -- port to control LINK

TASK PRODUCER;
BEGIN
   ENSURE (...) BY -- some acceptance test
   BEGIN
     FOR I:= 1 TO 3 DO
        USING SEND(LINK,I,FALSE) DO ; -- send the numbers
   END
   ELSE BY
   BEGIN...END -- a secondary algorithm
   ELSE BY ERROR;

   USING SEND(LINK,0,TRUE) DO;  -- signal termination
END;
```

Fig. 2.3 ...

```
TASK CONSUMER;
BEGIN
   LAST:=FALSE ;
   SUM := 0;
   WHILE NOT LAST DO
   BEGIN
      LINK.RECEIVE(I,LAST)
      IF NOT LAST THEN SUM:=SUM+I
      ELSE
      BEGIN...END  -- use the result
   END
END;
```

Fig. 2.3

Fig. 2.3 shows an example of how a PORT can be used to provide reliable communication between processes. It consists of two processes 'PRODUCER' and 'CONSUMER' which communicate via a MONITOR 'LINK'. The function of the system is for CONSUMER to calculate the sum of the numbers generated by PRODUCER, and the use of the PORT 'SEND' ensures that the sum is correct even if an error occurs in the producer. This is because each time the port is used to send a message a compensating value (the negative of the value being sent) is stored in PRODUCER's cache. Should an error occur in PRODUCER, the 'reverse' invocations of SEND will cause these negative values to be sent to CONSUMER, thereby correcting the sum that it holds. In this example the reverse procedure is actually null, and the compensating action is taken in the prelude and postlude of the PORT. This is to allow for the possibility of errors occuring whilst the INNER statement of the PORT is being executed, in which case the test on ERRORFLAG in the postlude will ensure that compensation takes place.

The main advantage of this scheme is that processes do not become dependent on each other and so recovery of an individual process can take place, meaning that the DOMINO EFFECT (Ran 75) will be avoided. This occurs when processes have interacted and become dependent on each other in such a way that successive state restorations must take place, until the processes have been wound back to their first checkpoints. The effectiveness of this structure has been demonstrated in an implementation based on Concurrent Pascal and the SOLO operating system (BrH 76), which is described, with several examples in (Shr 79a) and (Shr 79b).

From the programmer's point of view, however, there are several disadvantages. Firstly the suggested use of MONITOR's with PORT's, brings with it all the difficulties described in section 2.2.4. Secondly the use of the errorflag makes the prelude and postlude rather inelegant, giving the appearance of a somewhat adhoc addition. Thirdly, the use of PORT's adds considerable complexity to programs, as can be seen from the examples in (Shr 79a). Finally the need to provide compensation will restrict the user in the ways in which he can solve problems, circumstances being made even worse by the possible presence of errors in reverse procedures. There are also disadvantages from the point of view of recoverability because the programmer cannot be guaranteed that his attempts at compensation will have any effect, for other processes may have used the erroneous data and terminated before the error was discovered causing faulty results to be committed. In order to overcome this the programmer must ensure that processes are properly synchronised, but in cases where interactions take place with unknown processes (that is processes created by other programmers) it can never be certain whether they will behave in the correct manner. This problem does not occur in the limited environment of a Concurrent Pascal program, but in "real" systems this situation will arise fairly frequently.

2.3.4 Deadlines and safe programming -

None of the systems we have looked at so far have offered a solution to the problem of unintentional infinite loops or waits that was identified above. All of them would require manual intervention to stop the execution of a looping process. Of course a loop need not be infinite to be in error - especially in real time systems where the time taken to execute sections of program can have a significant result on the system state. Anderson (And 75) has attacked this problem by introducing a new approach to the way in which looping constructs are used in programs. He has identified three control structures as being necessary:-

1. REPEAT S UPTO N TIMES - which, with "EXIT" statements in S, can be used to implement constructs such as "WHILE" and "UNTIL" statements,

2. DO S EXACTLY N TIMES - where S contains no EXIT statements, thus providing "FOR" statment facilities,

3. CYCLE S INDEFINITELY - again with no EXITs in S, here the

programmer specifically intends this to be an infinite loop.

These constructs allow the more usual loop errors to be detected, but do not provide any way of trapping erroneous waits and loops (and other errors) caused by the corruption of the internal representation of a program. These can often be detected by checksumming straight line sections of code (that is sections containing no control transfers) at compile time and run time. A comparison of the two values can be carried out at the end of each straight line section - if they are not identical the code has been corrupted, so a failure condition can be raised in the process executing the code. Another approach adopted in the PLURIBUS system (Hea 73,Orn 75) is to periodically checksum all the program modules that make up the system, again comparing the value obtained with one that was statically determined at compile (or load) time. These two methods are not foolproof, as errors could cancel each other out (especially in the latter case where much larger quantities of data are being checked), but they will significantly enhance the reliability of a system by trapping errors which may not otherwise be detectable (though note that the former method will detect an error the first time a corrupted section is executed, whereas with the latter corrupted code could have been exeucted several times before it is detected).

Another technique which is often used for trapping faulty loops and waits is based on the use of watch-dog timers. This involves setting a maximum execution time for a program section and starting a timer when it is entered, if the time allotted is exceeded an error condition is raised. This technique has been used in the PLURIBUS system and was also suggested by Dennis and Van Horn for the system they describe in (Den 66). Horton and Campbell (HoC) have formalised this use of timers into the concept of DEADLINES, which allow for the detection and recovery from possible timing errors. The structure they develop allows the programmer to assign a maximum time limit to a section of program and to provide a recovery block structure (see section 3.4) to handle any errors that occur. The scheduling of the execution of such sections is very critical, and Liestman and Campbell (Lie 80) have shown how certain optimal schedules for systems using deadlines can be achieved and how idle time created by program successes (that is within their deadlines) can be rescheduled. Deadlines have their most important applications in real-time systems, (for example fig. 2.4 which shows a navigational application, due to Campbell), but should prove to be useful in all cases where looping errors occur.

```
EVERY SECOND -- frequency at which process is to be executed
SERVICE POSITION_UPDATE
WITHIN 1 MILLISECOND -- deadline time limit
BY
BEGIN
   READ_NAVIGATIONAL_DATA;
   CALCULATE_NEW_POSITION;
END
ELSE BY
   ESTIMATE_POSITION_FROM_OLD_DATA;
```

Fig. 2.4

2.3.5 <u>Distributed systems</u> -

Recently considerable attention has been given to the theoretical aspects of backward error recovery in concurrent systems and Merlin and Randell have developed a formal method of describing the concepts involved, by the use of "Occurrence Graphs", which are similar to Petri's Causal Nets (Pet 77). The special characteristic of Merlin and Randell's graphs is that they are regarded as being created and recorded dynamically by the system that they are modelling as it executes and they also have some extra features geared towards the problems of state restoration. One of the important concepts that they introduce is that of RESTORABLE PLACE or RECOVERY POINT (Ran 78), which is a point in the execution of a process to which the process can be returned, because checkpoint information had been built up after it, enabling state restoration to take place. Where the state restoration of several processes is related the set of recovery points to which they are wound back is known as a RECOVERY LINE.

In order to constitute a recovery line the set of recovery points chosen must each belong to a different process, and be consistent, that is:-

1. One of the recovery points must belong to the process in which the error that initiated recovery was detected;

2. No information must have passed between any two of the

processes in the set betweeen the saving of their recovery points;

3. No information must have passed between any process external to the set and a member of the set after its recovery point was saved.

It is the search for such a set of recovery points that causes the domino effect, described in section 2.3.3 to occur.

Merlin and Randell describe various operations which can be performed on occurence graphs, but their most important result is the development of what they term a "Chase Protocol". This can be used in a decentralised recovery mechanism for a distributed system and they present a proof that the use of such a protocol will provide system recoverability even when there are several faults in the overall system. Such a protocol involves the sending of messages between the modes of a system to propagate recovery activity; it is called a "chase protocol" because processes which are dependent on erroneous data will continue to execute normally, until the failure message manages to reach them. This means that there may be times at which the system state is inconsistent, but it is guaranteed that a consistent state will be reached after a finite, though perhaps arbitrarily long time.

Occurrence graphs are a useful tool for representing the state of concurrent systems, and the work of Best (Bes 78, Bes 79) on atomicity has extended their utility considerably. At the moment the work being done is purely theoretical, however it can be expected that ideas with practical application for the programmer will result from it.

2.4 Spheres of Control

As we have just indicated the theoretical aspects of both resource control and reliablity have been studied in considerable depth, and both topics are now fairly well understood. We have seen that the work on occurrence graphs has produced a flexible and powerful tool for modelling concurrent systems and the effect that recovery has on them, and that work is being carried out on modelling atomicity using them. However occurrence graphs are a mathematical tool and as such are inaccessible to many people, and the work of Davies on SPHERES OF CONTROL (Dav 73, Dav 79, Bjo 73) provide a simpler way of characterising the problems of resource control and recovery. Davies defines a sphere of control (SOC) as "a boundary around the effects of a process for the specific purpose of controlling commitments", and states that each SOC is an atomic process when seen from its enclosing level of control. He goes on to define three kinds of resource usage:-

1. Reference - where the value of a resource may alter at any time and the supplier of the resource does not have to inform users of the resource of the change, for example the system clock,

2. Dependent - where the supplier of the resource must inform the users of any change in value, and they in turn must give up the resource on demand, (providing backward error recovery)

3. Committed - which is the same as dependent, except that the user is not obliged to give up the resource when requested to do so, so a compensation function must exist to correct the system. (providing forward error recovery).

Then, using the definitions, he shows the steps needed to initiate and terminate a sphere of control so that integrity is preserved. In (Dav 79) he extends these notions to cover such things as consistency, auditing and the scheduling of transactions, providing a complete framework within which the problems of reliability can be discussed no matter whether the systems in question are based on computers or are completely operated by humans. Some of the ideas he presents could be built into real systems, however as yet there are still concepts which we do not know how to implement efficiently (or sometimes at all) and more research must be done before the benefits of such theoretical work reach the programmer.

2.5 Conclusion - the programmer interface

From the foregoing we can see that though several well structured methods of resource control are available to the programmer little thought has been given to making them recoverable and providing the user with adequate error handling facilities. The systems that have been developed all use the recovery block as the basic building unit for structured error handling, and this reflects the fundamental simplicity of the construct. However in order to allow reliable control of access to resources additional features, such as PORTS, must also be provided to augment whichever of the locking constructs is adopted. This makes the task of writing programs more complex and means that the code intended to increase the reliablity of programs could be a source of software error.

The access control methods discussed do not provide particularly good interfaces to the user, most of them being primarily concerned with implementing mutual exclusion, and not being flexible enough to allow extensions to include other lock modes. Only atomic actions could easily provide this facility and of the schemes requiring the use of shared resources they provide the interface which fits most easily into the widest range of languages, being closely related to the block structure of programs. However the use of communicating processes opens up a different set of possibilities and may require the development of completely new approaches to the way in which processes interact.

Of all the language interfaces and facilities we have looked at above, only the use of safe programming and deadlines combine reliability and simplicity and do not disadvantage the user in any way. These two features must be considered as an essential part of any highly reliable programming system because they are the only defence against looping and waiting errors. Some means of detecting code corruption must also be considered, as this also provides error detection that cannot be attained through any other means.

In general then, the programmer is not well provided for in the realms of resource control and reliability, probably due to the theoretical nature of most of the work done in this field. However this will have to change quite drastically if the present trend towards distributed systems continues at its present rate.

3.0 Uncooperative Processes

3.1 Introduction

In section 1.2 we introduced the concept of "uncooperativeness" to describe processes which release all the resources they have acquired together, before either terminating or starting to build up a new set of resources. In this chapter we shall show how a system supporting such processes providing full recoverability and a simple user interface can be constructed by using a modified version of the recovery cache mechanism (Hor 74, Ran 75). The system described will also form the basis for facilities to be developed in the following chapters that will allow cooperative processes to be supported, and the reader should bear this in mind throughout. Uncooperative processes form a very important class as most of the jobs run on any computer system either are or could easily be made "uncooperative". Therefore any changes in the way in which they are handled must not introduce cumbersome constructs to the user interface, nor should they adversely affect the efficiency of the system without providing some compensating benefits. We shall endeavour to show that the system developed here achieves these ends.

3.2 Multi-level systems

We must first briefly describe the abstract structure within which the system to be described is designed to fit. This will be assumed to consist of a series of LEVELS, one above the other, where each level provides a more abstract view of the underlying machine to the levels above it. Levels may be used to provide new facilities or to hide existing ones in a similar fashion to the CLASS structure of SIMULA (Bir 73) or the PRIVATE types of ADA (Ich 79). This new view provided by a level constitutes an INTERFACE for the level immediately above, programs written to run at the higher level being expressed in terms of the operations and types defined by the interface. Note however that the level supporting this interface "sees" the user's program in terms of operations and types provided by still lower levels. This is especially important to remember for interfaces supporting concurrency in the level above them. In this case it is not necessarily possible for the lower level to determine the validity of a sequence of interacting operations because the user expressed relationships between processes are not meaningful to it.

The idea of multi-level systems is discussed more fully in (Ver 77) and (And 78), both of which relate closely to the work described in (Dij 68b). However for the purposes of this thesis we need only be concerned with the topmost two levels of the system and the interface that lies between them. We shall call the higher the USER level, and the lower the INTERPRETER level. The interpreter level will be assumed to provide COMPLETE RECOVERABILITY to the user, that is every type provided by it is RECOVERABLE. This means that should an error arise in the user level (either detected by the interpreter or explicitly signalled by a program) the interpreter guarantees to be able to restore the states of objects of any type which have been altered to those pertaining at some previous user specified point in the execution of the program. Always assuming that the user has actually made use of the facility to indicate such points. If an interface does not provide complete recovery for every type it is said to be PARTIALLY RECOVERABLE. These are fully discussed in (Ver 77) and will not be considered further here.

Anderson et al (And 78) have introduced the concept of INTERPRETER EXTENSIONS to describe programs that provide users with new abstract types in addition to those supported by an existing interpreter. They have described two types - DISJOINT and INCLUSIVE, the difference between them being that in the former, recovery information for the new types is held by the program providing the extension, and in the latter this is held by the programs using the extended facilities. However, for the purposes of this thesis the use of interpreter extensions, and the problems this introduces, need not concern us and it will be assumed that the interface seen by a user is not provided by an interpreter extension.

3.3 Specification of uncooperative processes

Before embarking on the description of an interpreter to support uncooperative processes we must first set out the facilities that we intend it to provide the user. However we are not concerned with the actual details of the language in which the user expresses his programs. This topic is discussed briefly in section 3.9.1 and an actual example described in section 3.9.2.1.

There are five main features:-

1. Common resources should not be differentiated from private resources except by their declaration. This would mean, firstly, that the user need not be aware of the need for locking of resources, secondly that existing software, such as library procedures, could be used just as well on common resources as on private ones. Thirdly, that no program module using common resources is dependent on another module having been executed before it for the purpose of acquiring the resources, making testing easier. Finally that the system can easily hide the fact that some resources available to the programmer are actually common rather than private,

2. The user is always protected from the activities and errors of other users,

3. In the event of an error being detected in a program, causing it to fail, common resources that it was using will be released and left in a consistent state,

4. It should be easy to make existing programs uncooperative,

5. The above facilities should not impair the efficiency of the system and should be usable for real-time applications.

These five are fully in accord with the characteristics required of a programmer interface set out in section 2.1.4.

3.4 Recovery blocks and the recovery cache

As we have just seen, for the facilities of a completely recoverable interface to be of any utility the user must indicate certain points in his programs which can be regarded as checkpoints. In chapter two we saw that this facility could be provided by the use of RECOVERY BLOCKS supported by a RECOVERY CACHE, a scheme developed by the highly reliable computing systems project at the University of Newcastle upon Tyne. This has been described in (Hor 74) and (Ran 75), however as its use is so fundamental to the work developed in this thesis its characteristics and operation will be set out here.

A recovery block consists of a set of ALTERNATES, which are realisations of the algorithmn that the block is to implement, and an ACCEPTANCE TEST, which is used to determine the validity, of the results produced by the execution of an alternate. One of the alternates is designated the PRIMARY and is always executed. Should an error occur during its processing, or if the results it produces do not pass the acceptance test, the program state is restored to that existing just prior to the start of the recovery block and the SECONDARY alternate is executed. This process is repeated until an alternate executes successfully and passes the acceptance test or until either no more alternates are available or a deadline associated with the recovery block expires. If either of the latter

```
ENSURE <AT1> BY
BEGIN
   .                                        .......... (a)
   ENSURE <AT2> BY
   BEGIN
      .                                     .......... (b)
      ENSURE <AT3> BY
      BEGIN
         .
         .                                  .......... (c)
      END
      ELSE BY
      BEGIN
         .                                  .......... (d)
         .
      END
      ELSE ERROR;
      .
   END
   ELSE BY
   BEGIN
      .
      .                                     .......... (e)
   END
   ELSE ERROR;
END
ELSE BY
BEGIN
   .                                        .......... (f)
   .
END
ELSE ERROR;
```

Example Execution Sequences

a,b,c	- no errors
a,b,c,d	- error at c
a,b,c,d,e	- errors at c and d
a,b,f	- error at b

Fig. 3.1

events takes place the recovery block is said to have FAILED and an error is raised. Should the failing recovery block be nested within another, the enclosing alternate is wound back and the process of error recovery continues at its level. However if the error occurs in an outermost recovery block the program containing it has failed and no more recovery can take place. An event of this sort is designated a CATASTROPHE. Fig. 3.1 shows a typical nested recovery block structure and how a typical execution may progress.

Some other points to note about recovery blocks are:

1. A primary alternate must always be present, but the presence of other alternates is not mandatory,

2. Alternates need not be distinct, that is any alternate may be RETRIED, the number of times that it is attempted being specified by the user, an undefined number being permissible only when the recovery block has a deadline associated with it,

3. In the case of nested recovery blocks an expiring deadline causes restoration to take place to the start of the block enclosing the statements associated with the deadline, rather than the block executing at the time - deadlines cannot be extended and an enclosing deadline takes precedence over those set up inside it.

The mechansism used to provide the state restoration required by recovery blocks is the recovery cache which, as we have seen, is in essence a device for providing an automatic audit trail by arranging that when an operation takes place that alters the state of a resource, its previous state is recorded before the update is carried out. This ensures that the effects of the operation can be undone should an error occur. The advantages that the recovery cache has over other audit trail mechanisms are that only the first state change occuring to a resource in a recovery block is recorded, all others being unnecessary for state restoration, and that only those resources actually changed are entered into the cache. Let us now look at the mechanism in detail to see how these are achieved.

The cache itself is a stack consisting of elements which are either BARRIERS or records of a state change. Each time a recovery block is entered a new barrier is created in the cache, and the RECOVERY LEVEL is incremented. The barrier represents the point to which state restoration takes place for this recovery level, therefore by undoing all the state changes recorded after the barrier was created the program state will be identical to that existing when the barrier was created. The records of state change consist of three elements - firstly a pointer to the resource which was altered, then some representation of its state before it was altered and thirdly a copy of the recovery level field associated with the resource. Every object that can be cached has one of these fields which is used to record the last level at which a state was cached for the object. It is this field that is used to prevent multiple entries being made for a resource at any one recovery level. This is achieved by comparing the recovery level field with the current recovery level; if they are equal there is no need to cache the item, otherwise the object is cached and the recovery field is updated to contain the current level thus eliminating further cacheing. This has the additional advantage of allowing local variables of a recovery block to escape being cached, their recovery level fields being initialised to the value of the current recovery level when they are created.

When an error occurs and the entries are restored as described above, the operation is known as <u>REJECTION</u>, but if the recovery block terminates successfully the cache must be <u>ACCEPTED</u>. This activity is more complex than that of rejection because in order to maintain the process's recoverability the record of state changes must be cumulative. That is if a state change has taken place at level N to a resource not cached at level N-1 the entry must be preserved in the cache to allow level N-1 to recover fully. This occurence is, however, easily detected because if an entry needs to be preserved the value of its recovery level field stored in the cache will not be the same as the recovery level enclosing the block which is being processed. (Note that the barrier entry for a recovery block is deleted when the block is accepted).

This description is of course of a cache for unshared resources, and in the next section we shall show how this mechanism must be altered to support shared resources.

3.5 A recovery cache scheme to support uncooperative processes

One characteristic of recovery blocks which only becomes apparent when concurrency is required is that during the execution of an alternate the state changes made to resources are not committed. Hence, if the resources in question are shared, other processes must be prevented from using them until the alternate terminates successfully. If this is not done processes may build up dependencies upon each other which could be very complex and therefore difficult to undo in the event of an error. In order to avoid this every alternate must be atomic, therefore every shared resource used within an alternate must be locked and cannot be released until the alternate terminates. Thus any alternate of a recovery block can be represented by an ATOMIC ACTION (see section 2.2.5), as has been suggested by Lomet (Lom 76a). Of course an atomic action on its own is _not_ equivalent to a recovery block for its use carries no implication about the collection of recovery data, so we shall introduce the concept of a _RECOVERABLE ATOMIC ACTION_. This has all the properties of an atomic action, with the addition that at any time before its termination the action may be wound back to its starting point, releasing all the resources acquired during its execution without affecting the environment as seen by other processes. The advantages to be obtained from the use of recoverable atomic actions will become clear later. Suffice it to say here that from now on any unqualified reference to an atomic action or simply action will mean a recoverable atomic action, and that "recovery block" may be substituted for any such reference without affecting its meaning. Where a set of nested actions is intended we shall use the terms "process" and "program", which will be taken to refer to all the actions existing at a given instant. Now we shall examine the requirements of a system to support recoverable atomic actions.

The properties such a system must possess are:-

1. A recovery cache to collect state restoration data,

2. The ability to recognise the use of shared variables and place an appropriate lock before proceeding,

3. A book keeping scheme to record locks put on by an action, which can be used to release locks for the current action when an error occurs, but which will otherwise accumulate all the locks placed until the end of the outermost action.

Inspection of the above shows that properties two and three are very similar to those of the recovery cache, though with respect to lock status rather than "value". This suggests that it is possible to devise a cache mechanism which will handle both locking and state saving and the rest of this section will be devoted to the description of such a scheme. From now on we shall refer to the scheme described in the previous section as the simple recovery cache.

In order that the simple recovery cache may operate, each resource must have a recovery level field associated with it, and to support atomic actions each shared resource must have a field in which its lock status is recorded. However ANY resource may become shared if a process forks into several sub-processes which share the use of its private resources, so every resource would have to have a lock field as well as a recovery level field, which could mean an unacceptably large storage overhead. The solution to this is to combine the lock field and the recovery level field into a single <u>ATOMIC ACTION INDENTIFIER FIELD</u>. The value in this field will be such that

1. It will differentiate between different nested levels of actions within a single process, just as the recovery level does for nested recovery blocks.

2. It will allow the interpreter to determine, by inspection, whether a particular process has access permission to the resource to which the field is attached.

To achieve this it is necessary that every time an action is entered a unique identifier be generated for it and to store, in a area accessible to all processes and addressable using the generated identifier, the identity of the process that initiated the action. Similar requirements are needed for the identifiers used in information protection schemes (Den 66, Sal 75) and, as in those cases, they must be generated by the interpreter and not the user. This is for several reasons :-

1. User generated names may not be distinct,

2. Actions may be entered recursively,

3. The user may not identify the action (this occurs where shared atomic actions are used - see chapter 4),

4. User generated names could be "forged" to obtain access to resources where none is allowed.

Assuming that these requirements have been met the operation of the recovery cache will now be as follows :-

If the identifier field of a resource has the same value as the identifier of the current action then proceed, (cf. recovery levels)

otherwise wait until either the process is a member of the set of processes addressed by the identifier field or the field is null (that is the resource is not locked), then perform the usual encache operation (store the resource address, its state and its identifier field and update the field to contain the current action's identifier).

This sequence, if used in the simple cache environment, would successfully handle updates, however it is also necessary to lock common resources which are "read". Therefore operations that do this must recognise when a common resource is used and cache it if necessary. Cacheing of ALL reads, shared and private alike, would eliminate the need for this recognition (though it effectively takes place in the memory management hardware anyway), but this would presumably present unacceptable overheads in terms of cache size.

Fig. 3.2 shows various stages of cache growth for a simple process. It is important to note how the cache acceptance algorithm developed for the simple recovery cache guarantees that locks are accumulated until the outermost action terminates. The rejection algorithm also guarantees that locks placed by an action are released when an error occurs.

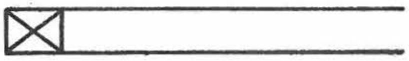
The size of the units which are cached we can term the GRANULARITY OF CACHEING and is simply related to the granularity of locking for the system. It must never be greater than the granularity of locking because the cache may then be recording data which the process has no access to, which can cause interference between processes if the cache has to be backed out. If the granularity of locking is larger than the granularity of cacheing extra action identifier fields will be needed to control the locking of groups of resources. However as these resources must all have such fields in order to control their recovery this would seem to be redundant, but it may provide advantages as far as preventing deadlocks and reducing waiting times for resource requests. For simplicity though we shall assume from now on that the granularities are equal.

The scheme we have just described only caters for requests in one lock mode (exclusive) but it can be adapted to implement the more complex locking schemes as described in section 2.1.3. However it must be remembered that as we are dealing with uncooperative processes locks cannot be released (except in the case of error)

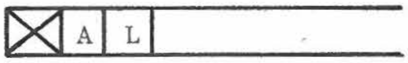
```
COMMON A,B,C : ....  -- some shared variables

TASK EXAMPLE;
L,M,N : ......       -- some local variables
BEGIN
   ENSURE ... BY     -- ......................(a)
   BEGIN
      L := A;        -- ......................(b)
      B := M;        -- ......................(c)
      ENSURE ... BY  -- ......................(d)
      BEGIN
         C := N;     -- ......................(e)
         M := B;     -- ......................(f)
      END
      ELSE ERROR ;
      L := C;        -- ......................(g)
   END
   ELSE ERROR;
END;                 -- .......................(h)
```

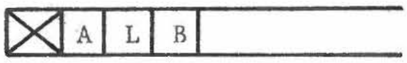
<u>Cache States</u>

a) - 

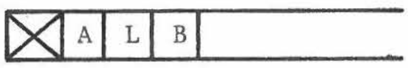
Create a new barrier in the cache.

b) - 

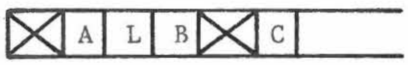
Cache/lock A as it is shared, update and cache local L.

c) - 

Cache/lock B, but not M as access is a local read.

d) - 

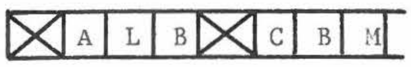
Another new barrier in the cache.

e) - 

Cache/lock C, but not local N.

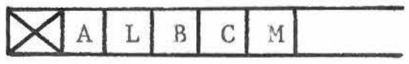
Fig. 3.2 ...

f) -



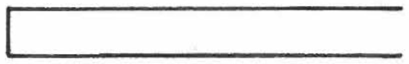
Cache B for this recovery level and updated local M.

g) -



Accept cache - maintain C and M from previous level.

h) -



Final cache acceptance - A, B and C are now released.

Fig. 3.2

until the process terminates. They can only be made more binding. The next section demonstrates this by giving the rules for controlling locking when exclusive and shared read locks are provided. The problem of deadlocks will be discussed in section 3.7.

3.6 Rules for providing X and SR locks

3.6.1 Non-Preemptive Systems -

Let us look first at the rules governing non-preemptive systems, that is systems where processes may have different priorities associated with them, but where a process of high priority cannot "snatch" a resource from a process of lower priority which is using it. However the queue of processes waiting to lock the resource could be priority ordered, thus ensuring that a high priority process will be delayed for as short a time as possible. Fig. 3.3 shows the

RESOURCE STATE / ACTION REQUEST	FREE	SR LOCK BY SAME ACTION	SR LOCK BY OTHER ACTION AT PRIORITY P2	SR LOCK BY OTHER ACTION AND SAME ACTION PRIORITY P2	X LOCK BY SAME ACTION	X LOCK BY OTHER ACTION AT PRIORITY P2
X LOCK AT PRIORITY P1	APPLY X AND CONTINUE	CONVERT TO X LOCK AND CONTINUE	P1>P2 : PRE-EMPT APPLY X AND CONTINUE P1<=P2 : JOIN QUEUE	P1>P2 : PRE-EMPT AND CONVERT TO X LOCK P1<=P2 : JOIN QUEUE	CONTINUE	P1>P2 : PRE-EMPT APPLY X AND CONTINUE P1<=P2 : JOIN QUEUE
SR LOCK AT PRIORITY P1	APPLY SR AND CONTINUE	CONTINUE	APPLY SR AND CONTINUE	CONTINUE	CONTINUE	P1>P2 :* PRE-EMPT APPLY SR AND CONTINUE P1<=P2 : JOIN QUEUE

*Note that here one may wish to wait to obtain the latest value.

Fig. 3.3

various combinations of lock request mode and the mode in which a resource is actually locked with the operations needed to acquire the resource. This diagram is related to a preemptive system, but if the reader lets the priorites P1 and P2 be equal the rules for a non-preemptive system will be obtained.

3.6.2 Preemptive System -

In a normal preemptive system a process of high priority may take a resource from a process of lower priority, halt that process, save the state of the resource, use the resource, and then restore its previous state, proceeding with its execution after reactivating the halted process. This is akin to the concept of lending mentioned in section 2.1.1, and assumes that the resource can be placed into some defined initial state after saving its current state, but this is a special case which will be dealt with in section 5.4 below. Since in general such an assumption cannot be made preemption is not often implemented, however as the system we are dealing with supports recoverable atomic actions preemption can take place. This is done by causing the action holding the resource in question to be wound back, thus releasing the resource in a consistent state, allowing it to be locked by the high priority process. There are several points to be aware of here:-

1. Preemption is not an error condition so an action that is backed out is restarted, if it is part of a recovery block the next alternate is not taken,

2. Deadlocks can only occur between processes of equal priority, as the system behaves as if it were non-preemptive in that case,

3. A special case arises when a process wishes to preempt a resource which is held exclusively by another process. Here the high priority process may wish to wait until the resource is released in order to obtain its most recent "value". This would seem to be advantageous, except where the extra delay involved would prejudice the performance of the process.

Fig. 3.3 shows the operations required for a preemptive system, and assumes that the priority of an action is identical to that of the process which initiated it. Where the resource is not locked the request may be granted immediately, and where the resource has already been locked by the action (or one of its enclosing actions) the same is true, though a conversion must take place when an X lock is requested on a resource held in SR. The remaining cases arise where the resource is already locked by other actions not enclosing the requesting action. In all cases, if the lock request and actual lock mode are compatible, the request is granted, but if this is not the case preemption may be possible. To determine if this is so the priority of the requesting action is compared with the maximum of the priority of the actions currently holding the resource (excepting itself should it be one of them). If it is greater, all the actions are wound back and the requesting action proceeds, possibly performing a lock conversion from SR to X in the process. Otherwise the action must wait until its request can be granted.

Similar systems have been described by Gray et al (Gra 76) and Chamberlin et al (Cha 74), the latter however restricting the places where preemption can occur to preserve consistency (see section 3.10).

3.7 Deadlocks

3.7.1 Deadlock prevention and avoidance -

Methods for handling system deadlocks are usually classified into three types after (Cof 71), these are:-

1. Deadlock prevention - the design of the system excludes, a priori, the possibility of any deadlock occurring,

2. Deadlock avoidance - programs must predeclare the usage they wish to make of shared resources. The system then analyses these requests in the light of other outstanding requests, and allows those programs whose requests are <u>SAFE</u> (that is will not cause a deadlock) to proceed.

3. Deadlock detection and recovery - The system has the ability to detect when a deadlock has or will occur, and to break

the deadlock in some way.

Systems providing deadlock prevention usually rely on special knowledge of the mix of programs they are to support, though many systems described as preventing deadlocks do so by the use of avoidance techniques. These techniques usually support <u>STATIC</u> allocation of resources, that is all the resources that a user has indicated he requires are given to him at the start of his program, when his request is adjudged to be safe. This means that users may have possession of resources for far longer than they actually need them.

Habermann (Hab 69) has, however, developed an algorithm where the user states the upper limit of his requirements, and is allowed to acquire resources dynamically as his program proceeds. However this solution and that of Holt (Hol 72) are only designed for systems providing <u>ARBITRARY</u> resources, that is where a user is constrained to specifying the type of resource he requires rather than precisely identifying a particular resource. Lomet (Lom 76b) has developed an algorithm, supporting both exclusive and shared read requests, which overcomes this disadvantage and can be used in, what he terms, <u>UNIT-RESOURCE</u> systems, such as those providing access to data bases. This method still depends on the predeclaration and static allocation of resources, and so cannot support programs whose requests fall into the following categories:-

1. Non-unique resource name - where one resource may have

several descriptions, which are not distinguishable,

2. Modifiable resource categories - where operations on the resource can change its description,

3. Interdependent locks - after locking one resource and examining it a program may decide to lock another resource, the identity of which depends on the results of the examination.

To provide support for such requests a dynamic resource allocation scheme must be adopted, and this requires that deadlocks be detected and recovered from. Chamberlin et al (Cha 74) develop an algorithm for this, which uses preemption of resources to break deadlocks, but constrains the user as to the way in which requests for resources are made (see section 3.10). This scheme possesses the disadvantage that a process may be kept waiting indefinitely if its resources are continually being preempted by other processes. This is also a possibility with Habermann's method, and Holt (Hol 71) has suggested associating a time limit with a resource request, after the expiry of which, the request must be granted, Chamberlin et al use a related solution with respect to preemption.

The implementation of recoverable atomic actions requires a deadlock detection and recovery solution as they require a dynamic resource allocation strategy to support requests of the types described above. Recovery can of course be provided by backing out one of the deadlocked actions, forcing it to release the resources it holds, which will break the deadlock. However the deadlock must first be detected, and we shall describe a method for this in the next section.

3.7.2 Deadlock detection for recoverable atomic action -

Most of the deadlock detection algorithms mentioned in the previous section are very complex. This is because they are usually designed to handle multiple requests for specified resources classes from a process and must delay the granting of the complete request until it can be guaranteed that a deadlock will not occur. However because the recovery cache mechanism can only operate on one resource at a time, and it is the part of the system which issues lock requests, the need to check multiple requests from a process is eliminated and this allows the deadlock detection scheme used to be very much simpler. There are two ways in which deadlocks can arise when resources are being allocated and we shall now show these can be detected.

The first deadlock is of the type known as the DEADLY EMBRACE (Dij 68), and occurs when an action A attempts to lock a resource held by another action, B, whose progress is BLOCKED by A. Such blocking occurs either directly, when A (or one of its enclosing actions) holds a resource required by B, or indirectly when A holds a resource required by another action which directly or indirectly blocks B. To detect this we shall introduce the concept of a

BLOCKING GRAPH, which is maintained by the system and consists of directed arcs indicating which action is blocking which others. Before adding a new arc to the graph when a request is blocked the graph is checked to see whether the addition of the arc would cause a cycle to occur in it. If so a deadlock has been detected and appropriate recovery action must be taken. Fig. 3.4 shows a set of actions and their associated blocking graph at various stages in their execution. This shows that the blocking graph exhibits the following properties:-

1. No action is represented more than once in the graph,

2. No action can block itself or an action nested within it,

3. No action can be blocked by more than one action,

4. The blocking graph can consist of several disjoint trees,

5. The number of directed arcs in a tree is always one less than the number of actions in it, unless a deadlock occurs in which case it equals the number of actions.

This last property could be used to provide a means of detecting deadlocks, however this can be done in other ways as we shall see in section 3.9. Similar schemes can be devised for handling multiple requests, for example Lomet's (Lom 76b) but are considerably more complicated.

```
COMMON Q,R,S,T,U,V,W :....

ACTION A;  ACTION B;  ACTION C;  ACTION D; -- BLOCKING GRAPH

   Q:=R;        .        S:=..;     T:=..;

     .        R:=..;     U:=..;       .       --   A→B

     .          |          .        T:=U..;--   A→B  C→D

   W:=..;       |          .          |       --   A→B  C→D

     .          |        U:=W..;      |       --   A→B  C→D

END;            .          .          |       --   C→D

              END;       END;         .

                                    END;
```

where | indicates that a process is waiting.

Fig 3.4

The second deadlock that can occur in recoverable atomic action systems only arises where several lock modes are in use and actions are allowed to convert the locks they hold from one mode to another more binding mode. For example, where exclusive and shared read locks are supported, a deadlock will arise when two or more processes that have locked a resource in shared read mode wish to convert this lock to an exclusive one. Only ONE process can be allowed to do this so the system must resolve the situation. Two possible ways in which this can be done both rely on the fact that lock requests must be processed atomically. Firstly preemption of resources can be introduced meaning that the first conversion request received will be granted and cause all the other actions involved to be stopped, effectively preventing the clashing requests from being made. However if priorities are in use preemption may not be permitted so the system must also incorporate the second method which is to make all conversion requests after the first encountered (which may not have yet been granted) illegal, causing the actions making them to be recovered, which may allow the first conversion to be processed. Of course, if priorities are supported, the first request may be preempted by a later one causing the waiting request to become invalid. The management of the recovery action for this and the previous deadlock will be discussed in the next section.

3.7.3 Deadlock recovery management -

In the recoverable atomic action system the deadlocks we have just described can only occur between two actions, because of the atomicity of requests, and are resolved by backing out one of the actions. This will release the resource being contended for and allow the other action to proceed. However the question of which action to back out must be given very careful consideration, the aim at all times being to maximise throughput and minimise system disruption.

Where actions have priorities attached the system will obviously back out the action with the lowest priority, guaranteeing (barring the incidence of program errors) that the action of highest priority will be executed without ever being backed out, though actions of lower priority could have been wound back several times. However for actions of equal priority (or where there are no priorities) some measure of the system disruption caused by backing out each of the actions involved is needed, the least disruptive atomic action being wound back.

Several possible criteria present themselves, though some may be viewed in different ways and the inherent non-determinism of the system makes finding a perfect, general solution impossible. Let us consider seven of the possibilities :

1. An action which is near termination should not be wound back in favour of one which has just started execution - a similar idea has been suggested in (Cof 71) and the reasoning behind it is obvious. However actually putting it into practice is hard because there is no surefire way of gauging what proportion of an action has been executed. Method 4, below, may sometimes provide a means of doing this, because an action accumulates locks as its execution proceeds, but the success of this is highly dependent on the way in which an action uses shared resources - some actions may only acquire resources very near their end and others may acquire all the resources they need when they begin. Method 7 would provide a much better indicator, but requires that deadlines are implemented,

2. The action which is blocking the larger number of other actions should be backed out - this would be done in order to increase the number of active processes in the system and, thereby, hopefully, the throughput. The difficulty

here is that the blocking graph does not indicate which resources are being requested by the blocked actions and if they are all in contention for the same one nothing has been gained. Moreover increasing the number of active processes in the system increases the number of actions which may deadlock, thus causing even more disruption in the system,

3. The action which is blocking the lesser number of other actions should be backed out - this would ensure that the action which is causing the greater bottleneck in the system would be allowed to proceed bringing it nearer to termination and its subsequent disappearance from the system. This presents the opposite view to method 2,

4. The action which holds the lesser number of locks (possibly including the locks held by its enclosing actions) should be backed out - the reasoning being either similar to that of method 1 or that of method 3,

5. The action which was blocked when the deadlock was discovered should be wound back - this is based on the idea that if the action which is progresssing is left alone it will be brought nearer to its termination. The method also has the advantage of being simple to implement as it does not require extra information to be accumulated to aid the decision process. It should also have a fairly consistent success rate which is unaffected by the mix of processes in

the system, a characteristic not shared by any of the above.

6. Where deadlines are in force, the action which has the longest time left till its deadline expires should be wound back - the reason behind this is obvious, and should prove fairly successful because the relationship between the length of a deadline and priority is very close (the shorter the time, the higher the priority), as is shown by the discussion in (Lie 80).

7. Again, where deadlines are being used, the length of time left in a deadline be taken as a measure of how near completion an action is, winding back the one which has progressed the least - this is another realisation of method 1 and, to be accurate, the comparison must be based on the proportion of the whole deadline period which is left to be executed rather than on the times themselves as in method 6. This should prove very effective, but of course deadlines may not necessarily have been used or even be supported.

Of these methods only number five can be guaranteed to function in every case. With all the others there is the possibility that the two quantities being compared are equal, in which case a decision cannot be made. For this reason method five was chosen for the experimental system described in section 3.9.2, and it was through its use that the problem described in the next section was discovered.

Another question which must be considered regarding the management of deadlock recovery is whether or not a deadlock is regarded as an error, that is, whether or not the next alternate of a recovery block should be executed when a deadlock occurs in an alternate and it is wound back. The answer must of course be no, because the deadlock was in effect caused by the supporting system scheduling the concurrent processes incorrectly and hence the error lies in the interpreter level rather than the user level. What must be done instead is that the backed out action should be retried. Note that the deadlock which caused recovery to take place cannot take place in the same way, because the two resources being contended for are now held by a single action.

3.7.4 A possible infinite loop in the deadlock recovery mechanism -

Fig. 3.5. shows part of a program involving three processes of equal priority each trying to use some common variables. Under certain timing conditions, when supported by the system described above using method five of the previous section to break deadlocks, a race condition arises where actions A1, A2 and A3 are repeatedly backed out and no progress is made. Programs exhibiting similar characteristics could also be constructed for the other methods we have described, except for those involving deadlines, where conditions are inherently unrepeatable. Fig. 3.6a shows an execution sequence which will cause the race to occur, and though the probability of such a sequence happening compared with other possible execution flows is low, a solution must be found, as, inevitably, the sequence will arise at some time.

```
COMMON A,B,C,D,E,F : INT ;

ACTION A1;        ACTION A2;        ACTION A3;
   .                 .                 .
   .                 .                 .
   P(A,B,C,D);       R(E,D,C,F);       S(F,D,B,A);
   .                 .                 .
   .                 .                 .
END;              END;              END;
```

Fig. 3.5

The main characteristic of this loop is that the actions are backed out in the same sequence every time. That is A1, then A2 then A3 and so on, so what is needed, therefore, is some way of breaking this ordering which will cause the loop to be broken. This is achieved by giving each action a priority, if they do not have them already, and then incrementing the priority of the action which is not backed out when a deadlock occurs. The effect of this being that the actions will now have different priorities so future deadlocks will be broken on this basis rather than any other. Fig. 3.6b shows how the execution flow in fig. 3.6a is affected by the use of this algorithm. Incrementing priorities also has the advantage that at least one process will pass through the system without ever being wound back.

A1	A2	A3	BLOCKING GRAPH
LOCK A	LOCK E	LOCK F	
LOCK B	LOCK D	LOCK E	$A2 \rightarrow A3$
LOCK C	LOCK C	|	$A1 \rightarrow A2 \rightarrow A3$
LOCK D	|	|	$\begin{tikzcd} A1 \arrow[r] & A2 \arrow[r] \arrow[l, bend right] & A3 \end{tikzcd}$
WIND BACK	LOCK F	|	$\begin{tikzcd} A2 \arrow[r] & A3 \arrow[l, bend right] \end{tikzcd}$
LOCK A	WIND BACK	LOCK B	
LOCK B	LOCK E		$\begin{tikzcd} A3 \arrow[r] \arrow[rr, bend left] & A2 & A1 \end{tikzcd}$
|	|	LOCK A	$\begin{tikzcd} A3 \arrow[r] \arrow[rr, bend left] & A2 & A1 \arrow[ll, bend left] \end{tikzcd}$
|	|	WIND BACK	
LOCK C	LOCK D	LOCK F	
	LOCK C	LOCK E	$A1 \rightarrow A2 \rightarrow A3$

etc.

(a)

A1	A2	A3	BLOCKING GRAPH
LOCK A	LOCK E	LOCK F	
LOCK B	LOCK D	LOCK E	$A2 \rightarrow A3$
LOCK C	LOCK C	|	$A1 \rightarrow A2 \rightarrow A3$
LOCK D	|	|	$\begin{tikzcd} A1 \arrow[r] & A2 \arrow[r] \arrow[l, bend right] & A3 \end{tikzcd}$
WIND BACK	LOCK F	|	$A2 \rightarrow A3$
LOCK A	END	WIND BACK	

only two processes remain, and hence will terminate.

(b)

Fig. 3.6

3.8 Synchronisation with external events

3.8.1 The AWAIT statement -

The provision of a facility allowing synchronisation with external events requires that a process must be able to delay its progress until some condition involving shared resources becomes true. This is normally implemented by some form of "busy waiting" where the synchronisation condition is repeatedly evaluated until it becomes true. However, as this requires the shared resources to be locked for the duration of the evaluation and then unlocked, if the result is false, to enable other processes to set up the desired state, busy waiting cannot be implemented inside an atomic action. The programmer must therefore be provided with an operation which will allow him to specify synchronisation without violating the atomicity of his process. Lomet (Lom 76a) has introduced the AWAIT statement for this purpose, with which the programmer specifies the condition he requires to be true and the interpreter level delays his process until the condition is satisfied. Execution of the process is then allowed to proceed, the resources involved in the condition having been locked. Best (Bes 79) has raised some doubt about whether atomicity can be implemented where AWAIT statements are used inside atomic actions, basing his comments on an analysis using occurrence graphs. He suggests that the interaction between processes implied by the use of synchronisation violates the criteria for atomicity. However from the programmer's point of view this is not so, because his program does not "see" the processes which make the synchronisation condition become true, even though a theoretical analysis of it shows that interaction has taken place. We shall therefore proceed under the assumption that the use of the AWAIT statement is valid, though future work may show this to be false.

In order to simplify the implementation of the AWAIT statement Lomet has suggested that the conditions attached to them are built up only of what he calls <u>SYNCHRONISING VARIABLES</u>. These are in essence shared booleans which can be set to true to signal that some event has taken place. This helps the system because it then knows which variables can occur in AWAIT statements, allowing them to be provided in a way that makes the process of waiting more efficient. However even if general conditions are allowed the interpreter can minimise the number of times the synchronising condition has to be evaluated because :-

1. The condition cannot be evaluated until the resources involved are accessible to the action executing the AWAIT, so the system can use its own locking information to determine when it is worth attempting an evaluation,

2. If the condition evaluates to false it need not be evaluated again until some other action has used one of the resources involved in it; when that action releases the resource then the condition may be re-evaluated.

There are two error conditions involving the AWAIT statement which are worth mentioning here. The first is when the synchronising condition will never become true and we have seen that the programmer must provide a deadline to overcome this type of error. However the second error can be detected by the interpreter and occurs when the programmer uses a resource that has already been locked by his process as part of a synchronising condition. If the value of this resource does not affect the result of evaluation of the condition there is no error, however if the state of the resource is such that it renders the condition false an error must be raised. This is because the resource state will never be changed as it is locked by the waiting process, so the AWAIT would never terminate. The interpreter level can detect this situation fairly easily by examining its lock information, however we shall see in chapter four that there are circumstances where this type of condition is not erroneous and is in fact very useful.

3.8.2 <u>Evaluation of synchronisation conditions</u> -

Apart from the error condition described above, there are other difficulties with the AWAIT statement, as described by Lomet. The first concerns the order of evaluation of expressions. Obviously the condition "A & B" must be fully evaluated to be true but "A v B" need not, provided A is true. However, if the condition were not completely evaluated, B would not be locked after the AWAIT statement, and, though this is not a source of error (if the user examines B it will be locked and if he makes an assumption about B's value then his program is wrong), it would seem to be against the spirit of atomic actions. We shall therefore state that AWAIT conditions are fully evaluated.

```
AWAIT A OR B THEN
BEGIN
   -- Both A and B will be locked when
   -- this statement is executed
END;
```

(a) a simple AWAIT statement

```
AWAIT
BEGIN
   WHEN A       -> BEGIN...END; -- A will be locked
   WHEN B OR C  -> BEGIN...END; -- B and C will be locked
   WHEN D AND E -> BEGIN...END; -- D and E will be locked
END;
-- at this point only those resources used in the
-- selected statement and its guard will be locked
```

(b) a guarded AWAIT statement

Fig. 3.7

The second difficulty with the AWAIT statement is that it does not support non-determinacy. That is, an action cannot detect one of a set of events without preventing other processes from detecting one of the others, and for this purpose we shall introduce the guarded AWAIT statement, in analogy to Dijkstra's guarded commands (Dij 75). An example is shown in fig. 3.7. Here the system delays execution of the action containing the statement, until ONE of the guard conditions becomes true and then causes the statement block connected with that guard to be executed. After the statement has terminated only those common resources used in the guard and its statement block will be locked, all the others used in the statement will be free. The question of which guard is selected if several become true at the same time will not be discussed here, as the general area of "fairness" in non-deterministic systems is still a topic of debate, there being a brief discussion of the topic in (Hoa 78).

3.9 Implementation of a system supporting uncooperative processes

3.9.1 General considerations -

Section 3.3 has described the facilities that a language interface to the system we have described should provide and we shall first look at ways in which this interface can be realised. At the simplest level, where the user is unaware of other processes and is unconcerned about error recovery, the best solution is to surround his program (whatever the language it is specified in) implicitly with an atomic action and associate a deadline with the action. This will ensure the safe use of all common resources, and will prevent infinite loops, though of course it will not guarantee the correctness of the program and its effects on the resources it uses. In fact this is in effect the solution adopted by all the typical small job, compile-and-go batch systems. such as WATFOR (Cre 78), where only one job runs at a time (hence it is atomic), common resources (input device and output device) are "locked" for the duration of the job and "released" if it fails, and a maximum execution time limit is set for each job to catch loops and improve turnaround. Such systems usually allow the user to control the deadline for his job, within certain defined limits, and also provide another feature which we have not considered previously. That is the ability to control the amount of "use" a program makes of a resource. Typically this is a limit on the number of pages of printed output generated or cards punched. We shall look at this facility in greater detail in section 5.4.4.

The more sophisticated user, who appreciates the complexity of the system, will wish to have an interface which allows him to use its full power. He will need a special language which provides recovery blocks, deadlines, atomic actions and the AWAIT statement and this may be constructed from the various language structures already existing for these facilities (Lom 76a, And 75, Ran 75, HoC). In section 3.10 we shall discuss the role that the language's compiler can have in increasing the efficiency of the system.

At the interpreter level consideration must be given to the requirement for unique naming of actions identified in section 3.5. Such names need only be unique for the existence of the actions they refer to, and may be re-used at any time afterwards. The same function is served in the simple recovery block scheme by the recovery level, which is also unique at a given time but is re-used, so the best general solution for uncooperative processes is to maintain a record of the depth of nesting of atomic actions and generate identifiers by combining this value with some representation of the identity of the process in which the action occurs (but see secton 4.3.1).

The next item to be considered must be the granularity of locking/cacheing of the system, and for the purposes of this discussion we shall assume that they are the same. There are two trade offs which must be examined with respect to this, the first being between concurrency and program size. This is because the smaller the unit of locking for any structural resource the greater the number of processes which can use its parts in parallel, but, as each unit requires an action identifier field with it, the greater the amount of space required to store lock data. As an example, consider a system capable of supporting fifteen tasks each with a maximum limit of fifteen nested actions. This would require eight bits to represent all the possible unique identifiers (assuming that all zeros indicate the unlocked state), and so, taking a byte addressable main store as our resource, would require twice as much store as was visible to the user to support locking at the level of the byte. Of course, with the current trend in storage prices this may not be unacceptable, especially where very high reliability is needed and special purpose hardware is being built, but in an interpretive system based on existing hardware limits on address space could make such a store size impossible.

The second trade off is between concurrency and frequency of deadlock, and has been discussed in section 2.1.2. The decision taken must be based, firstly on the kind of processes to be run on the system (in some cases deadlocks may never occur no matter what the granularity of locking) and secondly on the cost of backing out an action which will occur every time a deadlock arises. In the general case it would seem better to have a unit of locking larger than the unit of addressing, though not so large as to prevent any concurrency at all.

Section 3.7.3 has already discussed various criteria to be used to determine which atomic action should be backed out when a deadlock occurs between actions of equal priority. Consideration of this indicates that a combination of method five with either method six or seven would be best for a general system - the tests on deadlines being carried out first. However special cases may allow special solutions, the aim always being to minimise recovery activity. It must also be remembered that the loop described in section 3.7.4 must be prevented.

The final topic which we must look at is the way in which the parallelism seen by the user is implemented at the interpreter level. There are two options. Firstly each distinct process at the user level could be implemented by a distinct process at the interpreter level, all the processes having access to a common store, or secondly a sequential interpreter could multi-program the processes at the user level. Both schemes have their advantages and disadvantages. In the first case scheduling of user processes is not a concern of the interpreter as this will be handled by the level providing it with parallelism, however the interpreter level must ensure the atomicity of its operations on the common store. For the second case the opposite is true - atomicity is guaranteed as only one user process is ever active at a given time, but a scheduling algorithm must be provided for the processes. Ultimately the choice of method is dependent on the hardware and software that will underly the interpreter. If multiple processors sharing common store are available then they will be used, otherwise a sequential interpreter is more likely, unless the levels below it provide adequate support for parallelism. (Note here that uncooperative processes cannot be implemented using processes which are themselves uncooperative as the blocking graph needs to be accessible to all processes). The major advantage that the use of multiple processors has is that when an action is backed out because of a deadlock the processor time that has been spent attempting to execute has not been wasted, because with any other scheme the same amount of time would have been spent waiting to acquire the resources in question.

The next section will briefly describe a trial implementation of some of the ideas we have been discussing in this chapter and report on the problems encountered.

3.9.2 Implementation of a test system -

The system to be described below was implemented on the IBM 370/168 of the University of Newcastle upon Tyne, running under the MTS operating system. Its purpose was to determine whether or not a system of the type described above was feasible and was therefore not implemented with considerations of efficiency in mind.

3.9.2.1 The language interface -

The language interface to the system was provided using a modified version of the concurrent PASCAL compiler designed for the SOLO operating system (BrH 76). The compiler, due to Hartmann (Har 77), consisted of seven passes, and produced code designed to be run on the interpreter that supports the SOLO system. Several new types and statements were added to the language, whilst others, for example any using the type REAL, were removed from it completely. The compiler was also converted to assume a basic word length of thirty-two bits rather than the sixteen that it was set up with.

The additions to the language were as follows :-

1. Atomic actions - These were provided at the procedure level rather than by allowing any statement block to be made atomic. The keywords ACTION and AGENT being substituted for PROCEDURE and FUNCTION to indicate that atomicity was required. The body of an ACTION/AGENT could be of two forms, the first provided the user with recovery blocks, its syntax being :

```
ENSURE <acceptance test> BY
<statement block>
{ ELSE BY <statement block> };
```

   The compiler adding a default call to ERROR after the last alternate. The second form is a simple BEGIN...END block, which is translated into this recovery block:

```
                ENSURE TRUE BY
                BEGIN
                   .
                   .
                END
                ELSE ERROR;
```

Fig. 3.8 shows the form of code generated for a recovery block consisting of a primary and a secondary alternate.

```
     GOTO ALT1                 -- enter primary
ATST:    .                     -- acceptance test
         .
     IF TRUE THEN GOTO EXIT    -- test was successful
     GOTO NEXT@                -- NEXT points to alternate
ALT1:NEXT:=@ALT2               -- set up NEXT for secondary
         .
         .                     -- body of primary
         .
     GOTO ATST                 -- perform acceptance test
ALT2:NEXT:=@ALT3
         .
         .                     -- body of secondary
         .
     GOTO ATST                 -- perform acceptance test
ALT3:ERROR                     -- failure of recovery block
```

Fig. 3.8

2. The ERROR statement - when executed this statement caused an error to be signalled and whatever recovery action was possible to be initiated,

3. The RETRY statement - this statement could only occur as an alternate of a recovery block (not the primary) and caused the preceding alternate to be executed again.

4. The ASSERT statement - the syntax of this statement is

```
                ASSERT <condition>;
```

and is equivalent to the statement

```
IF NOT <condition> THEN ERROR;
```

5. The PRIOR operation - this facility, also provided in the system described in (And 76), is only allowed in the body of an acceptance test and permits the user to access the original value of a variable which has been stored in the cache. If the variable was not cached during the action which the acceptance test was attached to its current value was returned. The preferred syntax for this operation would have been of the form "V. PRIOR" making the prior operation an attribute of every variable V. However due to restrictions in the compiler this had to be implemented as "PRIOR V."

6. The basic type SYNCHRONISING (or SYNC) - variables of this type were exactly equivalent to booleans but had the extra property of being allowed to appear in AWAIT statements.

7. The AWAIT statement - the version that was implemented was highly restricted and constrained the user to waiting in a single variable of type SYNC. The syntax of the statement was

```
AWAIT <SYNC variable id>;
```

8. The basic type ALARM - objects of type ALARM provided the

user with a form of deadline and had two operations associated with them - A.enable(time) which "activated" A and caused an error to be raised after "time" clock ticks had passed, and, A.disable which stopped A and prevented it from raising an error.

Also if a user left a block which contained the declaration of an ALARM variable, which was still enabled, a warning was produced and the alarm disabled and deleted.

9. Shared atomic actions - these will be discussed in chapter four.

These facilities provided enough power for some experiments in the use of recoverable atomic actions to be carried out and we shall review them in section 3.9.2.3.

3.9.2.2 Interpreter structure -

The interpreter for the language was, as was indicated above, based on the inerpreter provided for use with the SOLO operating system on the DEC PDP-11. Much of the interpreter was machine dependent and several parts of it (such as input/output handlers) were completely ignored. However the biggest difficulty encountered in mounting the original interpreter on MTS was the need to convert from sixteen bit words to thirty two bit words. The interpreter was a sequential program using a "round-robin" scheduling algorithm, executing one instruction from each process in the run queue at a time. The scheduling in the original SOLO version involved three priority levels (processes in MONITOR'S, processes doing I/O and others) and time slicing, however as the two highest priority levels were irrelevant in the new system, and as the user could not attach priorities to processes when they were specified, this method was abandoned. The lack of priority structuring also constrained the system to be non-preemptive and for system queues to use a FIFO discipline. This had the advantage that no process could ever be kept waiting for a resource indefinitely, other than as the result of user error. The time slicing of the SOLO system was omitted (or at least reduced to one clock "tick" per process) as this enabled multi-processors to be modelled more closely. This had the effect of increasing the interaction between the processes, thereby exercising the systems capabilities more fully.

Only exclusive locking was supported and the granularity of locking/cacheing was chosen as one word. This was done even though the system allowed addressing to the byte level using the type CHAR, because this reduced space requirements and as it was felt that interactions at the level of adjacent bytes which would remain independent were unlikely. Each process had its own data area, including a cache whose size was set to be one quarter of the stack area allocated for the process. The data space for the initial process, which becomes the common area for all other processes, does not contain a cache as the initial process was intended simply to

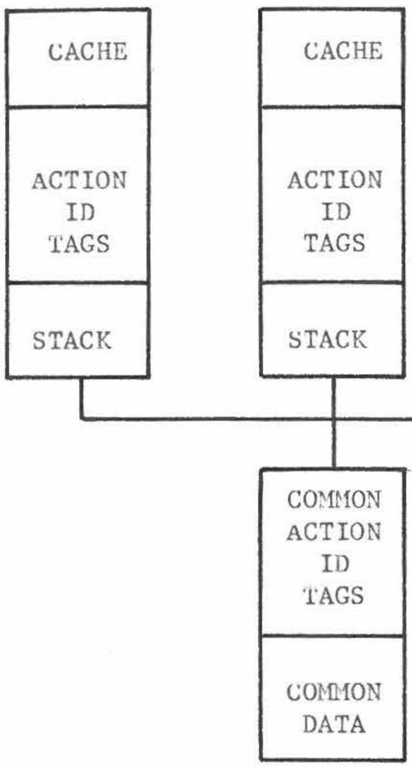


Fig. 3.9

spawn the other processes and stop, its termination activating the rest of the system. Fig. 3.9 shows the data areas allocated and fig. 3.10 shows the structure of a cache and its entries.

The scheme adopted for naming actions was less structured than the one suggested in section 3.9.1 because of the need to support shared atomic actions (see below). It consisted of restricting the user to sixty-three actions at any one time throughout his system, each one having a data area allocated for it. When a new action was required the array of data areas was searched until an unused one was found and its index was used as the action identifier. Fig. 3.11 shows the structure of an atomic action data area. One useful simplification used throughout the system was to use process identifiers rather than action identifiers when handling

```
TYPE BARRIER IS RECORD
   LAST_BARRIER : @BARRIER; -- enclosing action's barrier
   NEXT_ALTRNTE : LABEL;    -- address of next alternate
   ACCEPT_TEST  : LABEL;    --    "    "  acceptance test
   CURRENT_ALT  : LABEL;    --    "    "  current alternate
                            -- (this allows retries)
   OLD_SP       : ADDRESS   -- saved value of stack pointer
   LAST_ID      : ACTION_ID -- name of enclosing action
END;

TYPE DATA_RECORD IS
   VALUE  : DATA;           -- saved value
   OLD_ID : ACTION_ID;      -- saved action id tag
   WHERE  : @DATA_VALUE;    -- pointer to cached resource
END;

CACHE : ARRAY [ CACHE_SIZE ] OF RECORD
           CASE KIND : (BARRIER_ENTRY,DATA_ENTRY)
      WHEN BARRIER_ENTRY -> B : BARRIER;
      WHEN DATA_ENTRY    -> D : DATA_RECORD;
           END
        END;
```

Fig. 3.10

interactions, thus an action was not seen as preventing another action from executing, but another process. This had two advantages, the first being that, as the number of processes in a system was normally considerably less than the number of actions, a set (in the PASCAL sense) of process identifiers could be represented in a much smaller space. The second advantage was that the process identifier not only stood for its current atomic action, but also for all the actions enclosing it. This makes the implementation of the blocking graph and subsequent deadlock detection very much easier. The algorithm used was as follows :-

1. Each atomic action had associated with it a queue of processes whose progress was blocked by the action in question having possession of a variable they required. The identity of all these processes was recorded in a set

```
TYPE ACTION_DATA IS RECORD
   NAME       : ACTION_ID;           -- name of this action
   MEMBERS    : SET OF PROCESS_ID; -- identity of processes
                                     -- that are in the action
   BLOCKING   : SET OF PROCESS_ID; -- identity of processes
                                     -- that are blocked by
                                     -- this action
   WAITING    : QUEUE OF PROCESS;  -- processes waiting for
                                     -- resources held by
                                     -- this action
   SHARED     : BOOLEAN;             -- TRUE if this a shared
                                     -- atomic action
   PRIORITY   : (MIN_PRI..MAX_PRI);-- priority of action
END;
```

Fig. 3.11

variable, which then effectively represented the directed arcs in the blocking graph that linked this action to the ones it was blocking.

2. When a new action was created its blocking set was initialised with the value of that of its enclosing action's blocking set, for it too was now blocking those processes.

3. When a request was made for a common variable and was denied, the action (process) making the request had to join the queue of actions (processes) waiting for the action holding the variable to terminate, and have its identity recorded in the blocking set. However before this could be done safely, the intersection of the blocking set of the action whose request was denied and the set of processes that are members of the action blocking the request must be taken. If the result of this calculation is not the empty set a deadlock would arise if the requesting action joined the queue, so recovery action must be taken.

4. When a process was allowed to join a wait queue its identity was not only added to the blocking set of the action immediately stopping it, but also to the blocking sets of all its enclosing actions which are now effectively stopping it as well.

5. When an action terminates its new blocking set was passed back to its enclosing action and their wait queues amalgamated, unless the action was the outermost, in which case the wait queue can be released.

The deadlock detection part of this algorithm can be implemented very efficiently on most computers. For its functioning on a system supporting a maximum of 'p' processes and 'a' actions it only requires '2pa' bits of data in total, there being two sets of size 'p' for each of the actions, where each bit in a set is taken to represent a process. The test for a deadlock can be made by performing a logical 'and' between these sets, and testing for a zero result (no deadlock). These operations are usually two of the fastest in any machine's instruction set so this part of an interpretive system can be made very small and fast.

When a deadlock is detected by this method one of the actions involved has to be backed out. Which it was was decided by first comparing the priority fields in the action data areas, and backing out the action with the lowest priority. If the priorities were equal, the halted action (note that an action identifier could always be determined from a process identifier) was backed out and the priority of the other action was incremented. If the priority scheme had not been used the race condition described in section 3.7.4 could have occurred, and in fact the program shown in fig. 3.5 was used to create this condition when the priority scheme was disabled.

The method used to implement the AWAIT statement was very simple, and could be much improved. It simply consisted of maintaining a queue of processes that were waiting for synchronising variables to come true, and whenever an action terminated, possibly having altered the state of a synchronising variable, all the processes on the queue were re-activated to retry their AWAIT statements. The reason for such an inefficient implementation was that the use of a synchronising variable within an action was accomplished using the basic operations of the system. This meant that any alterations to the variable were not explicitly detectable, preventing the use of special queue's to eliminate busy waiting. However if these alterations had taken place through the use of instructions specific to that purpose, such a scheme could have been implemented, but limitations in the basic compiler prevented the generation of special instructions, so the above method was adopted.

We have now outlined the features of the support for recoverable atomic actions, the recovery cache mechanism being implemented exactly as described in section 3.5. All the basic operations of the system were modified to include cacheing of operands, though their function was not altered in anyway. The only other feature added to the interpreter was the use of code checksums as described in section 2.3.4. These were built up during execution and compared with a value computed at compile time whenever a transfer of control took place. If the values were not identical an error was raised and recovery initiated.

In the next section we shall describe the experience that was had with the whole system, and evaluate its usefulness.

3.9.2.3 Experience with the system -

The system just described was tested with a wide range of simple examples and in all cases was found to perform correctly. Testing was concentrated on the use of nested recovery blocks and on the deadlock detection/recovery mechanism, and one of the results of this was the discovery of the race condition discussed in section 3.7.4. However even though the system effectively demonstrated the feasibility of using the mechanisms developed in this chapter, it was not possible to use it to measure the overheads involved in their support. There were several reasons for this.

Firstly the language interface proved to be inconvenient and difficult to use for anything other than the simple test programs mentioned above. This was not due to the features added to the language, but was caused by the scope rules built into the Concurrent PASCAL language which was used as a starting point. These restricted procedures to accessing either local variables or variables declared in their immediately enclosing blocks. This was done to simplify the implementation of MONITOR's, whose correct functioning depends on just such a limited scope, and means that any accessing done outside these levels must use the parameter passing mechanism. However to use atomic actions effectively access of this type must be done frequently so very large numbers of parameters were required if operations of any complexity were attempted. Concurrent PASCAL is also not designed to allow the use of shared variables, again because it is a MONITOR based language, and restrictions against their use were built in to the compiler. It proved very difficult to eliminate all these controls, because of the multi-pass nature of the compiler and because no documentation describing the compiler was available when the system was being developed. The result of this was that the number of shared variables available to the programmer was severely limited, and the development of any "real" programs was prevented.

The interpreter also gave rise to several problems. The main difficulty, as regards performance measurement, was the lack of a suitable yardstick with which it could be compared. The only evaluation of performance that could be made was subjective, and, from a user's point of view the response obtained from the system was perfectly acceptable. Cache size measurements were hampered because the restrictions imposed by the compiler meant that a suitable cross-section of program types could not be tested. Nevertheless it was noticeable that allocated cache sizes, based on process stack space, were small (typically less than a hundred words) and cache space was never exhausted during any of the test runs.

One piece of information which did arise from the investigation was that the use of code checksums had a significant affect on code size. Every control transfer (jumps and calls) had an extra memory word with it to hold the compiler generated checksum for the straight line code sequence preceding, and this was found to increase code size by, on average, ten percent. The reason for this was the large number of control transfers that occurred in programs written for the system. These were generated because of the procedural mechanism used to invoke atomic actions and because of the control structure needed to support recovery blocks. Fig. 3.8 shows that where a statement is replaced by a recovery block consisting of a primary and secondary alternate an additional six control transfers are introduced, not including those that may be contained in the bodies of the alternates. This shows that the extra security provided by the code verification may introduce an unacceptable storage overhead especially considering that program size is considerably increased by the presence of recovery blocks. On machines with a large address space this may not present a problem, but on many small computers such overheads could be critical.

3.10 Efficiency of systems supporting recoverable atomic actions

As we have seen the test system did not provide much information by which the efficiency of recoverable atomic actions could be judged. However consideration of the various areas where performance may be affected can allow us to judge how efficient such a system may be. The recovery cache mechanism introduces overheads in terms of space and time. Section 3.9.1 has already discussed the storage overhead due to the need for atomic action identifier fields with every resource and indicated its relationship to the chosen granularity of locking. However there is also the overhead introduced by the cache that each process has. This topic has been discussed in several previous papers (Wye 73, Ran 75, Ver 77) and the conclusion has always been that the storage requirements for cacheing would not be excessive, and experience with the system just described has not contradicted this. Of course, these two overheads need not affect the address space available to programs as the storage can be provided in seperate memory. However, as we have seen, the use of fault-tolerant programming techniques will increase program size. The exact increase is difficult to estimate, as it depends on the number of alternates used and the algorithms contained in them, but it is obvious that program size could be doubled if every section of code was provided with an alternate, and this could present problems.

Execution overheads fall into two classes - those associated with the evaluation of acceptance tests, and those incurred by every instruction that loads or stores data. No data exists for the first class, though Kim (Kim 76) has considered it important enough to produce a design for a system that will execute acceptance tests in parallel with their recovery blocks. However as the specification of acceptance tests is still an area where much research remains to be done very little can be said about it.

Until recently very little information about the second class of delay was available either, for all the implementations of the recovery cache (And 76, Ver 77, Shr 79b) were interpretive and could not take advantage of parallelism to increase their efficiency. Shrivastara (Shr 79a) estimates that performance in his systems was degraded by eleven per cent when only assignments were being cached, but points out that his system was purely for experimental purposes and he makes no attempt to estimate the improvement that hardware support would provide. However such a hardware system has been implemented by Lee et al (Lee 79) which can be added to a PDP-11 UNIBUS to provide cache support. Their estimate, based on an analysis of PDP-11 bus activity is that performance would be degraded by eight per cent when their device was in operation but that this could be improved to four per cent if destructive read out were used in the memory unit. Of course to support recoverable atomic actions operations that "read" common resources must also be cached which could add to the overhead. However such cacheing could be performed completely in parallel with the operation performing the read, as the value obtained will not be affected, so this overhead could be eliminated. The problem with this is that access to the resource may be denied in which case the operation must not proceed. Therefore the lock check must be performed before continuing adding some delay to each shared read, though if access is granted the actual cacheing, should it be necessary, can take place concurrently with the operation. No reliable figures for the number of "read" references made to shared resources seem to be available, so it is difficult to estimate how large an overhead the checking will be. Wyeth (Wye 73) has analysed the references in a set of sequential programs and his figures show that reads occur three times as frequently as writes. This suggests that the number of read accesses made to shared resources will be fairly high, because the simplified interface to resources provided by recoverable atomic actions will encourage programmers to use them as they would private resources. Overheads may therefore be quite high, and ultimately there is a trade off between these inefficiencies and the simplicity and the recoverability provided by the system. The general consensus of opinion would seem to be that where reliability is required such overheads are acceptable.

The other area where questions of performance can be raised concerns the dynamic locking of resources and the deadlocks that may arise from it. The point is that processor time is wasted in carrying out computations which are subsequently rolled back because they reach a deadlock, when the use of a static allocation policy would avoid this. There are several answers to this objection, the first being that it is only true where parallelism is implemented using multi-programming techniques. For if each process were running on its own processor the time spent carrying out abortive computations would otherwise have been idle time spent waiting to acquire resources. The use of dynamic locking can also eliminate such waits altogether, because static deadlock detection does not take into account the pattern in which resources are used by processes and will often indicate the presence of a deadlock which would not actually arise.

Another argument for dynamic resource locking is that it is the only way to support run time resource address computation and synchronisation using shared resources. So if these facilities are needed, deadlocks, and their associated recovery, must be accepted as a necessary evil. The counter argument that a preemption scheme such as the one described in (Cha 74) would provide the same support without the need to roll back computation is weak. For, in order that consistency be maintained, preemption can only take place at certain "safe points". Chamberlin et al organise this by constraining the user to making all resource requests within what they call a SIEZE block, in which no computation other than resource specification can take place. This restriction ensures that no dependencies on values are built up in a SIEZE block, so preemption of resources can take place whilst it is being processed. However when a SEIZE block terminates the process cannot be preempted and no other SEIZE block can be executed till all the resources acquired in the first block have been released together. If the user is not to be restricted in such a fashion then some way must be provided for the results of computations to be undone automatically and this can only be achieved by a system providing the kind of support that we have described in this chapter. The decision as to whether the advantages given by the use of dynamic allocation outway the disadvantages brought by the mechanisms needed to support it must depend on circumstances, but in many cases it is certainly the case that they will. It would of course be possible to use a compromise method which involved some dynamic and some static locking, for example dynamic locking of conditionally acquired resources, static for those known to be needed, though some modification to the cache algorithm would be required.

3.11 Conclusion

Section 3.3 listed the features that it was hoped a system incorporating the mechanisms described above would possess so in conclusion let use see whether these features are in fact provided by it.

The first requirement was that the only difference between common resources and private ones that the programmer saw was in their declaration. This implies that the location of a resource must supply the system with enough information to be able to determine the way in which it should be treated, and, as we have seen, the modified recovery cache mechanism makes this possible. The action identifier field associated with each resource contains enough information to indicate whether a resource is available or whether the action requesting it must wait. This field is located solely using the resource address. In fact "read" operations are the only ones which need to know whether a given address refers to a shared area, so that the amount of cacheing can be minimised. Another advantage of the mechanism from the user's point of view is that a resource which is private to a process can be shared by any sub-process it spawns without additonal overheads. This is because the position of its declaration in the structure of his program ensures that its address will be in a shared area when the sub-processes are in existance.

The second requirement was protection from the activities and errors of other users and the locking, recovery and deadlock detection mechanisms developed certainly fulfil this. They also meet the third requirement which is essentially that the user is protected from his own run-time errors, as does his ability to use recovery blocks if he so wishes.

A simple method of making existing programs uncooperative was the fourth requirement. Section 3.9.1 showed how this can be done by enclosing a program in an atomic action, though this will not work for a program composed of several uncooperative transactions. In the latter case additional control structures would have to be added to the program, which could in fact be difficult if it were written badly or in an unstructured language such as FORTRAN, so the system does not fully meet this requirement.

The last feature wanted was that the efficiency of the system should not be impaired by the facilities provided, and that it should be usable for real time applications. We have discussed the efficiency of the system that has been developed in the previous section and seen that some degradation of performance is unavoidable, though it would appear that this is not as severe as might be expected. However a full hardware realisation of the mechanisms would be needed to evaluate performance, especially if real-time support is envisaged. One important point to remember here is that, as far as software fault tolerance goes, the use of recovery blocks in some shape or form is the only technique developed so far that can be described as successful. This entails the provision of a recovery cache to support them, so programmers seeking reliability will have to have a cache in their system, whether it degrades peformance or not.

In general, then, we can see that by combining the basic operation of locking with the activity of collecting error recovery information, we have produced a system which enables the programmer to use shared resources safely without recourse to their explicit acquistion. Not only that, the mechanism involved is simple and could be incorporated into existing systems with little difficulty, though performance will be degraded. However as it stands the system will not allow processes to communicate with each other and to make use of resources and then release them immediately. The succeeding chapters will attempt to show how these facilities may be built upon the basic mechanisms described above.

4.0 Closely Cooperative Processes

4.1 Introduction

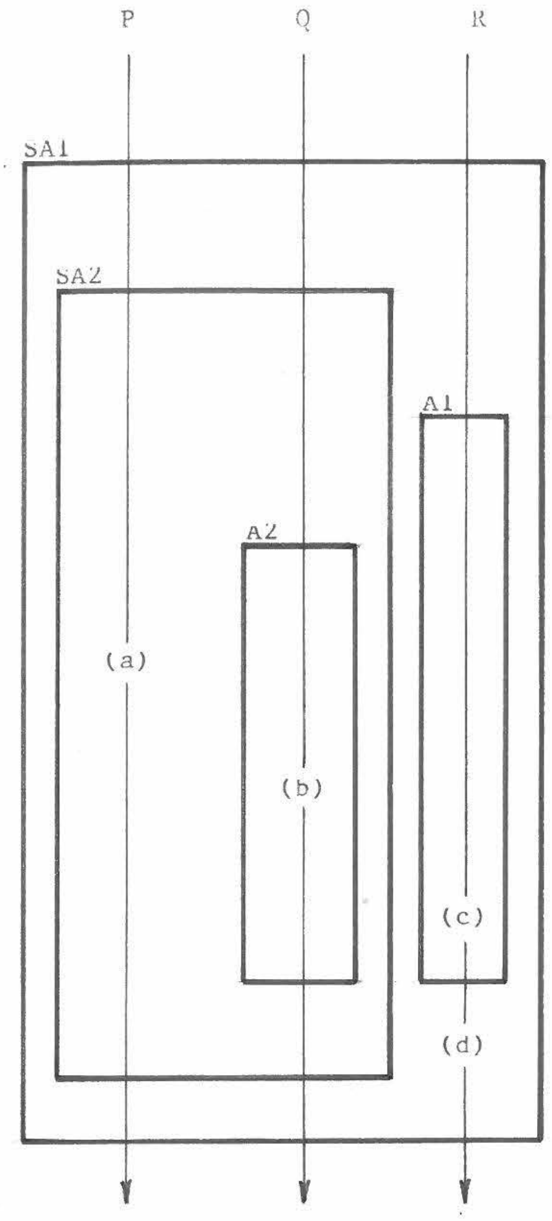
In this chapter we shall expand the concept of the atomic action to allow an action to contain several concurrently executed paths, rather than just one, creating what Lomet calls a SHARED ATOMIC ACTION. This structure possesses the important property that resources locked within a shared action, are usable by ALL the paths within the action, whilst their uses of them appear atomic to other processes not involved in the action. This means that the processes executing the paths of the action may communicate freely with each other by using these resources, and this makes them closely cooperative as defined in section 1.2. Any of the processes involved in a shared action may use a resource exclusively by carrying out their operations on it within a nested action, shared or unshared, but on the sub-action's termination the ownership of the resource reverts to the enclosing shared action and thus to all the processes involved in it. As it stands this structure is not recoverable because processes may leave the shared action at any time, thereby committing the results of their operations before other, possibly dependent, processes have terminated. To overcome this we shall apply the rules developed by Randell for "conversations" (Ran 75) and by Davis for "Spheres of Control" (Dav 73). Applied to shared atomic action, these are :-

1. No process may leave an action (that is commit its results)

until all the other processes involved in the action have succesfully executed their respective paths through the action and are ready to leave themselves (the processes are coupled).

2. If an error occurs in the execution of one of the paths through a shared action at a point where there is no nesting of actions within it, <u>all</u> the paths must be wound back.

These two rules make shared actions recoverable, and, because of the restrictions they impose, simplify the system needed to support such recoverability. The basic requirements for supporting shared actions are identical to those for unshared actions and this should allow us to use the modified recovery cache scheme to support them. However this cannot be achieved simply by allowing an action to have more than one member process, because the question of where resources are cached must be answered. In the system supporting unshared actions each process must have its own cache, and, as unshared actions can be nested within shared actions, each of the processes executing within an action must also have its own cache. If this is done no additional facilities are needed in the recovery cache mechanism. For when a process in a shared action first uses a resource which has not been used previously by any other member of the action, it will cache the resource and update its action identifier field to contain the name of the shared action. This means that all the other processes will "see" the resource as having already been locked should they come to use it, and so will not cache



Error at (a) - back out SA2, taking with it A2.

Error at (b) - back out A2, process P is unaffected.

Error at (c) - back out A1, processes P and Q unaffected.

Error at (d) - back out SA1, taking with it SA2.

Fig. 4.1

it. However this does not impair recoverability, because the second rule given above ensures that all caches are processed on an error so the resource will be restored correctly no matter how many of the processes involved have used it. Naturally this fact does not preclude special methods being applied to cacheing of resources within shared actions, but they would not bear the direct relationship with unshared actions that this one does. Section 4.3.1 will show other advantages that this scheme possesses.

The error handling mechanisms of the system must also be extended to cope with shared actions, as they must now be able to initiate recovery action in each of the member processes. Note that, as with deadlines, this may require several layers of nesting to be backed out if nested actions have been used. Fig. 4.1 shows a typical shared action structure, and indicates the ways in which recovery can take place within it. Later in this chapter we shall look at more specific points concerning the implementation of shared actions, but before that we must examine the kind of interface that is to be provided for the programmer when he wishes to use shared action.

4.2 Specification of shared atomic actions

4.2.1 Sub-processes and co-existing processes -

Because of the restrictions on the ways in which atomic actions may be nested it is not possible for a set of processes whose access rights are not identical to enter a shared atomic action together. In fact there are only two ways in which shared atomic actions can come into existence :-

1. By a single process forking into a set of sub-processes, which either consitute a shared action in their own right or EXPAND the action in which the fork operation occurs into a shared action.

2. By a set of processes within the same shared action forming a nested shared action (note that this also includes the case of a set of processes, none of which are members of any action).

Representation of the first case is very simple - any of the existing notations used for parallel statements being adequate, for example Dijkstra's PARBEGIN and PAREND (Dij 68a). If the user wishes to create a new action he need only prefix the statement with the attribute SHARED to indicate this. Omitting this will cause the membership of the action within which the statement occurs to be

```
ACTION OUTER;
BEGIN
   .
   .
   PARBEGIN ... | ... | ... END;
   .
   .
END;
```

(a) expansion of OUTER from unshared to shared

```
ACTION OUTER;
BEGIN
   .
   .
  SHARED A2 -- name sub action A2
   PARBEGIN ... | ... | ... END;
   .
   .
END;
```

(b) creation of a shared action inside an unshared action

Fig. 4.2

increased to include all the subprocesses, possibly turning an unshared action into a shared action. Fig. 4.2 shows an example of each case. There are two special points to make here. Firstly this is the case, referred to in section 1.5, where actions are created without user specified names, and secondly sub-process members of actions are allowed to terminate before the proper termination of an action, because their separate control streams rejoin into a single stream which must obey the termination rules.

The second case requires a different approach because of the need to bring disjointly specified processes together. The most structured method of doing this is to use a modified version of the SIMULA CLASS specification (Bir 73), where each procedure entry defines one of the execution paths within the shared action. In order to enter the action a process simply calls one of the procedures. However, before control is returned on its successful completion, all the other parallel paths through the action must also have been executed without error. Unfortunately the situation is more complex than this because the system must at least know how many processes are needed to form the action, in order to determine when all the paths have been processed or when the action may start, if synchronisation between processes before commencement is required. Three possible ways of overcoming this difficulty suggest themselves :-

1. The user must define the valid groupings of processes which can form shared actions, thus indicating memberships as well as the number of members,

2. Each process entering a shared action must do so through a different entry, and all entries must be used - then the number of entries equals the number of paths in the action,

3. The system records the identities of the processes that have entered the action and of those which have completed the entry they called, and when the two sets are identical the action is adjudged to have terminated.

All these solutions present difficulties of varying magnitude, either to the system or to the user. The third method is very simple and requires no extra information from the user, but the system cannot guarantee that when a process enters a shared action it will be sharing with the processes the user expects. This is because scheduling is essentially non-deterministic and even the number of processes in the action cannot be guaranteed. The only way to solve this is for the user to include his own code which will ensure that the correct processes come together, for example by including some synchronisation as the first statement of each entry.

The second method presents a similar problem in that, though the system knows how many processes are needed, it does not know their identities so cannot guarantee that the expected set of processes has come together. The system also has the problem of interpreting what is meant when the same entry is called at the same time by two different processes - is it an error, or is it two valid, seperate attempts to initiate the same action? If the latter, which process should be allowed into the action and which delayed? There is however an even worse problem because, through programmer error, the system can deadlock. This arises when one or more entry calls do not occur, hence the processes which are in the action will wait indefinitely for termination. Simple ommision of a call statement can cause this to happen, or incorrect ordering of calls to several actions shared by non-disjoint sets of processes which can lead to the situation where an action A, containing process P is waiting for a call from process Q which is in action B, which is itself waiting for a call from P. Unfortunately, because the system does not know the identity of the processes which a given action is waiting for the error will only become apparent when every process in the system becomes involved and all activity stops. In this case an error can be raised in the action (if any) enclosing the deadlocked actions and the process of recovery will break the deadlock. However, in the general case, such detection is not possible and only the use of deadlines can break the deadlocks that might arise.

The method whereby the user must declare which processes can come together to form a shared action still has the same problem, but because the system now knows which processes an action is waiting for it can detect the deadlock and initiate recovery. The easiest way to detect the deadlock is to build a blocking graph - a shared action, one of whose members is needed by another action, being said to block that action. When a cycle occurs in the graph denoting the relationship between actions, deadlock has been reached. However this does not solve all the problems, because an ommitted call may not cause a deadlock, and in that case will only be detected when the process which should have given the call attempts to terminate or leave the enclosing action if the shared action is nested. In the latter case recovery action can be taken, but in the former it is likely that commitment of the results of the incorrect process will have taken place (especially if it consists of a sequence of uncooperative transactions) and no recovery will be possible. A partial solution to this would be for a compiler to check that a call to the action was present in all the processes that make it up, but the presence of conditional calls makes this approach unreliable.

It would seem, therefore, that some other abstract structure, rather than the CLASS, must be used if complex patterns of action usage are to be supported. However the form it should take is not obvious, though the use of a type of path expression to indicate at compile time the correct sequence that actions should be executed in seems to be the most fruitful direction to take.

A final small point about shared actions, unrelated to the question of deadlocks, is that where the paths through an action are implemented as recovery blocks, all the blocks must have the same number of alternates. Any other arrangement would be meaningless and the check can easily be made at compile time.

4.2.2 Access to shared resources within shared actions -

When inside an unshared atomic action a process is fully shielded from interference from other processes, but inside a shared atomic action, though protected from processes that are not part of the action, there is no control over interaction with other member processes. This can obviously lead the programmer into difficulties. However it is this very lack of control that lends the concept of shared actions its power, so any methods devised to protect the user from erroneous interactions must not decrease the power available to him. The simplest way to ensure that operations on resources common to all processes in a shared action are performed correctly is to enclose them in a nested unshared atomic action. This is the approach adopted in the example described in section 4.4 and shown in fig. 4.3. A compiler could easily enforce that this took place by checking that every access to a shared resource took place within a unshared action. A similar check could be performed at run time if this compile time enforcement was not possible.

Another method that can be used is for the programmer to synchronise the activities of the processes within the action so that interference is eliminated. However this approach is highly error prone relying as it does on the correctness of the algorithms used and often on the programmer's assessment of the way in which the system will execute his processes. In fact the enforcement of atomicity, as above, in no way prevents the programmer from constructing synchronised systems and will eliminate interference. Where the synchronisation between the processes is correct the protection given by the use of unshared actions will be redundant but will mean that any recovery that has to take place will, in the first instance, only involve the unshared action rather than the whole action. This could represent a considerable improvement in performance, especially where deadlock recovery is concerned.

We can see therefore that by restricting the use of shared resources to the bodies of unshared actions the problem of interference can be eliminated. The power of shared actions is not reduced - communication and synchronisation between processes can still take place, and some of the recovery overhead involved with shared actions can be reduced. The user will have to add extra control structure to his program, but the advantages obtained far outweigh this disadvantage.

4.3 Implementation

4.3.1 General considerations -

Many of the comments made in section 3.9.1 concerning the implementation of unshared actions apply equally to shared actions. However certain areas need further elucidation.

Firstly there is the question of naming shared actions. In section 3.9.1 we suggested that names could be constructed for actions from the process identifier and recovery depth of the process involved but as there are several processes inside a shared action this is not posible (though note that the recovery depths of processes inside the same action, must be the same because of the nesting rules for actions). If the process identifier/recovery depth names are to be kept for unshared actions, the best solution is to assign each shared action an identifier which can be used in the same way as the process identifier, this has the advantage that nested shared actions can be referred to by the name of their outermost enclosing action and their depth of nesting, thus reducing the number of names required. However to implement this means either larger storage overheads or more restrictions on the user - consider the example given in section 3.9.1 of a system supporting a maximum of fifteen processes, allowing fifteen nested levels of recovery. Out of fifteen processes, seven outermost shared actions can be created and to provide room for these names in the action identifier field requires an extra bit, which must be obtained either by extending the field by one bit, or reducing the depth of nesting permitted to seven levels. This would give five bits for the process identifier giving thirty-two names, which will alllow expansion to twenty processes and the maximum ten shared actions possible with them. If more than seven nesting levels are required and eight bits is the maximum space available for identifier fields the number of processes allowed could be reduced to ten, giving five shared actions possible - whichever way is chosen depending on the programs that the system supports.

The problem with this sort of scheme, simple as it is to implement, is that circumstances may arise where a new name cannot be generated for an action even though many names are still unused. The solution is to generate names from a pool rather than to build them from the attributes of tasks. This, though still having an upper limit on the number of names imposed by the size of the action identifier field, allows for much greater individual variation in the structures and numbers of processes supported. This type of scheme was adopted in the implementation described in section 3.9.2 where only sixty-three names were available, and this limit was never exceeded though nesting was often deep.

Cacheing is another area where differences arise between shared and unshared actions, these being due to each process in a shared action having its own cache. The result of this is that when a resource is first used inside a shared action its recovery data is recorded in the cache of the process which uses it, after this all the other processes may use the resource but will not enter it into their caches, this of course being perfectly acceptable as ALL caches must be processed on termination or backing out of an action so recoverability is not impared. However in the case of a shared action created out of sub-processes, on its termination all the data in the individual caches which needs preserving must be merged into the cache of the process which spawned them, which introduces additional complexity, especially where the sub-processes acccept their caches concurrently, and thus have to compete for the use of the parent cache. Of course if the system provided a communal cache for the shared action (though not for nested actions) this activity would be much simpler, but competition would then be introduced every time a cache operation had to be made. The only advantage a communal cache would have is that the PRIOR opeation, described in section 3.9.2.1, could be provided for the acceptance tests of shared actions. If separate caches are used the interpreter would have to search the caches of all processes involved in the action to find the value, which, whilst theoretically possible, would be impractical, especially in distributed systems.

Synchronisation between processes in a shared action also needs examination, because the error described in section 3.8.1 where and unshared action waits for a condition involving a resource it has locked, is not an error in a shared action, for other processes can use the resource and make the condition true. This means that the system can detect when a user has failed to set such a condition, because if all the paths of a shared action have either finished or are waiting on a condition involving a resource locked by the action, then this error has occured.

In the next section we shall briefly describe how shared actions were incorporated into the system described in section 3.9.2.

4.3.2 Actual implementation -

Many of the difficulties described in the preceding sections did not arise in the trial system on MTS, simply because the language interface to the system was so restricted. The case of shared actions created by sub-processes could not arise as Concurrent Pascal does not have facilities for sub-processes, and shared actions had to be implemented using a CLASS structure with the third method of control described in section 4.2.1, because the scope rules made it impossible to define access rights using process identifiers. No attempt was made to enforce the rule that common resources may only be used inside unshared actions (to allow more experimental freedom), but, because of the way in which the AWAIT statement was implemented, (section 3.9.2.2) setting of SYNCHRONISING variables used between processes in a shared action had to be done inside an unshared action, so that the waiting process would be rescheduled and re-evaluate its condition.

As far as deadlocks were concerned, a shared action was assigned an initial priority equal to the number of processes involved in the action, and this meant that if a deadlock arose there was less likelihood of its having to be backed out. This device provides a simple measure of the cost of backing out an action and was found to be very effective in reducing the amount of recovery activity during a run. Deadlock detection using the blocking graph, is unaffected by the use of shared actions, but the implementation of the graph using blocking sets, means that when an action is blocking a resource request from one of the processes inside a shared action it must be regarded as blocking all the processes in the action rather than just the one which issued the request. If this is not done it would be possible for two processes within a shared action to deadlock each other, one unable to terminate because the other is still executing (albeit waiting) and so holding its locks, thus blocking other actions, which in turn could be blocking the other process. This means that membership of the blocking set of an action does not necessarily imply that a process is waiting for a resource held by the action, but rather that the termination of the action the process is a member of is dependent on that of the blocking action.

Altogether the implementation presented few problems, all of which were solved by simple extensions of the methods used for unshared actions. In the next section we shall look at an example to show how shared actions can be used.

4.4 The Dining Philosophers problem

The problem of the Dining Philosophers, or spaghetti eaters as they are sometimes known, was suggested by Dijkstra (Dij 72) and involves five philosophers who alternately think and eat. When a philosopher becomes hungry he sits down at a table and picks up two forks, one on either side of his plate, and eats. However, there are only five forks on the table, so if a philosopher is eating, neither of his neighbours can eat. When a philosopher finishes eating he puts down his forks and leaves the table. Therefore, if all five philosophers try to eat at the same time, each will pick up a fork, and there will then be none left on the table, so no philosopher will be able to make up a pair of forks and start eating. The problem is to prevent this and thus to stop the philosophers starving.

Various solutions have been presented in the literature - using semaphores in (Dij 72), critical regions in (HoA 72), communicating sequential processes in HOA 78, distributed processes in (BrH 78) and MONITOR's plus PORT's in (Shr 79a) - however the approach taken to solving the problem in every case requires the problem to be fully analysed before the algorithm can be developed. Fig. 4.3a shows the problem coded using shared atomic actions, implementing the dining philosophers exactly as they are described in the problem specification, and fig. 4.3b shows the execution flow and blocking graphs of the worst case, that is when the philosophers, having all sat at the table at the same time, do not spend any time thinking after finishing eating, but return at once to the table. After the initial deadlock, which the system detects and breaks by forcing one of the philosophers to return his fork to the table, each philosopher then eats in turn and nobody starves. When the philosophers start thinking again the situation will become more normal and two philosophers will be able to eat at the same time, the deadlock only occuring again should the five sit down to eat at exactly the same time. Note that when this program is run on the test system described above it is impossible for two philosophers to starve another sitting between them, as in the solution given in (Brh 78). This is because the way in which the queues of processes waiting to lock resources are implemented ensures that requests are processed in the order they occur.

This solution, however, does not support the kind of recoverability provided by the use of PORT's as, even when each of the atomic modules is made a recovery block, an error during a philosopher's thinking phase will cause all the philosophers to be rolled back, rather than just the one which failed, though errors during eating will be handled on an individual basis. We shall discuss the question of how such recoverability should be provided in chapter five and a different solution for the dining philosophers' problem will be presented in section 5.4.3.

```
TASK PHILS;
FORK: ARRAY [ 0..4 ] OF BOOLEAN;
I:INTEGER;
ACTION EAT (PHIL_NUMBER : 0..4);
BEGIN
   FORK [ PHIL_NUMBER ] := TRUE; -- pick up forks
   FORK [ (PHIL_NUMBER + 1) REM 5 ] := TRUE;
   .
   .
   -- eating
   .
   .
   FORK [ PHIL_NUMBER ]:= FALSE; -- put down forks
   FORK [ (PHIL_NUMBER + 1) REM 5 ] := FALSE;
END;

BEGIN
   FOR I := 0 TO 4 DO
      FORK [ I ] := FALSE; -- initial values
   SHARED PARBEGIN
      DO BEGIN THINK ; EAT(0) END 10000 TIMES |
      DO BEGIN THINK ; EAT(1) END 10000 TIMES |
      DO BEGIN THINK ; EAT(2) END 10000 TIMES |
      DO BEGIN THINK ; EAT(3) END 10000 TIMES |
      DO BEGIN THINK ; EAT(4) END 10000 TIMES
   PAREND;
END;
```

(a) The Dining Philosophers' problem

PHIL0	PHIL1	PHIL2	PHIL3	PHIL4	BLOCKING GRAPH
THINK	THINK	THINK	THINK	THINK	
LOCK F0	LOCK F1	LOCK F2	LOCK F3	LOCK F4	
LOCK F1	LOCK F2	LOCK F3	LOCK F4	LOCK F0	$\begin{tikzcd}[column sep=small] P4 \arrow[r] & P3 \arrow[r] & P2 \arrow[r] & P1 \arrow[r] & P0 \arrow[llll, bend right] \end{tikzcd}$
|	|	|	|	BACK OUT	$P3 \to P2 \to P1 \to P0$
|	|	|	END	LOCK F4	$\begin{tikzcd}[column sep=small] P3 \arrow[r] \arrow[rr, bend left] & P4 & P2 \arrow[r] & P1 \arrow[r] & P0 \end{tikzcd}$
|	|	|	THINK	|	$P2 \to P1 \to P0$
|	|	END	LOCK F3	LOCK F0	$\begin{tikzcd}[column sep=small] P2 \arrow[r] \arrow[rr, bend left] & P3 & P1 \arrow[r] & P0 \arrow[r] & P4 \end{tikzcd}$
|	|	THINK	|	|	$P1 \to P0 \to P4$
|	END	LOCK F2	LOCK F4	|	$\begin{tikzcd}[column sep=small] P1 \arrow[r] \arrow[rr, bend left] & P2 & P0 \arrow[r] & P4 \arrow[r] & P3 \end{tikzcd}$
|	THINK	|	|	|	$P0 \to P4 \to P3$
END	LOCK F1	LOCK F3	|	|	$\begin{tikzcd}[column sep=small] P0 \arrow[r] \arrow[rr, bend left] & P1 & P4 \arrow[r] & P3 \arrow[r] & P2 \end{tikzcd}$
THINK	|	|	|	|	$P4 \to P3 \to P2$

etc. etc.

(b) Worst case execution flow.

Fig. 4.3

4.5 Efficiency

The points raised in section 3.10 about the efficiency of the mechanisms described in chapter three are all still relevant when shared actions are implemented using them, but some additional problems are introduced. The first has been mentioned in section 4.3.2 and is the need to back out and retry all the processes involved in a shared action in the case of a deadlock. One reason for this was described above, but it may be argued that this simply brings to light an inadequacy in the deadlock detection mechanism, and that some other method, perhaps using more information about the resources that are being contended for, would allow individual processes in a shared action to be retried when a deadlock arose. However it must be remembered that the cache of a member of a shared action, does not contain a record of those resources which it has used after another member has locked them, so backing out the process on its own would not necessarily restore the system state correctly. Also if the processes in the action have been communicating, backing out an individual process would create inconsistent states, and so would not have the desired effect. It is clear that we must accept that all the processes have to be rolled back if one has to be. Section 4.3.2 described a method for assigning action priorities which reduces the risk of this having to happen, however the possibility cannot be ruled out altogether, and so this inefficiency cannot be eliminated.

The second area which needs to be mentioned is that of the necessity for all the processes involved in a shared action to terminate before commitment can take place, which may mean processes having to wait until others finish. This must happen because of the definition of recoverable shared atomic actions, however, if the time spent waiting for termination can be minimised, the action will block other actions for the least time. In order to achieve this all the processes must enter the action at the same time. Then termiation occurs when the longest individual path through the action finishes, and in order to achieve this the processes must all synchronise immediately before entry. Thus by shifting any excess waiting time from the end of an action to its beginning we have reduced its effect on the system. However, note that the system must know how many processes are involved in an action for this to be done, so the specification method used in the test system would not allow this to be implemented.

Finally, synchronisation between processes in a shared action using a resource local to the action means that the AWAIT statement mechanism must now look for events signalled during the course of an action, rather than checking when an action which may have flagged them terminates. This means that the amount of "busy-waiting" in the system may increase, thus reducing its efficiency. The use of specially handled SYNCHRONISING variables would allow the system to overcome this, but some overhead would still be felt, due to the checking that would then be carried out each time a SYNCHRONISING variable was used.

4.6 Conclusion

We have now seen how shared atomic actions can be integrated into the system supporting uncooperative processes, introducing the capability for inter-process communication, "cooperative" resource use (that is resources can be released befor committment) and controlling the way in which recovery takes place amongst groups of processes. However, for several reasons, these are only of limited usefulness. Fistly inter-process communication is hard to control and so increases the chance of programmer error, this being due to the fact that the processes within a shared action are not protected from each other Secondly, communication is still not general enough, because it can only take place between the members of the shared action, and not with any process.

The capability for releasing resources before final committment is more generally useful (as the example of the Dining Philosophers' problem shows) and is easy to control, but the restricted set of processes which can compete freely using it is again a disadvantage. However, the example also highlights the weaknesses of the recovery structure for, when the processes within an action only compete or communicate uni-directionally, recovery entails undoing large quantities of correct work done by processes which have not failed. Of course where bi-directional communication has taken place, or the processes are mutually dependent in some other way, as in the case of sub-processes, this type of recovery is exactly what is needed, and shared atomic actions, if carefully used, can make their implementation easier. Nevertheless, in many cases, for example where software modules are produced by programmers working independently of each other, more general methods of interaction are required and in the next chapter we shall show how these can be provided.

5.0 Cooperative Processes

5.1 Introduction

The preceding chapters have described a system that can support processes whose operations do not cause them to become dependent on others. Where dependency is required, that is where processes communicate, the programmer must encapsulate the processes in a shared atomic action, making them appear as a single process to the rest of the system. However, as we have seen, this structure has two major disadvantages. Firstly because of the recovery structure provided, the processes are too closely coupled, introducing the possibility of unnecessary recovery activity and excluding the use of compensation mechanisms. Secondly, the coupling of the processes means that fully asynchronous operations are impossible as <u>all</u> the processes within a shared action must synchronise on termination. This makes the implementation of certain kinds of systems inefficient and clumsy, for example one where a process collects information from a unknown number of other processes. In this chapter we shall describe some additional program structures which will allow systems like this to be constructed much more easily, whilst still being recoverable. The techniques to be described below, in effect, allow the support system to construct shared atomic actions, invisible to the user, made up of the processes which are communicating. This means that the rules of atomicity (Lom 76a) are not violated, and also permits the system to eliminate unnecessary recovery activity, as it has full knowledge of the way in which interactions have taken place. However, before describing these techniques, we must investigate the type of interface a programmer should have in order to construct such systems as the one mentioned above.

5.2 Programmer's Interface

The language interface that this thesis has developed to enable programmers to access shared resources is very simple - using a resource implies that it must be locked - and it is important that this simplicity be maintained as far as possible for any new structures. Bearing this in mind, there are two facilities that the programmer needs to be provided with :-

1. The ability to send a message to another process whose identity may or may not be known,

2. The ability to release certain resources when they are no longer needed by the process, thereby increasing the amount of concurrency possible (for example the dining philosophers' forks).

The first facility can be provided by the type of structures used by Hoare (Hoa 78) and Brinch Hansen (BrH 78), where messages are passed to either explicitly named processes or to one of an array of processes, picked out by a dynamically computed index. This construct does not allow a process to send a message to an unknown process, but this can be achieved by the use of BUFFER PROCESSES acting as intermediaries. The second facility can also be provided with this type of structure by using SERVICE PROCESSES, whose sole function is to perform an operation using some resource and return the results of this operation to the process that requested the service, thereby eliminating the need for the requestor to acquire and release the resource. This has considerable advantages from the point of view of program modularity, and hence software reliability, as it means that certain frequent operations need only be coded once and only one active copy need be kept in the system. Also, the programmer need only know about the functional properties of the resources so controlled, without having to know their exact structure, thus increasing the security of the system.

However, as we have seen in section 2.2.5 the synchronisation between processes inherent in this kind of system can impair its efficiency by reducing the level of concurrency and so the programmer may require another type of interface which will allow asynchronous communication. Message passing can be achieved by the support system itself buffering messages sent by a process and holding them until they are requested by another process. However, if messages must be directed to named processes the programmer must still use buffer processes to communicate with unknown processes, forcing him to replicate a facility provided by the system. For this reason, it would seem better to provide the user with buffers controlled by the interpreter level so that processes need only know the identity of a buffer where messages can be found or deposited rather than that of the messages' source or recipient. Unfortunately, this introduces the possibility of a process receiving a message not intended for it and this can only be avoided by programmer discipline and careful project management.

In many cases, however, the use of inter-task communication is not really necessary, and will introduce considerable overheads. The programmer must, therefore, be provided with some means of acquiring and releasing certain types of resources from within his processes, without violating the rules of atomicity, and we shall look at this topic in more detail in section 5.4.2.

Let us now look at some ways in which the above requirements could be realised, starting with a structure based on direct communication between processes.

5.3 The MARSHAL

5.3.1 General Description -

In the next few sections we shall look at a possible method of providing the programmer with the means to communicate between processes and to control resource allocation, based on the idea of "secretaries" and "directors", introduced by Dijkstra in (Dij 72). The MONITOR (Hoa 74) is the usual realisation of this concept, but, as it is implemented using direct procedure calls on a shared object, requires that processes be able to lock and release resources during the course of their executions, and, as we have seen, this violates the rules of atomicity and makes recovery difficult. It is possible to devise a scheme whereby recoverability can be maintained in these circumstances, but it either requires recovery information to be distributed throughout the caches of all the processes that have used a MONITOR, which presents problems of ensuring that values are restored correctly, or each MONITOR must have a cache associated with it, to centralise recovery information. The latter alternative is obviously preferable, but makes the MONITOR rather more than a data object, so in order to make it fit in better with the structure of the system we have developed, we shall introduce the <u>MARSHAL</u> as a type to take its place. A MARSHAL is a special process with its own cache, the body of which, like a monitor, consists of several atomic "entries" which can be called by other processes, using the type of remote procedure call described in (Brh 78) and (Ich 79), which we shall call a <u>RENDEZVOUS</u>. The MARSHAL itself controls which entries can be called at a given time by <u>GUARDS</u> - an entry only being accepted if the guard associated with it is true - and, as for MONITORs, only one entry can be active at a given time. The major advantage that this structure has over the MONITOR is that being an autonomous process any resources used in the body of a MARSHAL are locked by it, and are available to other processes through its auspices, thus avoiding the need for processes to lock and unlock resources. This is not the only advantage, for, if access to resources is controlled by MARSHAL's, they do not need to be available to all processes and each resource can be made local to the

MARSHAL controlling it, thereby reducing, and in some cases even eliminating, the need for common areas in the system. Also, the way in which access to entries is handled hides the queueing involved in their use, unlike the method used in MONITOR's, leading to more elegant algorithms.

The language structures used to define MARSHALs can take many forms but all have four features in common:-

1. Specification of the entries, as they are to be seen by an external user,

2. Declaration of resources local to the MARSHAL, but global to all its entries,

3. A piece of code used to initialize the resources when the MARSHAL is first initiated,

4. Specification of the bodies of the entries, and of the guards controlling their activation.

Fig. 5.1 shows two ways that MARSHAL's could be specified, the first being loosely based on the tasking structures in the ADA language (Ich 79) and the second being more close to the CLASS structure of MONITORs, each having its advantages and disadvantages, and showing that the MARSHAL structure can be represented in different ways.

```
MARSHAL BB IS
   ENTRY PUT ( X : IN DATA );
   ENTRY GET RETURNS DATA;
END;
MARSHAL BODY BB IS
ST : ARRAY [ BUFFER_RANGE ] OF DATA;
HEAD,TAIL : BUFFER_RANGE;
BEGIN
   HEAD := BUFFER_MIN; TAIL := BUFFER_MIN;
   LOOP
      SELECT
      WHEN HEAD <> TAIL
         ACCEPT GET;
         P : BUFFER_RANGE;
         BEGIN
            P := TAIL;
            TAIL := (TAIL+1) MOD BUFFER_TOP;
            RETURN ST[P]
         END;
   OR WHEN ( HEAD+1 ) MOD BUFFER_TOP <> TAIL
         ACCEPT PUT ( X : IN DATA );
         BEGIN
            ST[HEAD] := X;
            HEAD := (HEAD+1) MOD BUFFER_TOP;
         END;
      END SELECT;
   END LOOP;
END;
```

(a) Task notation for MARSHAL's

```
TYPE BB IS MARSHAL
   ST : ARRAY [ BUFFER_RANGE ] OF DATA;
   HEAD,TAIL : BUFFER_RANGE;

WHEN HEAD<>TAIL
   ENTRY PROCEDURE GET RETURNS DATA;
   P : BUFFER_RANGE;
   BEGIN
      P := TAIL;
      TAIL := (TAIL+1) MOD BUFFER_TOP;
   END;
WHEN (HEAD+1) MOD BUFFER_TOP<>TAIL
   ENTRY PROCEDURE PUT ( X : IN DATA );
   BEGIN
      ST[HEAD] := X;
      HEAD := (HEAD+1) MOD BUFFER_TOP;
   END;
   BEGIN
      HEAD:=BUFFER_MIN; TAIL:=BUFFER_MIN;
   END;
```

(b) Class type notation for MARSHAL's

Fig. 5.1

However no matter what notation is chosen, the underlying support mechanism for the MARSHAL is the same, and we shall now turn our attention to that.

<u>P</u>	<u>Q</u>	<u>R</u>	
ACTION A1	ACTION A2	ACTION A3	(1)
.	BB.PUT	.	(2)
.	.	BB.PUT	(3)
BB.GET	.	.	(4)
.	ACTION A21	.	(5)
ACTION A11	.	.	(6)
.	BB.PUT	.	(7)
BB.GET	.	.	(8)
END	.	.	(9)
.	END	.	(10)
BB.GET	.	.	(11)
END	.	.	(12)
	END	.	(13)
		END	(14)

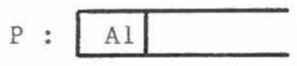
Fig. 5.2

5.3.2 <u>Special cache mechanism for MARSHAL's</u> -

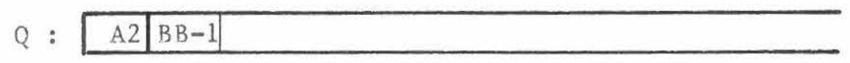
Whenever processes communicate dependencies are built upon uncommitted data, and so, if an error arises, the system must have recorded these relationships so that any atomic action that has used erroneous data can be wound back. Where communication is uncontrolled, this can be very difficult, but the restrictions enforced by the MARSHAL structure allow this to be done fairly easily. In order to maintain the necessary data, the system has to support MARSHAL execution with a cache mechanism that is different from the one we have described above. We shall describe how this operates with the aid of an example.

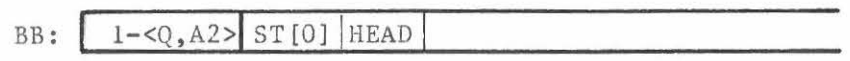
Consider the three processes - P, Q and R - shown in fig. 5.2. Processes Q and R communicate with P via a bounded buffer BB of the kind shown in fig. 5.1 with the type BUFFER_RANGE defined as the sub-range (0..1). The atomic actions A1,A2 and A3 constitute transactions and so are the outermost level of nesting for the parts of the processes shown in the figure. It is assumed that the processes execute, as far as possible, in parallel, so on that basis let us consider each numbered stage of execution in turn :-

1. All the processes' caches are empty, as is the cache of the MARSHAL BB. At this point barriers are created in each of the process caches to indicate the start of a new action. Process P's cache will look like this :-

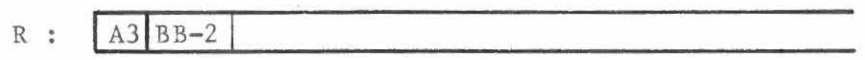
   

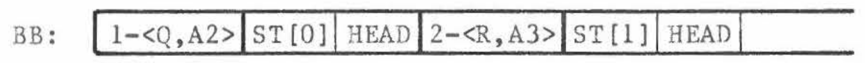
2. Process Q remotely calls the MARSHAL BB to put some data into the buffer. When BB accepts the rendezvous it generates a new sequence number to uniquely identify it and records this value in its cache, along with the identity of the action that requested the rendezvous. BB then executes the entry PUT, causing updated variables to be entered in its cache in the normal way. It then returns the rendezvous identifier to the calling process Q and waits for the next rendezvous. In the meantime Q has been inactive, but when the rendezvous is complete it caches the identity of the MARSHAL it has called and the sequence number it has been returned and continues. The cache changes are thus :-



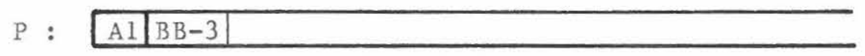


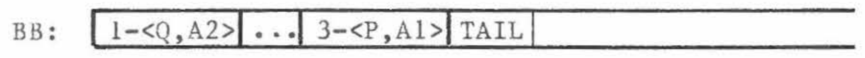
3. The same sequence of events is repeated but this time involving process R :-



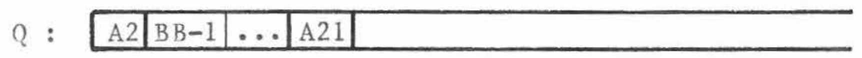


4. P now calls the entry GET to retrieve some data from the buffer, this call is processed in a similar fashion to the two preceding calls :-

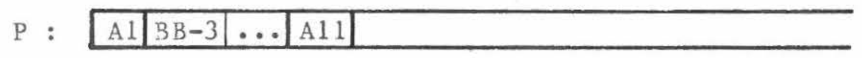




5. Process Q now enters a sub-action, creating a new barrier in its cache :-



6. Process P does the same :-



7. Q agains calls BE to put data into the buffer. When the rendezvous is accepted, the interpreter checks to see if the calling action has rendezvoused with BB before. If it has

not, as in this case, a new sequence number is generated and returned to the calling process. If it has, cacheing of updates takes place as normal, but no new rendezvous identifier is generated (see 11 below). The caches now look like this :-

Q :	A2	BB-1	...	A21	BB-4		
BB:	1-<Q,A2>	...	4-<Q,A21>	ST[0]	HEAD		

8. P calls BB to get some more data, and, as in the previous case, a new identifier is generated :-

P :	A1	BB-3	...	A11	BB-5	
BB:	1-<Q,A2>	...	5-<P,A11>	TAIL		

9. The sub-action A11 within process P terminates, and P's cache is processed. A message is sent by the support system to BB to say that A11 wishes to commit rendezvous 5, and BB's interpreter checks that this can be done. This rendezvous can be committed because P has a previous rendezvous with BB (number 3) to which recovery can be made if necessary. Note, however, that the value for the variable TAIL cached during the execution of rendezvous 5 must be propagated back to rendezvous 4 when 5 is deleted, so that recoverability can be maintained :-

P :	A1	BB-3	

BB:	1-<Q,A2>	...	4-<Q,A21>	ST[0]	HEAD	TAIL	

10. Q's sub-action also terminates, causing rendezvous 4 to be processed :-

Q :	A2	BB-1	

BB:	1-<Q,A2>	...	3-<P,A1>	TAIL	ST[0]	HEAD	

11. P calls BB yet again, but action A1 has already interacted with BB so there is no need to create a new rendezvous name. The reason for this is exactly the same as that for the single cacheing of resources in the normal cache mechanism, namely that only the earliest interaction need be recorded as that is the point to which recovery will take place. The execution of the entry GET does not alter any variables of BB that are not already cached, so the caches remain unchanged.

12. Action A1 terminates causing rendezvous 3 to be processed, P's cache is now empty again :-

BB:	1-<Q,A2>	...	2-<R,A3>	ST[1]	HEAD	TAIL	ST[0]	

13. A2 now terminates and rendezvous 1 must be processed. In this case acceptance can proceed, because it is the first rendezvous in the cache. However, if there had been others before it, acceptance would have been delayed until they had been committed and removed from BB's cache. If A2 had not

been an outermost action, the rendezvous would have been propagated back to its enclosing action, and the information in BB's cache about the rendezvous would have been altered to point to this other action. Note that cache processing can take place at either end of a MARSHAL cache, unlike the stack mechanism of the normal system. BB's cache now looks like this :-

BB:

2-<R,A3>	ST[1]	HEAD	TAIL	ST[0]	

14. A3 terminates and commits rendezvous 2, leaving all the processes' caches and the MARSHAL's cache empty.

This example has shown what occurs during normal system operation, however if an error arises recovery action must be taken. What happens then, is that the cache of the process in which the error has arisen is rejected and the system finds that it has rendezvoused with a MARSHAL. An interpreter level message is sent to the MARSHAL involved, indicating the rendezvous identifier that was in the cache. The MARSHAL's cache is then processed, rejecting the named rendezvous and all those that followed it. When a rendezvous is rejected, a message is sent to the process involved raising an error in that process and initiating recovery action for it. Thus recovery is propagated throughout the system. The reader can try this with the example given above by postulatng an error at some point in the execution flow and following the search for a recovery line. This will show that the mechanism described does collect sufficient information to allow recovery to take place.

Let us now summarise the way in which the MARSHAL cache mechanism would work :-

1. Only one rendezvous between a given action and a given MARSHAL is ever recorded,

2. Rendezvous information is recorded in both the calling process' cache and the MARSHAL's cache,

3. Commitment of an entry in the MARSHAL's cache occurs only when the action which requested the rendezvous concerned terminates and the conditions described in 4 hold,

4. An action may commit a rendezvous either if the rendezvous in question is the earliest in the MARSHAL's cache, or if its enclosing action has an earlier rendezvous with the MARSHAL.

   If neither of these is the case then, if the action is not an outermost action, responsibility for the rendezvous passes to its immediately enclosing action. This causes the rendezvous information to be propagated back in the process' cache, and to be altered in the MARSHAL's cache to indicate the new "owner". The final case is when an outermost action tries to commit a rendezvous which is not the earliest entry in a MARSHAL's cache. Here, cache acceptance must be delayed until all the earlier rendezvous have been committed, in which case it is safe to proceed.

5. When a rendezvous other than the earliest one is accepted, any alterations to variables not entered in the cache by its preceding rendezvous must be propagated back.

5.3.3 Conclusions Regarding MARSHAL's -

We have seen that the MARSHAL provides programmers with recoverable means of communicating between processes, by acting as an intermediary, and of cooperating, by allowing service processes to be constructed. It may be possible to develop the ideas presented to allow more general use of entries in processes, as suggested in (BrH 78). However, as the method has several disadvantages, this line of investigation appears not to be worth following. The most obvious disadvantage is shown by the example of the bounded buffer - actions which have called the MARSHAL will be wound back if a rendezvous earlier than their first is backed out, EVEN THOUGH NO ERRORS HAVE OCCURED IN THEM. This, as with shared actions, means that large quantities of correct work have to be undone, but, unlike shared actions, the programmer may have intended no dependency between the actions involved. In the example, this dependency comes about because of the method used to implement the buffer, which involves variables that are used by every call and so rendezvous cannot be independent. This dependency between processes can give rise to the "domino-effect", mentioned in section 2.3.3, where the search for a

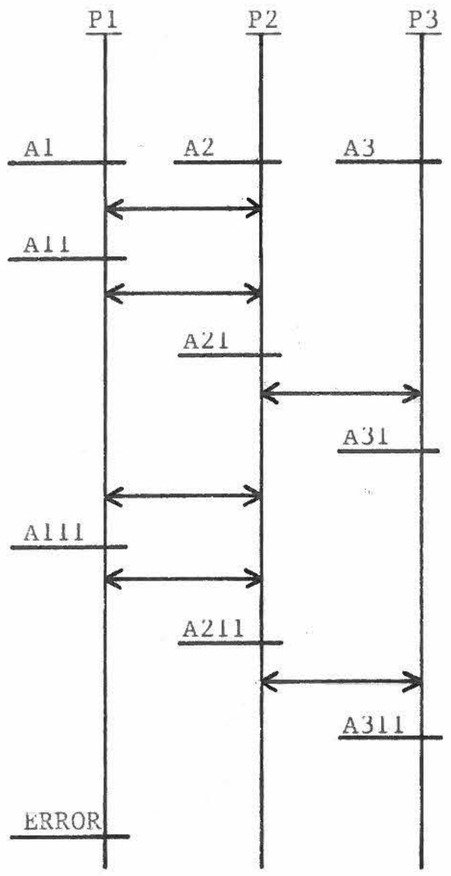


Fig. 5.3

recovery line causes nested actions to be backed out to their outermost level because of the way interactions have take place. Fig. 5.3 shows how this can happen in a simple three process system. The error in P1 causes A111 to be wound back, bringing with it A211 and thus A311, and so on, until the outermost level is reached.

The use of MARSHAL's also introduces the possibility of deadlocks which the support system cannot detect. These arise when guard conditions are malformed or when a deadly embrace occurs, and, as the relationships between the processes involved are only known at the user-level, these conditions are not apparent to the system, so no recovery can be initiated.

Another problem area is the remote procedure call mechanism used to invoke MARSHAL's, for this involves synchronisation between the MARSHAL and the calling action, and also actions waiting for guards to come true for their entry call to be processed. In many cases programmers just wish to leave a message to be "collected" any time after it has been left, and for their processes to proceed immediately without any waiting. Unfortunately the use of MARSHAL's may cause their processes to be delayed for unnecessary amounts of time, especially if the MARSHAL body contains any significant amount of computation. In the following section we shall show how other approaches to communication and cooperation can avoid the problems encountered by the use of MARSHAL's, which can really only be usefully used in a very limited number of applications.

5.4 Resources and their Use

5.4.1 Classification of Resources based on their usage -

In section 5.2 we looked at the sort of facilities programmers would like to have when implementing cooperative processes, and to develop a better way of supporting them than MARSHAL's we must look more closely at the way in which resources are used. The term resource has been used throughout this thesis to describe any "object" which a programmer may EXPLICITLY use, for example a named variable. Any object which he uses implicitly, that is not by name or reference in his program, is not considered a resource at the user level, though it may well be at the interpreter level. We shall split resources up into three categories based on the functions they perform for a programmer and the way in which they are used. Other classifications than the one to be described are possible and some resources may not fall easily into one category or another and some may even change category at some point in their lifetime. Nevertheless, based on these classes we can develop a recovery method for supporting cooperative processes. The categories are :-

1. Mutable resources - As its name suggests, a mutable resource is one which can be changed. That is an action may lock it, use its value, change that value, and free it for other actions to use, the best example of such a resource being a record in a data-base.

2. Consumable resources - these resources are locked by a process in the usual way, but disappear from the system when they are freed. Inter-process messages are a good example of consumable resources.

3. Reusable resources - the hallmark of a reusable resource is that it is always in the same state when an action locks it - that is, no inter-process communication can take place using it. Perfect examples are the "forks" used by the Dining Philosophers, which are always "clean" when picked up.

Mutable resources can obviously be used to communicate between actions - by their very nature the data they hold can be read by many actions, possibly over long periods of time. This is an extremely useful property, but when the contents of a mutable resource are uncommitted the support system must maintain a record of all actions which have become dependent on its value, so that recoverability is not impaired. However, this data can become very complicated and space consuming, so, as mutable resources are generally used for the storage of data with a relatively long life span, there is no harm in preventing access to them whilst their values are uncommitted. This means that actions may be delayed by having to wait till the contents are committed, but this disadvantage is outweighed by the facts that no data need be stored about dependencies and that the possibility of other actions being backed out due to an error in the value is eliminated. The preceding chapters in this thesis have developed a system which supports this type of use of resources and we must therefore turn our attention to the other categories of resources.

Having prevented actions from communicating during their lifespans using mutable resources, some other means must be found of allowing this facility, and consumable resources are ideal for this purpose. Consumable resources can only be used by one action after they have been created so there is a strictly one-to-one relationship between sender and receiver, which reduces the complexity of the recovery data that needs to be built up. In section 5.5 we shall show, in detail, how recoverability is maintained when consumable resources are used, and also develop language facilities which allow programmers to specify the use of resources which are created dynamically. Of course, an error in the creator of a consumable resource automatically means that the action which used it must be wound back. However, the dependency between creator and user is intentional and the programmer can foresee this eventuality, and can gauge the disruption an error can cause to a system.

We now have a means of inter-process communication, but actions can still not co-operate without becoming unnecessarily dependent on each other- the purpose of cooperation between actions being to allow a limited set of resources to be shared between them, without causing undue delays when an action′s needs cannot be satisfied. The resources which are being contended for are normally "tools" which an action wishes to use for a fixed period of time to operate on other resources, and reusable resources can be used for this purpose, providing several advantages. Firstly the "tool" is always in a predefined state whenever an action acquires it, which makes using it much more reliable. Secondly, because no communication can take place through reusable resources, an action need simply restore the resource to its initial state and can then free it, at any time, for use by other actions, without impairing recoverability. Finally, an action can ask for a resource by type rather than by specific identity, which reduces conflict between actions considerably.

When an action acquires a reusable resource, it is locked in the normal way and the fact of its acquisition recorded in the cache. The action may then use the facilities provided by the resource to perform operations on other resources, and then, when it is no longer needed, the action can free it deleting the entry made in the cache. Recoverability is not impaired by this, because any modifications made to other resources by the action using a reusable resource will cause cacheing in the normal way and so the recovery process can restore them without the need to know how the state changes were made. After the reusable resource has been freed no recovery action need involve it, but if an error should occur before it is released, the release must be done by the recovery mechanism - hence the cache entry for the duration of the action's possession of the resource.

The fact that a reusable resource is always in the same state when acquired by an action means that after it has been used its initial state must be restored. This can be done either with a "prelude" or a "postlude". The "prelude" method involves the execution of user supplied code which sets up the state when the resource is acquired and the "postlude" method either requires the state to be recorded in the cache so that it can be restored or, again, the execution of a piece of code. The former method, using a facility similar to that provided for CLASS and MONITOR initialisation in Concurrent Pascal has two advantages. It means that reusable resources do not need to be initialised when they are created, and also that when an action frees a resource it need only release the lock it holds, which is especially useful when recovery is in process. The only advantage that the latter method has is that, because a representation of a resource's state when it is first acquired can be stored in the cache (provided that the resource is created in the correct state) the system can automatically perform the reinitiallisation without the programmer providing a special procedure. In terms of efficiency the first method would therefore be preferable, but where the language interface does not allow for initialisation procedures, the second would have to be used.

The choice of language interface also affects the way in which the acquisition and release of reusable resources is specified and in the next section we shall look at several different ways this can be done.

5.4.2 Specification of Resource Acquisition and Release -

The simplest way of providing the programmer with a means of accessing reusable resources is to state that the procedural operations "acquire" and "release" are defined for any object described as reusable, thus:-

```
ACTION A1;
F: REUSABLE FORK;
BEGIN
   .
   F.ACQUIRE;
   .
   -- use F
   .
   F.RELEASE;
   .
END;
```

This method, being unstructured, gives the programmer complete freedom over where in his program acquisition and release of resources are carried out, allowing him to nest the calls within different procedure bodies. However, it has the disadvantage that should the programmer omit the "release" call the resource will be held by the process until its outermost action terminates, thus eliminating the possibility of competition with other processes. The use of the procedural notation is also at variance with the way mutable resources are acquired in the rest of the system, but this may be regarded as an advantage as it can be seen as a way of highlighting those parts of a program where cooperative use of resources is intended.

If the procedural notation is not to be used, some other way of indicating the programmer's intentions must be found. The acquisition of a resource is not difficult, the locking method described in the preceding chapters takes care of that, but automatically determining when a resource can be released, before the termination of the outermost action, cannot be done at run time (and would require a full analysis of the program at compile time). To overcome this, the user must be constrained to structuring the way in which he uses reusable resources so that the system may know when they can be released. There are two ways of doing this :-

1. Define the operation of the interpreter to be such that any reusable resource is released when the action that acquired it terminates,

2. Provide some language structure like the "WITH" statement of Pascal, of Shrivastava's "USING" statement, the start of

which indicates resource acquisition and the end, release.

Both these methods preclude the acquisition and release of resources from being in seperate program modules, and the former, though requiring no special syntax has the disadvantage that users may forget that a resource has been released and attempt to continue using it. This will not be seen as an error, but will be regarded as a valid attempt to re-acquire the resource, and when the request has been granted the action will proceed, possibly making invalid assumptions about the state of the resource.

The second method is identical to the use of Critical Regions by Brinch Hansen (BrH 72), though applied to a strictly limited set of resources, and has the advantage of allowing a compiler to detect where reusable resources have been used outside the appropriate language structure, thus preventing the error described above. From the point of view of reliability and efficiency, then, this would seem to be the best method and we shall now look at an example using it.

5.4.3 Reusable resources - the Dining Philosophers' problem -

We have already seen, in section 4.4, how this problem may be implemented using shared atomic actions, and have described the difficulties that the programmer encounters using them. The major difficulty was that independent recovery of a philosopher was not

```
TYPE FORK IS .... ; -- whatever a FORK looks like

TYPE PHIL IS
TASK ( F1,F2 : REUSABLE FORK )
BEGIN
   ENSURE ... BY -- some acceptance test
   DO
   BEGIN
      -- think
      WITH F1,F2 DO
      BEGIN
         .
         -- use the forks
         .
      END;
   END 10000 TIMES;
   ELSE BY ERROR;
END;

F : ARRAY [ 0..4 ] OF REUSABLE FORK;
PHILS : ARRAY [ 0..4 ] OF PHIL;

INIT PHILS[0](F(0),F(1)),PHILS[1](F(1),F(2)),
     PHILS[2](F(2),F(3)),PHILS[3](F(3),F(4)),
     PHILS[4](F(4),F(5));
```

Fig. 5.4

possible - a failure in one of them causing all the philosophers to be wound back. However, by using reusable resources this problem is overcome. Fig. 5.4 shows a program that implements this solution.

Each of the philosophers executes independently of all the others, and the deadlock recovery mechanism described in chapter three will detect and break any deadlocks that occur. This is because a request for a reusable resource must be passed through the same channels as for mutable resources and so the blocking graph will still indicate the presence of deadlocks. The processes can now compete with each other for the use of the forks, and dependency between them will only arise if there is explicit communication between any of the philosophers.

5.4.4 Limits on the use of Resources -

In section 3.9.1 we mentioned how most batch processing system apply not only time limits but also limits on the amount of output produced, to the programs they support. This is an example of a type of constraint associated with quantity of use rather than duration (though there is obviously some relationship between the two) and is not normally provided in programming languages. The safe programming constructs defined in (And 75) give the programmer this control over looping, and the various "range errors", like integer overflow or floating underflow, can be regarded as falling into this class of constraints, but control over the number of times a resource is used is not catered for. This type of control may be invisible to the end user of a resource, and thus allows the creator of the resource to have some control over how it is used. Other controls could be provided - a deadline specifying the maximum time a process may hold a resource for, and a limit to the maximum number of resources of a given type (or of any type) that a action can acquire. There are many ways of implementing and specifying such restraints and we shall not enumerate them here, however it is important that this kind of facility be considered when new systems are being designed.

5.5 Pools and Sequences

5.5.1 Structures for manipulating consumable resources -

As we have indicated above, the fact that consumable resources are usually created dynamically means the programmer can only access them through indirect references, as their names are not known at compile time. The language interface, therefore, must provide a way of supporting this. The use of reference variables is the normal method, but, in most programming languages, these are used either to point to existing variables, or, in conjunction with a procedure that returns a reference, to point to variables created in a free storage area. Consumable resources come into existence from different sources, and may not exist when an action attempts to use them, this means that the programmer must be able to specify which source he is requesting a resource from, and that the interpreter level must conceal any waiting that may occur. We shall introduce two structures with these properties - the POOL and the SEQUENCE. The declaration of pools and sequences follows the PASCAL syntax for sets and arrays :-

```
P1:POOL OF <type>;
S1:SEQUENCE OF <type>;
```

but they can only be used via two operators - "put" and "get". The "get" function waits until an object of the required type is created in the pool or sequence, removes it, and returns a pointer to it. The "put" operation creates a new object of the specified type, copies the object passed as a parameter into it, and adds it to the named pool or sequence. This far pools and sequences are identical and the difference between them arises because of their properties when recovery takes place.

We have already seen that when an error occurs in the creator of a consumable resource, the recovery action taken must be propagated to the consumer of the resource, but where an error occurs in the consumer it is obviously unnecessary for the producer to be wound back (one of the major disadvantages of MARSHAL's). What is needed is the ability to return consumable resources to their sources and then to re-read them if the alternate statement so wishes. However in what order must these messages be read back? Must it be the same as previously? By choosing a POOL or SEQUENCE the programmer can control this - pools implying no ordering relationship between messages, the reverse being true for sequences. If a pool is used to communicate between two actions it will probably look very like a sequence, as this would be the easiest way to implement it, however where several actions are creating messages for several others to consume, the difference between pools and sequences becomes marked. First consider a pool. If an error occurs in a resource creating action, only those actions which have consumed resources created by it need be backed out, other communications set up via the pool are not affected, but in a sequence this is not the case. If a creator error occurs, all resources created after the earliest erroneous resource will be regarded as in error and all their producers and all their consumers will be backed out. If a consumer error occurs, all consumers that have used resources created later than the first resource consumed in error must be backed out. Obviously this disruptive effect limits the usefulness of sequences, but where actions are highly dependent on each other, and are not within a shared action, this kind of recovery is necessary, and cannot be provided by the use of pools (note that using sequences is very similar to using MARSHAL's).

Let us now consider how pools and sequences are implemented at the interpreter level.

5.5.2 Implementation of pools and sequences -

We need not concern ourselves here with how the dynamic creation of resources is managed - in most systems this will be done using a common storage area controlled by an allocation package - but we do need to specify what recovery information needs to be kept with each resource and in the caches of the processes that manipulate them. Let us first look at a resource creator.

When an action creates a resource in a pool (or sequence - from now on any reference to a pool can be taken as including sequences, unless otherwise stated), the new resource is assigned an identifier uniquely identifing this member of the pool. Associated with the resource are two system controlled tag fields, one of which points to the action that created the resource, and the other the action which consumed it. This latter field is initialised to a value that indicates that the resource has not been consumed. If the action has not created a resource in this pool before, an entry is made in its cache recording the identity of the pool involved and that of the resource created. The newly created resource is now available to any consumers which may use the pool, though in the case of sequences, it will not be allocated until ALL the resources created before it have been removed. When the resource is finally taken by an action, the identity of this consumer is recorded in another system controlled tag kept with the resource, and the value of the resource is recorded in the consuming action′s cache. (This cacheing of the resource′s value, allows the consumer to alter the contents of the resource without impairing recoverability. However this additional freedom may not be needed and all consumable resources could be treated as read only, in which case only the identity of the resource need be recorded.)

Commitment of consumable resources occurs when the outermost enclosing action of the creating action terminates - responsibility for the resources being passed outwards on the acceptance of each of the nested actions. However, in the case of sequences committment cannot be completed until all other resources created before the first resource recorded in the action′s cache have been committed. When commitment is completed, the creating action identifier field of the resource is set to null, and this allows the consumers of the resources to terminate their activities, for a consumer cannot commit until all the resources it has consumed have been committed. When this has taken place, the system may delete the resource from the

$$\begin{tikzcd}
A7 \arrow[dr] & & & & \\
& A1 \arrow[r] & A2 \arrow[r] & A3 \arrow[dr] & \\
A6 \arrow[ur] & & A4 \arrow[ur] \arrow[ll] & & A5 \arrow[ll]
\end{tikzcd}$$

(a) Action dependency graph

$$\begin{tikzcd}
A7 \arrow[dr] & & \\
& A1 \arrow[r] & A2 \arrow[d] \\
A6 \arrow[ur] & & (A3,A4,A5) \arrow[ll]
\end{tikzcd}$$

(b) First stage of graph collapse

$$A7 \longrightarrow (A1,A2,(A3,A4,A5),A6)$$

(c) Final stage of graph collapse

Fig. 5.5

pool. However, this rule means that actions could become deadlocked when their commitments are mutually dependent, for example where messages have passed between two actions in both directions. The system, therefore, must take steps to detect this situation in order to allow termination to proceed, and the blocking graph method described in previous chapters can be adapted to do this. The detection algorithm must be modified, though, as actions are no longer involved in one-to-one relationships with each other - an action possibly being blocked by, and blocking, several others. Fig. 5.5 shows a possible graph, the direction of the arrows being from creator to consumer. In this example the commitment of action A7 is the crucial event, and when that has taken place all the other actions can be allowed to terminate, the interpreter having detected their interdependence. To do this it is necessary to "collapse" cycles that occur in the graph into larger nodes, as in fig. 5.5(b) and (c) - all the actions in a node being allowed to commit when the node no longer has any dependencies on other nodes.

The error recovery method described in the preceding section is very easy to implement, because of the inter-action relationship recorded in the tag fields associated with the resources. However, as with the other pool operations, error recovery in systems supporting true parallelism must be performed atomically in case several processes are trying to use the pool at the same time.

5.5.3 A test system -

In order to investigate the feasibility of pools, a trial implementation was carried out on a DEC PDP-11 under the RSX-11M operating system (DEC 79). This system provides powerful inter-process communication facilities, allowing sets of communicating tasks to be implemented very easily. It was decided to implement pools by the use of an interpreter extension (section 3.2), realised as a special task that carried out various pool operations at the request of "user" processes. These user processes were other RSX-11M tasks which simulated the use of atomic actions and exercised the pool handler via the system message passing facilities.

Four pool operations were supported :-

1. GET - this requested the pool handler to return the next data item from a pool named in the request message. The

item was marked as consumed, the dependency graph updated, and the data value returned to the requestor. If the pool was empty, the request was queued until another process put an item into the pool. This request involved two inter-task messages.

2. PUT - the data item contained in the request message was added to the specified pool. If there were processes waiting to GET a value from this pool, a GET operation was carried out for the process at the head of the queue. This request involved one inter-task message.

3. ERROR - all the pools were searched for resources created and consumed by the process that sent the message (as the users did not have recovery caches, this proved to be simpler to implement than the scheme described above). Consumed resources were marked unconsumed, and the processes which had consumed created resources were sent a message indicating that they should recover (this was a source of difficulty, as it required the use of asynchronous message handling which was not supported by the language used to implement the user tasks). This involves as many inter-task messages as are necessary to propagate the error throughout the system.

4. COMMIT - this request indicated that the process wished to commit its operations, and caused the pool handler to

analyse its dependency graph to see if this was permissible. If so, the process was allowed to proceed and all trace of it removed from the handler's tables. If not, the pool handler recorded the fact that this process wished to commit, only reactivating it either when an error arose, or when re-analysis of the dependency graph indicated that it could proceed. This involves two inter-task messages.

```
VAR
DEPEND : ARRAY [ ACTION_ID ] OF SET OF ACTION_ID;
CHECK  : ARRAY [ ACTION_ID ] OF BOOLEAN;
COMMIT : SET OF ACTION_ID;

FUNCTION CANCOMMIT( ACTION : ACTION_ID ) : BOOLEAN;
VAR LC : ACTION; RESULT : BOOLEAN;
BEGIN
   -- The set COMMIT contains all the actions which
   -- are waiting to commit. The array DEPEND contains
   -- the dependency information, and CHECK is used to
   -- detect circuits in the graph.

   IF NOT (ACTION IN COMMIT) THEN
      CANCOMMIT := FALSE -- this action is active
   ELSE
   BEGIN
      RESULT := TRUE;
      IF NOT CHECK [ ACTION ] THEN
      BEGIN -- this path has not been checked
         FOR LC IN DEPEND [ ACTION ] DO
         BEGIN -- check each blocking action
            CHECK[ACTION] := TRUE ; -- flag as checked
            RESULT := RESULT AND CANCOMMIT(LC);
         END;
      END;
      CANCOMMIT := RESULT;
   END;
END;
```

Fig. 5.6

The dependency graph was implemented as an array of sets of action identifiers, one element for each action in the system. When a GET request was processed, the set indexed by the requestor's name had the identity of the consumed resource's creator added to it. To determine whether or not an action could commit this "graph" was processed using the algorithm shown in fig. 5.6. The array CHECK is assumed to have all its elements set to FALSE before each call on CANCOMMIT, and is used to indicate whether the status of an action has been determined or not. This device prevents the program from entering an infinite loop, when analysing graphs with circuits in them.

The tests run on this system did not perform any computations, but simply used the pools to pass data between tasks. This meant that there was a very large number of inter-task messages being generated, and thus the overhead due to calls on operating system primitives was very high. In fact, most of the execution time for the tests was absorbed in executing these system functions and this was borne out by a simple test. If processes simply passed messages directly to each other, only one inter-task message would be generated, however, when we consider the strategy adopted for pools, we can see that this involves three messages to establish communication. Measurements taken on the system verified that performance was indeed degraded by a factor of about three, verifying that system overheads were swamping the small amount of computation in each task. In a "real" system, the number of messages being passed using pools would probably be considerably less than in this test, and so the performance of the system would be better. Nevertheless, it is obvious that the use of such an interpreter extension would not be desirable in a production system and experience with other systems of communicating processes under RSX-11M backs this up. A better way of implementing pools would be to provide each process with a copy of the pool handler and have each process perform its own operations (atomically) on a shared data area. This would reduce the number of inter-task messages required and increase the performance of the system. This could not be verified for the pool system, as modifications to the operating system would have been required, but the technique has been applied in another system (BSR 79) and significant improvements in performance were obtained.

Aside from the question of efficiency, pools were found to be easy to use (even with the low-level interface provided by the test system), and were flexible enough to allow a wide variety of communication patterns to be tried out. It also became apparent that the error recovery provided by pools did not need the support of the recovery cache in user processes, and could as easily have been built into a system based on, for example, exception handling mechanisms. Altogether the test system showed that pools were indeed a feasible way of supporting inter-task communication, and the indications were that the simplicity of the mechanisms involved would lend themselves to highly efficient implementation when incorporated directly into a system.

5.5.4 Additional features for POOL's and SEQUENCE's -

Several enhancements to the pool mechanism suggest themselves, the first of which is the attachment of priorities to resources put into a pool. This allows the programmer a certain amount of control over the order in which resources are consumed - the interpreter level always allocating the resource with the highest priority. (Note however that this kind of "queue-jumping" would not be acceptable for sequences where it would destroy their function). The introduction of priorities, however, brings with it the possibility of resources never being allocated, because their priority is always lower than others in the pool. Careful use of the facility should eliminate this risk, but, where this had to be avoided, the system could implement the kind of measures mentioned in section 3.9.1, where, after a predefined time has passed, a resource becomes FAVOURED and will be allocated in preference to one of higher priority. It would also be possible to associate deadlines with consumable resources, and allow the programmer to specify a time limit within which a resource must be consumed, though this would give rise to an error in the creating action rather than causing a resource to become favoured. In fact, the interpreter could use the user's deadline to control its mechanism for selecting resources to become favoured - resources with the shortest time left before their deadline expires being allocated first.

Another possible enhancement would be to provide a compensation mechanism for pools, rather like that of the reverse procedures of

PORT'S. When an error arose, rather than allowing a message to be reconsumed, the pool handler would transform the erroneous message into a compensating one, using a procedure provided by the user. This would allow completely uncoupled communication between actions, but, as with PORT's, would not guarantee that the compensating message would be consumed (for example, where the consumer stops consuming <u>before</u> an error arises in the producer). However, the advantage of the compensation mechanism would be that the dependency graph need not be maintained, and that commitment could always be allowed.

Apart from its use for message passing, the POOL structure could be used as a program interface for allocating any kind of dynamically created resources. Naturally, different internal structures would have to be developed to allow the allocation of resources other than messages and variables, however the commitment of an action using a resource obtained from a pool would still be delayed until the resource's creator had committed. This usage of pools also permits the allocation of such system resources as files to be integrated into a programming language without the need for structures that are type specific.

5.5.5 <u>Mutually suspicious processes</u> -

Now that we have seen how direct communication may be established between actions, we must give some consideration to the question of mutually suspicious processes. That is processes which,

```
-- some pools
A,B,C,D : POOL OF MESSAGE1;
X       : POOL OF MESSAGE2;
Y       : POOL OF MESSAGE3;

-- some pointers
M1      : @MESSAGE1;
M2      : @MESSAGE2;
M3      : @MESSAGE3;

.
SELECT M1 FROM A,B,C,D;
.
M1:=<A|B|C|D>;
.
```

(a) Two ways of writing simple pool selection

```
.
SELECT M1 -- pools all one type
-- M1 will point to message
WHEN A -> ....
WHEN B -> ...;
WHEN C -> ...;
WHEN D -> ...;
END;
.
SELECT -- pools different types
-- named pointer will point to
-- the resource obtained
WHEN A -> M1 : ...;
WHEN X -> M2 : ...;
WHEN Y -> M3 : ...;
END;
```

(b) Complex pool selection

```
.
SELECT M1 FROM A,B,C,D OR NONE;
.
M1:=<A|B|C|D|NONE>;
.
```

(c) Simple selection without waiting

```
SELECT
WHEN X    -> M2 : ...;
WHEN Y    -> M3 : ...;
WHEN NONE ->      ...;
END;
```

(d) Complex selection without waiting

Fig. 5.7

before requesting a resource, look to see if the request will be granted immediately and, if not, do not issue the request. This facility is generally employed where a process can use one of several different resources and will request the first one that is available, thus reducing the time it may have to spend waiting. In section 3.8.1 we discussed a similar requirement that arose with the use of the AWAIT statement, and the same type of non-deterministic structure adopted there - a guarded AWAIT statement - would seem to suit this case. There are two different circumstances in which a programmer could use a construct of this type. The first is where the programmer wishes to obtain a resource of a specific type, but from any one of several sources, and the second is where different computations are performed depending on the identity of the pool which provided the resource. Fig. 5.7 a and b show ways of representing such usage of pools. The question of "fairness", alluded to in section 3.9.1, arises here also in respect of which resource will be chosen if several are available. However, as was indicated, considerable research is still required into this problem before it can be resolved one way or another.

One further development of this usage, in the case where all the pools named in a statement are empty, is to give the programmer the option of waiting until one of his requests can be satisfied or of proceeding. This can be done, in a fashion similar to the DEFAULT option of the BCPL case statement (Ric 69), by providing a pseudo-pool which will always return a null pointer when accessed. Note, however, that this pool may be selected even when the other specified pools are not empty. This must be done in order to hide the vagaries of the process scheduling algorithm from the user, who could otherwise write programs which were dependent on certain execution flows (note that this possibility only arises in the case of empty pools, for at other times the user cannot determine the state of the other pools he has specified). Fig. 5.7 c and d show examples of this usage.

5.6 Conclusion

In the preceding sections we have discussed the implementation of two different styles of interface with which the programmer can be provided for implementing cooperative processes (i.e. processes which communicate and cooperate with each other) and we shall now summarise their advantages and disadvantages.

The method of direct communication, though providing a simple and modular interface, has two important disadvantages :-

1. Direct asynchronous communication between two processes is not allowed, thereby reducing system performance,

2. The use of buffer processes to allow messages to be passed to unknown processes can allow unwanted dependencies between processes to be built up - specifically where the buffer is being used by several processes that are otherwise independent - and this will lead to recovery action being

taken in processes where it is strictly not needed, should an error or a deadlock occur.

Where speed of operation is of no importance both these disadvantages can be discounted but there are many applications where this is not the case, so the less structured, but more efficient methods, based on the classification of resources described in section 5.4.1 would be preferable. However they also have their disadvantages :-

1. The use of consumable resources involves the system in extra book-keeping to control the termination of inter-acting processes,

2. The use of pools means that the system has to have quite a considerable storage area available to it for their allocation (c.f. the use of "pipes" in UNIX (Rit 78)),

3. The interface provided to reusable resources is such that multiple copies of operations on them, possibly using different algorithms, could exist, which could reduce the overall reliability of the system.

Nevertheless the structures described are sufficiently powerful that these disadvantages should be outweighed by the advantages that they give to the programmer.

6.0 Conclusion

6.1 The work presented

In chapter one the goal of this thesis was defined to be the development of a system that could support fault-tolerant concurrent programming without requiring complex extensions to the language interface to enable programmers to use its facilities. This was motivated by the need to make such facilities available to as wide a programmer base as possible so that the cost of software production could be kept down by reducing the amount of specialized knowledge needed to implement concurrent programs. Chapter two discussed the most commonly known techniques for controlling the use of shared resources and compared their characteristics with those of a hypothetical "easy to use" interface. This discussion showed that all the methods exhibited at least one of the following inadequacies:-

1. The technique was unstructured - that is the acquisition of a resource was not directly linked with its release by an explicit program structure;

2. The technique was unreliable - program errors occurring after a resource was acquired and before it was released, could result in either the release of erroneous information to other processes, or the permanent locking of the resource. Another source of difficulty was the possibility

of unrecoverable deadlocks,

3. The technique was too complex - complexity was introduced in two main ways, either through the need to perform analysis to ensure that a set of processes would not deadlock or by the limitations placed on the programmer by the technique making problems hard to solve.

Various approaches to improving the reliability of concurrent progamming by the incorporation of error recovery were then examined. These too had their difficulties, however, out of all the techniques examined in both areas, five constructs seemed to provide a basis for the kind of system that was required. They were :-

1. The recovery block - a clear, simple structure whose value had already been demonstrated by its use in fault tolerant sequential programs,

2. Deadlines/safe programming - also clear and well-structured, these were needed to overcome the problem of looping errors, undetectable in any other way,

3. Atomic actions - this construct, though having no implementation strategy defined for it, fitted the requirements set out in chapter two for a transaction orientated program structure which could support automatic control over resource usage,

4. Communicating processes - another concept which had not been tested in practical implementation, but which offered a well-structured means of handling messages between processes and which did not require the use of shared resources,

5. Reverse procedures - these provided a simple means of expressing compensation mechanisms. However there were reservations about them, especially concerning software errors within them.

The next chapter then showed the relationships between atomic actions and recovery blocks. Based on this it was demonstrated how the recovery cache mechanism, used to support recovery blocks, could be adapted to combine the automatic acquisition and release of resources with the collecton of checkpoint information about them. The use of this mechanism meant that deadlocks could arise when atomic action were competing for resources and a method to overcome this problem was developed. The strategy adopted was one of detecting an incipient deadlock and recovering from it. Detection was carried out by analysing the relationships between atomic actions with respect to outstanding resource requests and the resources each atomic action already owned. These relationships were recorded by means of a graph - a cycle in this graph indicating that a deadlock had arisen. One feature of this system was that only two actions could be involved in a deadlock thereby simplifying the task of breaking it. This was achieved by selecting one of the actions, judged to be the least important, and using the recovery cache mechanism to wind it back to its start, thus releasing the resources it held and breaking the deadlock. The selection of the action to be backed out should be based, if possible, on a priority assigned to the action, but if the priorities of the two actions are equal some other criterion must be used. Several criteria were suggested, but none of them were guaranteed to identify the "best" action to back out in all cases.

Various aspects of the control of synchronisation were then discussed, and program structures were suggested that could aviod the problems that were uncovered. At this stage we had a system which could support uncooperative processes (that is processes which hold resources for the duration of transactions and do not communicate with each other) and the interface to it had the characteristics that were set out in chapter two. Unfortunately the system could not support cooperative processes and so its usefulness was strictly limited. Chapter 4 introduced the concept of the shared atomic action which allowed a subset of cooperative processes, styled closely-cooperative processes, to be implemented. This extension allowed a process within a shared action to cooperate with the other processes in the action, forcing their recovery to be coupled. The interface provided was, however, too uncontrolled and methodologies were developed to enable processes to interact in relative security. However, even with this improvement, the structure, though useful in certain cases, was not sufficiently general and so other ways of allowing cooperation between processes had to be found.

Chapter five turned to the use of communicating processes and introduced the MARSHAL, a reliable "secretary". However the use of this structure was shown to lead, in certain common cases, to unnecessary coupling between actions, meaning that actions which should have been independent of each other were all wound back, if one of them failed. This structure was therefore rejected and we then examined the way in which resources were used by programmers. This led to the identification of three distinct classes - mutable, reusable and consumable. Mutable resources (those which held information for relatively long periods of time) could be controlled by the interface developed in chapter three, but the other types needed special interfaces. For reusable resources (those that are involved in an operation without their final state being affected) a PASCAL type WITH statement was suggested to indicate the points where they should be acquired and released. This structure, similar to the REGION statement described in section 2.2.3, limits the user slightly, but conforms to the requirements of chapter two.

Finally we investigated the class of consumable resources (those used to pass information between processes), introducing the new type forming operations POOL and SEQUENCE. It was then shown how resources defined with these operators could be used to communicate between actions and a system structure was described which allowed recovery to take place without affecting actions independent of the one that was in error. Two methods of recovery were allowed by this structure. Firstly actions could be coupled by the conceptual construction of shared atomic actions involving two communicating processes and secondly a compensation mechanism was suggested, to allow actions to remain uncoupled. The programmer interface provided to these facilities was again very simple, and allowed for considerable flexibility in program construction.

We can see then that the system described in this thesis will support fault-tolerant concurrent programming with a simple user interface, as was required of it. However one area has not been fully resolved. That is the question of the efficiency of the mechanisms that have been developed. Throughout this thesis the various factors involved have been discussed, and it would seem that the advantages to be gained from using the system will outweigh any inefficiencies in it.

6.2 Directions for future research

Having summarised the work that has been presented in this thesis and shown that the aims set out in chapter one have been met, we must examine some of the avenues down which further research could be directed. Some of these are concerned with general questions about the system itself, but others will have bearing on specific problems that arise in highly reliable concurrent systems.

6.2.1 Implementation -

So far only experimental implementations of parts of the system have been carried out, and there is still a considerable quantity of work to be done in the evaluation of the efficiency of the structures described. This is especially true of their use of multi-procesors, where the problems that arise are quite different to those encountered in a multi- programming implementation. The type of multi-processor used will also affect the implementation carried out, for where resources are shared between processors there must also be a common storage area in which system information required by all the processors is held. If the hardware available is distributed, that is the procesors are only connected by communication lines, the use of shared resources (if any exist) would be difficult to control, so the implementation of the inter-process communication features that have been described would take priority. Individual processors in a distributed network may also support multi-programmed processes introducing an additional level of complexity. To allow the implementation of reusable resources the use of service processes could be investigated. These are processes which can perform a specific set of operations for other processes, but have the reusable characteristic that their state is always the same when a request for an operation is received. An example of such a process would be a processor offering fast floating point computations to other processors in a network.

If distributed processes are not used, the main area of investigation is in the development of efficient hardware recovery cache mechanisms which can support the locking protocols we have described. Lee et al (Lee 79) have suggested that their hardware cache can be augmented to include support for other than sequential programs and this device would be an interesting start point for experiments. The system they have designed could only be used, as it stands, to support a multi-programmed version of the structures we have described, but incorporating interfaces to several buses would allow multi-processor implementations to be investigated. Deadlines are also an area where much work needs to be done. The basic type ALARM introduced in section 3.9.2 leads to unstructured use and is not linked closely enough with the recovery structure of a program. The published work of Campbell et al (Lie 80, Hor) tends to be theoretical rather than practical, though the structures they have introduced are excellent. The use of such facilities however, has not been investigated properly, and there appears to be no data available on how the programmer best determines the appropriate time interval to specify for a section of code. Obviously specific applications will have predefined time limits, but a set of guidelines that could be applied to programs will help the design process by indicating where these limits are unrealistic or where program efficiency need be improved.

One further topic which could be investigated is the enhancement of the data protection facilities provided by the system. With the current design a user may access any shared resource and the system will lock it, however the addition of a capability mechanism, such as that described by Needham (Nee 79), would allow invalid requests to be trapped, improving the reliability of the system considerably. A certain amount of control over resource access can also be achieved through management of the production of software, using methods of seperate compilation and program derivation such as those defined for the ADA language (Ich 79) and its support environment. The modular nature of the constructs that have been developed making them ideally suited for such treatment.

6.2.2 Systems without interrupts -

One area of development that has been suggested by Brinch Hansen (BrH 78) is the use of the concept of communicating processes in the development of systems without an interrupt mechanism (at least not one that is visible to the programmer). This would be achieved by having processes which explicitly wait for each event that can occur in the system and perform the necessary processing when they occur; returning to their wait state when this has been done. The overall effect of this should be to make event handling easier to program and more reliable, whilst still maintaining the essential non-determinism of the interrupt mechanism. Brinch Hansen envisaged such a system using the direct process to process communication he describes, but the Pool structure developed in chapter five can also be used to support this type of operation. For example, an (unreliable) teletype handler could be represented as shown in fig. 6.1.

```
KEYBOARD,PRINTER : POOL OF CHAR; -- char by char hardware
READ,PRINT : POOL OF BUFFER; -- whole line buffers
TASK TTY;
CH : CHAR; B:BUFFER;
   LOOP
      SELECT
WHEN KEYBOARD -> CH : -- key has been struck
         BEGIN
            BUFFER.FLUSH; -- new line so clear buffer
            WHILE CH<>EOL DO -- read in up to end of line
            BEGIN
               PRINTER:=CH; -- echo character
               BUFFER.PUT(CH); -- store character
               CH:=KEYBOARD; -- get next character
            END;
            READ:=BUFFER; -- make line available to user
         END;
WHEN PRINT -> B : -- user wishes to output a line
         BEGIN
            WHILE NOT BUFFER.EMPTY DO
               PRINTER := BUFFER.GET; -- type line
            PRINTER:=EOL; -- end line character
         END;
      END SELECT;
   END LOOP;
```

Fig. 6.1

6.2.3 A base for software testing and development -

One of the perennial problems of computing is the need to have systems providing a service continuously twenty-four hours a day, seven days a week, whilst still requiring new software to be developed and integrated into the system. These new program modules will introduce errors and need to be thouroughly exercised before they can be allowed to take up their intended place in the system. However, some of this testing must be carried out on the real system, off line testing not being fully adequate, and so a means of safely introducing new modules into the system must be found.

This can be easily achieved using the structures we have developed by allowing any atomic module to be converted into a recovery block (if it is not already written that way) with the new software as its primary. Then, should an error occur in the execution of the new module, the system can fall back to the previous version of the module which has been made the secondary alternate. An extra facility would be the ability to make the use of the new module conditional so that it was only used at predefined intervals, the old module being used as a primary for all the other occasions.

REFERENCES

And75
Anderson, T.

Provably safe programs.

Technical Report No. 70, Computing Laboratory, University of Newcastle upon Tyne February 1975

And76
Anderson, T. and Kerr, R.

Recovery blocks in action : A system supporting high reliability.

Proc. 2nd Int. Conf. on Software Engineering, San Francisco, U.S.A, October 1976, pp.447-457

And78
Anderson, T., Lee, P.A. and Shrivastava, S.K.

A model of recoverability in multi-level systems.

IEE Transactions on Software Engineering, Vol. SE-4, No. 6. (November 1978), pp.486-494

Ast76
Astrahan, M.M. et al

System R: Relational approach to data base management.

ACM Transactions on data base systems, Vol. 1, No. 2 (June 1976), pp.97-137

Bes78
Best, E. and Randell, B.

A formal model of atomicity in asynchronous systems.

Technical Report No. 130, Computing Laboratory, University of Newcastle upon Tyne, December 1978

Bes79
Best, E.

Aspects of occurrence nets.

Advanced Course on General Net Theory of Processes and Systems, Hamburg, October 8th-19th 1979

Bir73
Birtwistle, G.M., Dahl O-J, Myhrhaug, B. and Nygaard, K.

SIMULA BEGIN.

Auerbach Publishers Inc., Philadelphia, Pa. 1973

Bjo73
Bjork, L.A.

Recovery scenario for a DB/DC system.

Proc. of the ACM, 1973, pp.142-146

BrH72
Brinch Hansen, P.

Structured multiprogramming.

Comms. ACM, Vol. 15, No. 7 (July 1972), pp.574-577

BrH73
Brinch Hansen, P.

Operating system principles.

Prentice Hall, Cliffs, N.J., U.S.A., 1973

BrH75
Brinch Hansen, P.

The programming language Concurrent PASCAL.

IEEE Trans. on Software Engineering, Vol. 1, No. 2 (1975), pp.199-207

BrH76
Brinch Hansen, P.

Papers on the Solo operating system.

Software - Practice and Experience, Vol. 6, April-June 1976, pp.139-205

BrH78
Brinch Hansen, P.

Distributed processes: A concurrent programming concept.

Comms. ACM, Vol. 21, No. 11 (November 1978), pp.934-941

BSR79
British Ship Research Association

Results of internal experiments.

1979

Cam74
Campbell, R.H. and Habermann, A.N.

The specification of process synchronisation by path expressions.

Lecture notes in Computer Science 16, Springer Verlag, 1974, pp.89-102

Cha74
Chamberlin, D.D., Boyce, R.F. and Traiger, I.L.

A deadlock-free scheme for resource locking in a data base environment.

Information Processing 74, North Holland Publishing Co., Amsterdam, 1974, pp.341-343

Cof71
Coffman, E.G., Elphick, M.J. and Shoshani, A.

System deadlocks.

ACM Computing Surveys, Vol. 3, No. 2 (June 1971), pp.67-68

Con63
Conway, M.E.

Design of a seperable transition-diagram compiler.

Comms. ACM, Vol. 6, No. 7 (July 1963), pp.396-408

Cre78
Cress P. et al

/360 WATFOR implementation guide.

University of Waterloo, Waterloo, Ontario, 1978

Cri79
Cristian, F.

A recovery mechanism for modular software.

IEEE 4th Int. Conference on Software Engineering, Munich, September 1979, pp.42-50

Dav73
Davies, C.T.

Recovery semantics for a DB/DC system.

Proc. of the ACM, 1973, pp.136-141

Dav79
Davies, C.T.

Data processing integrity.

in Computing Systems Reliability : an advanced course (Ed. Anderson, T. and Randell, B.), Cambridge University Press, 1979, Chapter 8, pp.288-354

DEC79
Digital Equipment Corporation

Introduction to RSX-11M

Digital Equipment Corporation, AA-2555D-TC, 1979

Den66
Dennis, J.B. and Van Horn, E.C.

Programming semantics for multi-programmed computations.

Comms. ACM, Vol. 9, No. 3 (March 1966), pp.143-155

Dij68a
Dijkstra, E.W.

Co-operating sequential processes.

in Programming Languages (Ed. Genuys, F.), Academic Press, New York, 1968

Dij68b
Dijkstra, E.W.

The structure of the THE multiprogramming system.

Comms. ACM, Vol. 11, No. 5 (May 1968), pp.341-346

Dij72
Dijkstra, E.W.

Hierarchical ordering of sequential processes.

in Operating System Techniques, Academic Press, New York, 1972, pp.72-93

Dij75
Dijkstra, E.W.

Guarded commands, nondeterminacy and formal derivation of programs.

Comms. ACM, Vol. 18, No. 8 (August 1975), pp.453-457

DoD78
U.S. Department of Defence HOLWG.

Preliminary "STEELMAN".

in Proceedings of the "IRONMAN" language seminar (ed. Freeman W.), York Computer Science Report No.22, University of York, England, 1979

Esw76
Eswaran, K.P., Gray, J.N., Loeiw, R.A. and Traiger, I.L.

The notions of consistency and predicate locks in a database system.

Comms. ACM, Vol. 19, No. 11 (November 1976), pp.624-633

Goo75a
Goodenough, J.B.

Structured exception handling.

Second ACM Symposium on Principles of Prog. Lang., Palo Alto, January 20th-22nd 1975, pp.204-224

Goo75b
Goodenough, J.B.

Exception handling: Issues and a proposed notation.

Comms. ACM, Vol. 18, No. 12 (December 1975), pp.683-696

Gra76
Gray, J.N., Lorie, R.A., Putzolu, G.R. and Traiger, I.L.

Granularity of locks and degrees of consistency in a shared data base.

in Modelling In Data Base Management Systems (Ed. Nijssen, G.M.), North Holland Publishing Co., 1976, pp.365-394

Hab69
Habermann, A.N.

Prevention of system deadlocks.

Comms. ACM, Vol. 12, No. 7 (July 1969), pp.373-385

Har77
Hartmann, A.C.

A Concurrent PASCAL compiler for minicomputers.

Lecture Notes in Computer Science 50, Springer Verlag, 1977

Hea73
Hearst, F.E., Ornstein, S.M., Crowther, W.R. and Barker, W.B.

A new minicomputer/multiprocessor for the ARPA network.

Proc. 1973 AFIPS National Comp. Conference, Vol. 42, AFIPS Press, Montrale, N.J., pp.529-537

Hoa72
Hoare, C.A.R.

Towards a theory of parallel programming.

in Operating System Techniques, Academic Press, New York, 1972, pp.61-71

Hoa74
Hoare, C.A.R.

Monitors, an operating system structuring concept.

Comm. ACM, Vol. 17, No. 10 (October 1974), pp.549-557

Hoa78
Hoare, C.A.R.

Communicating sequential processes.

Comms. ACM, Vol. 21, No. 8 (August 1978), pp.666-677

HoC
Horton, K.H., Campbell, R.H. and Belford, G.G.

Meeting real-time deadlines.

Department of Computer Science, University of Illinois at Urbana-Champaign

Hol71
Holt, R.C.

Comments on prevention of system deadlocks.

Comms. ACM, Vol. 14, No. 1 (January 1971), pp.36-38

Hol72
Holt, R.C.

Some deadlock properties of computer systems.

ACM Computing Surveys, Vol. 4, No. 3 (September 1972), pp.179-196

Hor74
Horning, J.J., Lauer, H.C., Mellier-Smith, P.M. and Randell, B.

A program structure for error detection and recovery.

Proc. Conf. on Operating Systems, IRIA, 1974, pp.177-193

Ibma
IBM system/360 operating system

PL/I (F) Language Reference Manual.

GC28-8201-4, File No. 5360-29

Ich79
Ichbiah, J. et al

Prelimary ADA reference manual and rationale for the design of the ADA programming language.

ACM Sigplan notices, Vol.14, No. 6 (June 1979), Parts A and B

Kah76
Kahn, G and MacQueen, D.

Coroutines and networks of parallel processes.

IRIA Research Report No. 202, November 1976

Kie79
Kieburty, R.B. and Silberschatz, A.

Comments on "Communicating sequential processes".

ACM Tran. on Prog. Langs. and Systems, Vol.1, No.2 (October 19)79, pp.218-225

Kim76
Kim, K.H. and Ramamoorthy, C.V.

Failure-tolerant parallel programming and its support architecture.

NCC 76, pp.413-423

Kim78
Kim, K.H.

An approach to programmer transparent coordination of recovering parallel processes and its efficient implementation rules.

Presented at 1978 Int. Conf. on Parallel Processing, August 1978

Lau75
Lauer, P.E. and Campbell, R.H.

Formal semantics of a class of high-level primitives for co-ordinating concurrent processes.

Technical Report No. 74, Computing Laboratory, University of Newcastle upon Tyne, April 1975

Lee79
Lee, P.A., Ghani, N. and Heron, K.

A recovery cache for the PDP-11.

Technical Report No. 134, Computing Laboratory, University of Newcastle upon Tyne, March 1979

Lie80
Liestman, A.L., and Campbell, R.H.

A fault-tolerant scheduling problem.

Department of Computer Science, University of Illinois at Urbana-Champaign

Lip75
Lipton, R.

Reduction: A Method of proving properties of parallel programs.

Comms. ACM, Vol. 18, No. 12 (December 1975), pp.717-721

Lom76a
Lomet, D.B.

Process structuring, synchronisation and recovery using atomic actions.

Technical Report No. 92, Computing Laboratory, University of Newcastle upon Tyne, July 1976

Lom76b
Lomet, D.B.

A practical deadlock avoidance algorithmn for data base systems.

IBM Research Report RC 6288 (26989), 11th February 1976

Mel76
Melliar-Smith, P.M. and Randell, B.

Software Reliability : The role of programmed exception handling.

Technical Report No. 95, Computing Laboratory, University of Newcastle upon Tyne, December 1976

Mer77
Merlin, P.M. and Randell, B.

Consistent state restoration in distributed systems.

Technical Report No. 113, Computing Laboratory, University of Newcastle upon Tyne, October 1977

Nee79
Needham, R.M.

Protection.

in Computing Systems Reliability : An advanced course (Ed. Anderson, T., and Randell, B.), Cambridge University Press, 1979, Chapter 7, pp.264-287

Orn75
Ornstein, S.M., Crowther, W.R., Kraley, M.F., Bressler, R.D., Micheal, A. and Hearst, F.E.

Pluribus - a reliable multi-processor.

Proc. 1975 AFIPS Natl. Computer Conference, Vol. 44, AFIPS Press Montrale NJ, pp.551-559

Par72
Parnas, D.L.

Response to detected errors in well-structured programs.

Carnegie-Mellon University, July 1972

Pet77
Petri, C.A.

General net theory.

Proc. of the joint IBM/University of Newcastle upon Tyne Seminar on Computing System Design (Ed. Shaw B.), Computing Laboratory, University of Newcastle upon Tyne, 1977, pp.131-169

Ran75
Randell, B.

System structure for software fault-tolerance.

Sigplan notices, Vol. 10, No. 6 (June 1975), pp.437-449

Ran78
Randell, B., Lee, P.A. and Trelevan, P.C.

Reliability issues in computing system design.

ACM Computing Surveys, Vol. 10, No. 2 (June 1978), pp.123-165

Ric69
Richards, M.

BCPL: A tool for compiler writing and system programming.

Spring Joint Computer Conference, 1969, pp.557-566

Rit78
Ritchie, D.M.

Unix time-sharing system : A retrospective.

The Bell System Technical Journal, Vol.57, No.6. Part 2 (July-August 1978), pp.1947-1969

Sal75
Saltzer, J.H. and Schroeder, M.D.

The protection of information in computer systems.

Proc. IEEE, Vol.63, No.9 (1975), pp.1278-1308

Shr78a
Shrivastava, S.K. and Banatre, J-P.

Reliable resource allocation between unreliable processes.

IEEE Trans. on Software Engineering, Vol. 4 No.3 (1978), pp.230-241

Shr78b
Shrivastava, S.K.

Sequential PASCAL with recovery blocks.

Software - Practice and Experience, Vol. 8 (1978), pp.177-185

Shr79a
Shrivastava, S.K.

Concurrent PASCAL with backward error recovery : Language features and examples.

Software - Practice and Experience, Vol. 9 (1979), pp.1001-1020

Shr79b
Shrivastava, S.K.

Concurrent PASCAL with backward error recovery : implementation.

Software - Practice and Experience, Vol. 9 (1979), pp.1021-1033

Ver77
Verhofstad, J.S.M.

The construction of recoverable multi-level systems.

PhD Thesis, University of Newcastle upon Tyne, August 1977

Ver78
Verhofstad, J.S.M.

Recovery techniques for database systems.

ACM Computing Surveys, Vol.10, No.2 (June 1978), pp.167-195

Wij69
V. Wijngaarden, A.

Report on the algorithmic language ALGOL 68.

Num. Mathematik 14, 1969, pp.79-218

Wir71
Wirth, N.

The programming language PASCAL.

Acta Informatica 1, 1971, pp.35-63

Wir77
Wirth, N.

MODULA : A programming language for modular multi-programming.

Software - Practice and Experience, Vol. 7, No. 1 (1977), pp.3-35