

THE CONSTRUCTION OF RECOVERABLE MULTI-LEVEL SYSTEMS

Joost S.M. Verhofstad

Ph.D. Thesis

Computing Laboratory,
University of Newcastle upon Tyne, England.

August 1977

Abstract

Systems structures and data structures which make possible the state restoration of user objects, are described in this thesis. Recovery is linked with types, which suggests making a distinction between recoverable and unrecoverable types. For convenience, recovery is discussed in terms of recovery blocks as developed at the University of Newcastle upon Tyne. Recovery is taken to mean restoring the values of recoverable types.

Recoverable multi-level systems are considered. On the one hand levels in such systems can be backed out. On the other hand these levels provide explicit recovery for new types they introduce, and so can be called on to restore states of objects used in higher levels. The concepts and issues are discussed and explained; mechanisms and techniques for building such systems are presented.

Recovery techniques for complex global data structures and techniques to maintain consistency at any time, even when recovery is impossible such as after a crash, are described and compared.

Many of the presented techniques are employed in an implemented recoverable two-level system, with a recoverable filing system. This two-level system is described in detail.

It is argued that in order to implement recoverability in multi-level systems with efficiency and flexibility, the interfaces of the system should provide both recoverable and unrecoverable types.

It is also shown that the way in which complex data structures are updated is of major importance if recovery is to be provided in a "reasonably" efficient way and consistency is to be guaranteed after a crash.

Acknowledgements

Professor B.Randell, who has been the supervisor for my Ph.D. study, has been a constant source of encouragement and helpful suggestions. I would like to thank him for his critical reading of many of my memos, reports and early drafts which finally resulted in this thesis. His criticisms and comments were always of such quality that I regard myself extremely lucky in having been given the opportunity to be one of his students, hard taskmaster though he is.

My special thanks are due to Dr. T.Anderson and Mr. P.M. Melliar-Smith who have been invaluable sounding boards for many of my ideas and have spent a great deal of time with me discussing these ideas in detail.

I would also like to thank everybody in the Computing Laboratory who has helped me by reading, criticizing and discussing earlier drafts of (parts of) this thesis, and for explaining and discussing their ideas on recovery.

I gratefully acknowledge the financial support I received for twelve months from the Netherlands Organisation for the Advancement of Pure Research (Z.W.O.).

IBM UK Scientific Centre kindly allowed me to use their facilities to produce the thesis as presented here.

CONTENTS

1.0	Introduction.	1
1.1	The goal of the thesis	5
1.2	Summary of thesis.	7
2.0	A survey of techniques for recovery and crash resistance.	9
2.1	Introduction	9
2.2	The categorization of the techniques	12
2.3	Salvation programs	18
2.4	Incremental dumping.	19
2.5	Audit trail.	21
2.6	Differential files	24
2.7	Backup and current versions.	27
2.8	Multiple copies.	30
2.9	Careful replacement.	31
2.10	Summary and conclusions	37
3.0	Partially recoverable interfaces in multi-level systems	41
3.1	Introduction	41
3.2	Definition of multi-level system and recoverability	42
3.3	Basic principles	47
3.4	Completely recoverable interfaces.	49
3.4.1	Level: programs and data objects.	49
3.4.2	Interpreters for CRI-levels	56
3.5	The disadvantages of completely recoverable interfaces.	58
3.5.1	An alternative CRI-scheme: "bottom level" recovery only	60
3.6	Partially recoverable interfaces	61
3.7	The problems and constraints of the PRI-scheme .64	
3.7.1	The logging mechanism	65
3.7.1.1	An implementation of the logging mechanism.	68
3.7.1.2	The relation between the logging and the cacheing mechanism	72
3.7.1.3	The properties and constraints of PRI-levels	76
3.7.2	The data structure of the log	78
3.7.3	Systems consisting of PRI-levels.	81
3.8	A two-level prototype system	86
3.9	Recoverable types as a basic concept for recoverable multi-level systems.	89
3.9.1	A recoverable lineprinter manager on top of the recoverable file manager.	89
3.9.2	Recoverable types versus recoverable operations in a multi-level system.	91
3.10	Conclusions and relation to other areas	94
4.0	Recovery for complex data structures.	96

4.1	Definitions96
4.2	An example of a complex data structure: a filing system98
4.2.1	The structure of the filing system98
4.2.2	The mechanisms to provide recoverability for files	102
4.2.2.1	The mechanisms for updating and cacheing of files	103
4.2.2.2	Processing of caches and files after the acceptance test	106
4.3	Cacheing for complex data structures	108
4.3.1	Separating data providing an abstraction and information carrying data	109
4.3.2	The cacheing problems for complex data structures	111
4.3.2.1	A general solution and two specific implementations	113
4.3.2.2	A comparison of the two methods	118
4.3.3	Alternative solutions to the cacheing problem	120
4.3.4	Main conclusions	121
4.4	Maintaining consistency in recoverable complex data structures	122
4.4.1	Crash and crash resistance	122
4.4.2	The provision of crash resistance with recovery	125
4.4.2.1	The critical updates	126
4.4.3	Conclusions	128
5.0	A cost analysis of the implemented recoverable prototype system	129
5.1	Introduction	129
5.2	Basic principles of the cost analysis	130
5.3	Execution times: an analysis of the overhead in disk accesses	131
5.4	Program sizes	137
5.5	Cache space	137
5.6	A comparison with other techniques	138
5.7	Some experimental results	141
5.7.1	A little test program	143
5.7.2	A real-life utility program	144
5.8	Conclusions	146
6.0	Directions for future research and conclusions	147
6.1	Directions for future research	147
6.1.1	The construction of complete acceptance tests	150
6.1.2	A single computation	151
6.1.3	The construction of acceptance tests from abstract specifications	152
6.1.4	An acceptance test problem in multi-level systems	157
6.2	Conclusions	159

References 162

1.0 INTRODUCTION

Recovery techniques can be used to restore data in a system to a usable state. Such techniques are widely used in filing systems and data base systems, in order to cope with failures. A failure of a system occurs when that system does not perform its service in the manner specified, so a failure is an event. These failures can be of different natures, such as failures caused by hardware faults (e.g. a power failure or disk failure), software faults (e.g. bugs in programs or invalid data) or human errors (such as when the operator puts a wrong tape on a drive, or a user does something he did not intend doing). A failure occurs when the system is in an erroneous state and the normal algorithms of the system continue processing (MeR76). The term error is, in this context, used for that part of the state which is "incorrect". An error is thus a piece of information used as a casual equivalent for erroneous state. A fault is a mechanical or algorithmic cause of an error.

A system can be designed to be fault tolerant by incorporating into it additional components and abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures, or which deal with these failures and restore the system to a "correct" state from which normal processing (using the normal algorithms) can continue. These additional components and abnormal algorithms will in this thesis be termed recovery techniques and are the subject of the present thesis. There are many kinds of failures and therefore many kinds of recovery possible. For the recoverable systems considered here, the recovery mechanisms and redundant data maintained to make recovery possible (recovery data), form an integral part of the system. There is always a limit to the kind of recovery that can be provided. If a failure not only corrupts the ordinary data, but also the recovery data, then there are obviously problems. As described by Randell (RLT77), a recovery mechanism will only cope with certain failures. The failures it does not cope with may, for example, be rare, or not have been thought of, or have no effects, or it could be too expensive to provide recovery to cope with them. For example, a head crash on disk may not only destroy the data, but also the recovery data. It would therefore be preferable to maintain these recovery data on a separate device. However, there are obviously other failures which may effect that separate device (for example failures in the machinery that writes the recovery data to that storage device). Recovery data can itself be protected from the consequences of failures by the provision of a yet further form of recovery data to provide the ability to restore the "primary" recovery data in the event of a failure which

corrupts those recovery data. This progression could in theory go on indefinitely. In practice of course, there must be some total reliance on the reliability of some ultimate recovery data (or rather, some degree of acceptance of the fact that such recovery data is not totally reliable).

Reliability, or some level of reliability of a system, can be achieved by avoiding faults or by tolerating them. The second approach, fault tolerance, is a common way of dealing with (certain types of) hardware faults; read after write plus retry as a means of tolerating faults is an example. This fault tolerant approach has been used extensively for hardware where very high reliability is needed (Avi71), (Wen72), (Bor72), (Sk176).

The traditional, and often satisfactory, approach to achieving reliable software has been based on avoiding faults. This approach has been termed fault intolerance by Avizienis (Avi75). The aim of this approach is to build systems such that all the causes of failures are eliminated as much as possible. Occasional system failures are accepted as a necessary evil, and (usually manual) maintenance is provided for their correction. Over the past years a variety of methods and utilities have been developed which can be used to try to obtain reliable software using the fault intolerant approach. Examples of such utilities are debugging facilities (Sat72), (IBMa), (IBMb) and methods of systematically testing programs. In the area of program testing a method capable of demonstrating the absence of errors using condition tables has been reported (GoG75). Also methods ensuring that all of the possible paths through the program will be traversed have been developed (MiM75). The latter methods, however, do not demonstrate the absence of errors. However, all of these testing methods seem to have been defeated by theoretical difficulties and practical problems, so they have not yet led to any practicable utilities. Systems have been built to allow the symbolic execution of programs which can be used to test and debug programs (Kin76) or to find test data (Boy75). Much work has also been done in the area of program design (DDH72), (Jac75) and language design (WLS76), (Ast75) to facilitate the construction of more reliable programs. Also specification methods used to give abstract specifications of programs such that more reliable software can be obtained, have been developed (Par72a), (Par72b), (Rob75), (Gut76). One method on which a lot of work has been done and goes as far as you can go to obtain reliable software using the fault intolerant approach, is the proving of correctness of programs by formal analysis. Much work has been done in this area (BaW76), (Els72), (Gut76), however, the practical accomplishments in this area so far fall short of a tool for routine use.

There are two major reasons why the fault intolerant

approach is not a complete solution for software for which high reliability is required:

1. All the strategies and techniques based on the fault tolerant approach have in common that at some stage a program is assumed to be correct (i.e. fully debugged or proven or demonstrated by testing to be correct). However, none of the strategies and techniques guarantee that the final program will be 100% correct. If the program is large then it will probably still contain some errors, as indeed even small apparently proven programs can do (see (GoG75)).
2. The correct working of the program, i.e. the required reliability and availability, does not only depend on the correctness of the program. A failure can be caused by any other part of the system in which the program is running, for example, by the operator, a mechanical defect or other programs in the system.

Since the fault intolerant approach may not be sufficient if high reliability is required, fault tolerance can be sought, not as an alternative, but rather as a complementary technique. The key to fault tolerance is redundancy. The three principle forms of software redundancy for obtaining fault tolerance, which were also distinguished by Avizienis, are:

1. Multiple storage of programs and data.
Kopetz (Kop74) shows with a mathematical model how great the improvements in reliability are when stand by software is used (different algorithms should be used for the processing modules and spare modules). Several applications of this principle are mentioned by Gilb (Gil74).
2. Tests to detect errors.
The results of programs have to be tested, in order to guarantee reliability for these programs. A stand by module will only be invoked after the main processing module fails its test. (An alternative is to do majority voting as is used, for example, in the space shuttle computers (Sk176).) Hardly any work at all has been done so far on the design and implementation of run time tests which are built in programs to guarantee that the programs are performing correctly if their tests do not fail.
3. Executive programs for restart and recovery.
After an error has been detected, the module that failed to perform correctly is recovered and the stand by module is invoked (or the module is retried after certain errors have been corrected). Any system that uses recovery to provide fault tolerance needs such an

executive program. Most of the work in this area has been done for data base systems (see the survey given in a subsequent chapter) or for systems that implement reconfiguration or graceful degradation (Avi71), (Wen72), (Bor72), (Sk176).

An example of a system, which has many of the features of fault tolerant systems (even though these features were included for obtaining efficiency) has been described by Lampson (Lam75). The system described makes use of "hints", for example to find a file. A hint is information provided solely to improve efficiency of the implementation. Whenever a hint is used, it is checked against some "absolute" (a non redundant information item which is presumed to be always correct) to confirm its continued validity. If a hint appears to be wrong then an alternative and less efficient algorithm is used to do whatever the program that initially used the hint wants to do. Several schemes for fault tolerance are used in a number of systems at present, such as input validation schemes or schemes designed to tolerate hardware faults. However, it is only recently that efforts have been undertaken to extend the fault tolerant approach to include design faults. The three forms of software redundancy described above have been incorporated in a scheme facilitating (a certain degree of hardware and software) fault tolerance, designed and implemented by the project team on highly reliable software at the University of Newcastle upon Tyne, England (Hor74), (Ran75). The scheme is called the recovery block scheme.

A recovery block is a set of alternative program blocks, each of which is an alternative implementation intended to satisfy the same abstract specification. Associated with a recovery block is an acceptance test, which is a piece of program which tests the results of an alternative against the abstract specification (which is the same for all alternatives of a recovery block). The first alternative will be invoked when the recovery block is entered. When the alternative has been executed, the acceptance test will be evaluated. If the acceptance test is successful (i.e. the results conform to the specification) then the recovery block will be exited. If it is not successful then the effects of the first alternative will be undone and the next alternative invoked. In that case the first alternative is said to be backed out. (The term recovery is used for data, the term backing out is used for programs and processes.) This sequence will be repeated until one of the alternatives passes the acceptance test; if none of the alternatives passes the acceptance test then this causes an error to occur. If an error occurs (an error may occur for various reasons such as an exhaustion of recovery block alternatives or a failing of the acceptance test) within an alternative of a recovery block then this alternative is backed out (and the next alternative is

invoked without performing the acceptance test). Recovery blocks can be nested and if an error occurs during the execution of the last alternative of the outermost recovery block or if this alternative fails its acceptance test, then the program in which this recovery block is used has to be aborted.

The term "recoverability" will be used in this thesis for the kind of recoverability provided by the recovery block scheme. The recovery and recovery techniques discussed in the present thesis are based on these three forms of redundancy mentioned above. Since the recovery block concepts appear to have all of the features for fault tolerance as discussed in this thesis, recovery is discussed in terms of recovery blocks. Recovery blocks provide a very convenient forum for discussing the recovery techniques of this thesis. However, the work on recovery presented in this thesis stands on its own and need not be incorporated in a system providing recovery block structures.

1.1 The goal of the thesis

The main results of the research work on fault tolerant systems in Newcastle so far consist of:

- * The recovery block scheme.
- * The design and implementation of a mechanism and machine architecture (the fault tolerant interpreter) which provide recoverability for program variables inside recovery blocks.
- * Techniques, rather limited at present, for extending these ideas so as to provide recoverability for systems involving asynchronous processes.

The research on recovery blocks and mechanisms to implement them has mainly dealt with the problems of providing recoverability for simple variables, for example integer variables, real variables and boolean variables in user programs.

The goal of the present thesis is to examine how the recovery block concepts and principles can be used, generalized and implemented for the construction of recoverable multi-level systems and recoverable abstract (and complex) data types such as files.

The levels in the recoverable multi-level systems considered are levels that provide new recoverable types to higher levels. A type (which is defined by the operations on

the type and the mapping function mapping the type onto lower level types) and the recoverability of that type (which is provided by the recovery mechanisms) are implemented by the level providing the type. The levels can be backed out themselves and can be called on to recover states of objects of the recoverable types they provide, used in higher levels.

This thesis concentrates on the problems, requirements and constraints of providing recovery for objects in multi-level systems and for complex data types. Different approaches and a number of techniques and mechanisms that can be used to implement recovery for multi-level systems and complex data types are described and compared. A prototype system implementing many of the techniques and mechanisms described, has been built and evaluated.

Mechanisms and system structures to facilitate the multiple storage of programs and data, and the incorporation of executive programs for restart and recovery, are described. The problems of error detection (i.e. the construction of good tests) are largely ignored; a little discussion on some initial work done is given at the end of this thesis.

Recovery between interacting processes will not be dealt with at all. The problems of providing recovery for interacting processes have been described elsewhere (Ran75), (GiS76), (Ast76), (RLT77) and appear to be a major topic for research in their own right. A major problem is that if processes interact somehow and one process needs to be backed out, then a "domino"-effect (Ran75) may result. If parallel processes do not interact, but instead only share data such that these processes do not need to run concurrently, but could progress independently, then a mechanism that would prevent the interactions would avoid the recovery problems for interacting processes. A resource locking scheme, as for example described by Gray (Gra76), could be used to achieve this. If processes have to interact in order to make progress then something like conversations, as defined by Randell (Ran75), will be needed. If several processes need to interact to progress then, in fact, they co-operate to perform a task, or do subtasks of a bigger task. A conversation incorporates the processes working on one task and prevents other tasks from reading or writing objects updated by this task. The checkpoints and commitment points of processes involved in a conversation are synchronized, such that if one process fails all the processes involved in the conversation can be backed out to undo their parts of the task performed so far. In recovery block terms this will mean that entering and exiting of recovery blocks is to be synchronized for interacting processes. A full discussion on these problems, in very general terms, is given elsewhere (RLT77).

1.2 Summary of thesis

A description of how recoverable multi-level systems can be constructed by distinguishing between recoverable and unrecoverable types forms the first major topic of the thesis. A special mechanism that copes with the problems of operating on unrecoverable types inside recovery blocks has been developed. The implementation of a prototype system which incorporates the most important techniques is described.

The second major topic is the presentation of techniques that can be used to provide recoverability for complex data structures. A recoverable filing system, which is incorporated in the prototype system, is used to illustrate the problems and techniques discussed.

However, the thesis first gives, in chapter two, a comprehensive survey of the techniques used in filing systems, data base systems and operating systems for recovery, restart and the maintenance of consistency. Examples of existing systems using the various techniques and descriptions of the ways in which these techniques are implemented, are given.

Chapter three deals with the requirements and constraints of the architecture of a single level in a recoverable multi-level system. Several possible solutions, designs and mechanisms satisfying the requirements and constraints are described. An implementation of a recoverable two-level prototype system is described for illustration.

Chapter four describes the problems of providing recoverability for complex global data structures. A number of recovery techniques are described and compared. Several mechanisms that assure consistency and recoverability at any time have been found and are described. The recoverable filing system that is implemented as the second level in the prototype two-level system is described and used to illustrate the problems.

Chapter five gives a cost analysis of the recoverability provided in the prototype recoverable system. The three criteria used for the cost analysis are a) the extra execution time needed for programs that make use of the recoverability provided, b) the impact on the sizes of the system programs due to the provision of recoverability, and c) the extra data space necessary to provide the recoverability.

Finally chapter six gives some directions for future research and presents the conclusions of this research. One

of the main conclusions is that the distinction between recoverable and unrecoverable types leads to a better understanding of the problems and issues involved in recoverable multi-level systems. It is shown that in order to implement recoverability in multi-level systems with efficiency and flexibility, the interfaces of the system should provide both recoverable and unrecoverable types. Another main conclusion is that three factors are of major importance if recovery for complex data structures is to be provided with "reasonable" efficiency and if consistency of the data is to be guaranteed after a crash:

- 1) The structure used to implement the complex data.
- 2) The redundant information maintained to make recovery possible.
- 3) The way in which the data structures are updated.

2.0 A SURVEY OF TECHNIQUES FOR RECOVERY AND CRASH RESISTANCE

2.1 Introduction

This survey describes techniques and utilities that can be used for recovery, crash resistance, and maintaining consistency after a crash. The techniques and utilities describe how data structures should be constructed and updated, and how redundancy should be retained to provide these facilities.

This survey deals with recovery for data structures and data bases; not with other issues that are important when processes operate on data, such as locking, security and protection (Lin76).

For the purpose of this survey the notions of filing system and data base system are treated as synonymous. The definition of the notion of data base given by Martin (Mar76) is used here, and is: "a data base is a collection of interrelated data stored together with controlled redundancy to serve one or more applications in an optimal fashion; the data are stored so that they are independent of programs which use the data; a common and controlled approach is used in adding new data and modifying and retrieving existing data within the data base".

A data base may consist of a number of files. A file is a logical unit in the data base, used to group data. The data that can be retrieved by users from the data base forms the information in the data base. If some of the data stored in the data base cannot be retrieved, then some of the information put in the data base is lost.

The notions of data base, file and information are logical. In physical terms the data base is held on secondary storage. Secondary storage is permanent storage space, which is separate from the computer and retains the data base whether it is on-line (mounted on a storage device unit and readable by a computer) or not. Secondary storage consists of records, which are the smallest (fixed length) accessible units.

A data base is an abstraction of secondary storage provided to the user by a data base system. The data base system implements the user operations on the data base and implements the data structures on secondary storage. Objects are the substructures from which these data structures are built. Examples of objects are: a logical

record, a header of a file, a linked list of pages or records and an entry in a directory.

Users may add data, delete data and update data in a data base. A data base is in the correct state if all the information in the data base consists of the data which is stored in the data base by users, and is in its most recent state (after the last updates), minus the data deleted by users. A data base is in a valid state if all the information in the data base consists of data stored in the data base by users. The data base is in a consistent state if it is in a valid state and the information held satisfies the users' consistency constraints. It is assumed here that a correct state is also consistent. The notion of consistency will have to be a well-defined notion for every data base. Different sorts of consistencies (possibly at different levels of abstraction) or degrees of consistencies (Gra76) may be defined. No exacter definition of consistency will be given here, since it is assumed that the notion may be defined differently for different data bases.

For example, suppose that a user maintains a source file and a text file (produced by a compilation of a source file). The data base will then be in a correct state if the most recent source and text files are available. The data base is will be in a valid state, for example, if a source file and a text file, but not necessarily the most recent ones, are available. The data base may be in a consistent state if a corresponding source and text file are available.

A failure of the system occurs when that system does not perform its service in the manner specified (MeR76). Recovery is the restoration of the data base after a failure to a state that is acceptable to the users. The notion of "acceptable" is different for different environments; in general it will be correct, valid or consistent. There are many sorts of failures and therefore many sorts of recoveries possible. A recovery technique provides recovery from certain kinds of failures. One data base system may use different recovery techniques to provide recovery from different failures. Typically, if several recovery techniques are used then the sets of failures from which they provide recovery are subsets of each other. The recovery technique which provides recovery from the biggest set of failures is, in general, the least efficient and least used technique, and may involve the loss of some information (i.e. the data base may not be restored to the correct state). The recovery technique which provides recovery from the smallest set of failures, is usually the most efficient technique involving none or little loss of information.

A recovery technique maintains recovery data to make recovery possible. A recovery technique provides recovery

from any failure which does not affect the recovery data nor the mechanisms used to maintain these data and to restore the states of the data in the data base. A failure with which a recovery technique can cope is said to be a crash of the system with respect to that recovery technique in case no normal continuation of processing is possible without using the recovery technique to cope with the failure. A failure with which a recovery technique cannot cope is called a catastrophe with respect to that technique.

A system using three recovery techniques could, for example, consist of the following subsystems (see also Fig.2.1):

- A) The data base system without any recovery techniques.
- B) A plus a recovery technique that uses built-in redundant pointers in data structures to be able to recover from certain failures causing particular errors in the data structures.
- C) B plus a recovery technique that does not use built-in redundancy in data structures, but maintains backup copies of (parts of) the data structures.
- D) C plus a recovery technique that keeps a complete backup copy of the data base on a separate device.

These systems could be built using an approach similar to the so-called safe programming approach described by Anderson (And75). The bigger the damage the cruder the recovery technique used. Restoration of the correct state is most desirable and can be done, say, in B. However, if the damage is such that recovery in B is not possible then the restoration to a consistent, but not the correct state may be the only alternative in C, and so on.

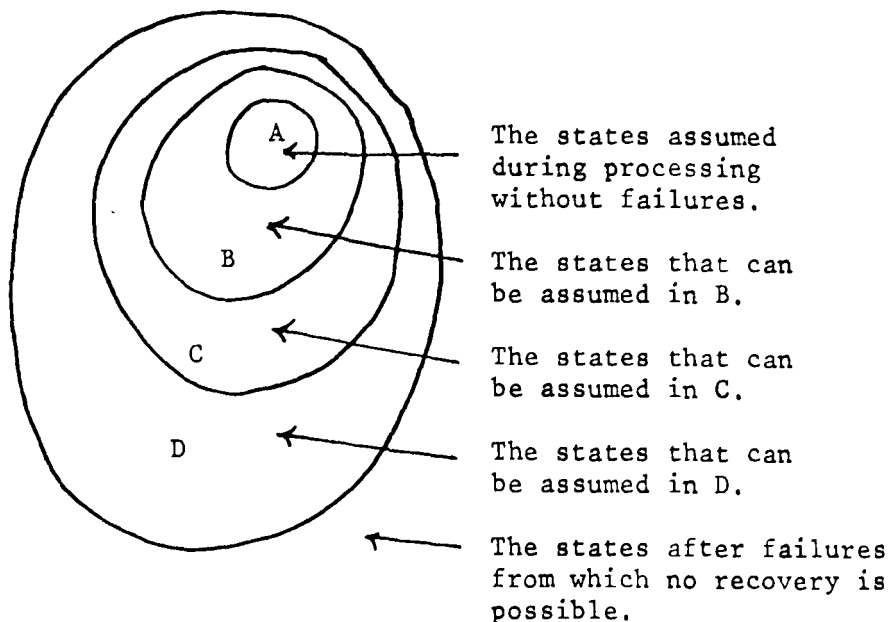


Fig.2.1, The state space for a data base system with several recovery techniques, coping with subsequently larger sets of failures.

No recovery technique, nor any series of recovery techniques, will cope with every possible failure. Many different kinds of recovery techniques have been developed, each with its own particular advantages and disadvantages. These recovery techniques are therefore used to cope with different kinds of failures in different environments. This survey describes recovery techniques known and used at present; an attempt is made to make this survey complete by categorizing these techniques. First the categories of recovery techniques considered are briefly described, the kinds of recoveries they provide and the relationships between the techniques are given. Then the different techniques are defined and described in detail, and the purposes for which they can be employed and existing systems using them, are described. Finally some conclusions are drawn and some developments in the techniques in recent years are identified and anticipated trends are described.

2.2 The categorization of the techniques

The different kinds of recovery for a data base considered are:

1. Recovery to the correct state.
2. Recovery to a state which existed at some moment in the past (i.e. a checkpoint).
3. Recovery to a state which is effectively a previously existing state (such as a set of previously existing states of logically independent files, which may not have existed at exactly the same moment in time).
4. Recovery to a valid state.
5. Recovery to a consistent state.
6. Crash resistance. If the system is always left in a state from which normal continuation of processing is possible after a failure of the kind a particular recovery technique used copes with, then that recovery technique is said to provide crash resistance. Crash resistance is different from other kinds of recovery in the sense that the other kinds of recovery involve explicit state restoration, while crash resistance does not. Crash resistance is provided by techniques used during normal processing, which ensure that state restoration is done implicitly at the time of a failure; no special actions are required. The differences between crash resistance and other kinds of recovery are fully explained in the descriptions of the various recovery techniques in this survey. The notion of crash resistance is defined as vaguely as the notion of consistency was. For example, different degrees of crash resistance could be defined. However, making such distinctions is not of importance for the purpose of this survey; the definition of crash resistance can be tied in with the definition of consistency given above.

A checkpoint is a state which existed in the past, which can be restored, and may or may not have been established explicitly, for example by taking a snapshot. Checkpoints are used by recovery techniques of kinds 1, 2 and 3. Checkpoints can be established for files or the whole data base. The creation of a checkpoint is called checkpointing. If a checkpoint is established explicitly then this implies that a backup version is created. A backup version is a complete copy of the checkpointed file or data base.

The term backing out is related to processes or transactions. A process is backed out if all the effects of the operations performed by that process are undone. This means that only files affected by that process are restored. Backing out of some processes may be required, for example, to resolve a deadlock, or to undo the operations of a failing process. Backing out is a special sort of recovery

of kind 3.

The survey of techniques described is intended to be complete, although the categorization is the responsibility of the author and definitely not the only possible one. The seven categories into which the techniques used for recovery, restart and maintenance of consistency are divided in this chapter are introduced below and their relationships briefly described.

* Salvation program.

A salvation program is run after a crash to restore the system to some valid state, without using recovery data. This program is needed after a crash if other recovery techniques (using recovery data) fail or are not used, and if no crash resistance is provided. This program scans the data base after a crash to ascertain the damage and to restore the data base to some valid state. This program rescues the information that it can still recognize in the data base after a crash.

* Incremental dumping.

Incremental dumping involves the dumping of updated files onto archive storage (usually tape) after a job has finished or at regular intervals. Incremental dumping provides checkpoints for updated files. It provides backup copies of files which can be restored after a crash.

* Audit trail.

An audit trail records, "who did what to which files, and in what sequence". An audit trail can be used to restore files after a crash to the state they were in at the time of the crash, to back out processes and for certifying the integrity of the system (i.e. verifying that rules and laws are obeyed). An audit trail keeps track of operations performed by processes, whereas incremental dumping, for example, keeps track of updates made on files. An audit trail thus also provides the means to back out a process while incremental dumping just provides the means to restore files to previous consistent states.

* Differential files.

A file can be implemented such that it consists of a main file, which is kept unchanged, and a differential file to register the alterations made to the file. At regular times the main files must be merged with the differential files, after which the differential files are empty again. Records in the differential files can be stored with the process identifier, a time stamp and other identification information to provide special facilities, such as recovery or crash resistance. In a sense the differential file could be regarded as a

special audit trail in a system where the forming of the audit trail is the only effect of file updates. So facilities similar to audit trail facilities are provided. Since the differential file is part of the logical file it may be used to provide other facilities such as crash resistance.

* Backup and current version.

The files containing the present values of existing files form the current version of the data base. Files containing previous values (values that existed earlier in time), which form a consistent data base, comprise a backup version of the data base. Backup versions can be used to restore files to previous values.

* Multiple copies.

More than one copy of each file is held. The different copies always have the same value, except during update of course. An inconsistent copy can always be recognized either by voting or an "update-in-progress" bit. So after a crash during update the most recent consistent state of a file can always be retrieved. Consequently, this technique provides crash resistance and may be used to detect faults if the different copies are maintained on different devices or by different processors and the validity of the data can be checked.

* Careful replacement.

Under the careful replacement scheme no part of a data structure is ever updated "in place". Altered parts are instead copied into new objects. Therefore during update there are two copies of the data structure which overlap in identical parts (objects). One copy contains the pre-update value, the other one is in a transition to a new value. At the end of the update the copy containing the pre-update value is destroyed (only objects which are not shared with the other copy are released). The difference with all of the other methods is that two copies exist only during update. The two copies have the same structure (so it is not a differential file scheme), but will overlap in identical objects that form part of the structure. The technique is used to provide crash resistance. The pre-update copy will always be available after a crash during update, the other copy will then be lost.

A cross-reference table showing for which kinds of recovery the recovery techniques described above, can be used is given in Fig.2.2.

	1) correct state	2) previous state	3) eff. prev state	4) valid state	5) consistent state	6) crash resistance
salvation program				*	*	
increment dumping			*			
audit trail	*	*	*			
diff. files		*	*			*
backup current		*	*			
multiple copies						*
careful replacem.						*

Fig.2.2, A cross reference table indicating for what purposes the various recovery techniques can be used.

From the description of the techniques and Fig.2.2 the following relationships between the techniques are apparent:

- * The differential file technique makes incremental dumping very easy to implement. Incremental dumping, in general, copes with failures the differential file technique cannot cope with. Thus the two techniques may complement each other very well.
- * The audit trail technique is an alternative technique to differential files, careful replacement or multiple copies, that can be used to provide the facility of restoring the correct state after a crash. Audit trail is therefore hardly ever used as a recovery technique in one system together with differential files, multiple copies or careful replacement.

- * It will be shown in this survey that multiple copies and careful replacement may be used as alternative techniques, but also to complement each other to provide crash resistance against similar or the same set of failures.
- * Also the incremental dumping, the audit trail, the differential files and the backup current version techniques can be used as alternative techniques or to complement each other to provide recovery from different failures.
- * The salvation program is a recovery technique which is used if all other techniques fail. It cannot put the data base back to a previous state, because it, in fact, rescues what is left in the data base.

These seven techniques could be said to provide recovery, crash resistance and maintenance of consistency in one of the following three ways:

- * The way in which the data is structured.
The multiple copies, differential files and backup techniques, are techniques to structure the data or data base such that the required facilities are provided.
- * The way in which the data is updated and manipulated.
The careful replacement technique is a crash resistant way of updating complex data structures. It will be shown in this thesis that this also sets special constraints and requirements for the data structures.
- * The provision of utilities.
The salvation program, incremental dumping and audit trail are utilities which have nothing to do with the way in which the data is structured or updated. They could be regarded as external utilities which can usually be added to any data base system without great difficulty.

The division of the techniques in the three groups could be misleading in cases where techniques complement each other or are alternatives. The seven techniques will therefore be discussed separately. The categorization of the techniques, however, is partially justified by distinguishing the three groups of techniques. For example, the backup and current version technique is in a different group than incremental dumping. It is therefore justified to distinguish them as two different techniques.

The seven techniques are described in detail in the following seven sections. Examples of systems using the techniques are given for illustration.

2.3 Salvation programs

A salvation program in a data base system is used after a crash to restore the data base to some consistent state. The salvation program tries to restore the state of the data base as it was before or at the time of the failure. However, in general some files or data may be lost. A salvation program basically scans through the data structures and tries to reconstruct the data base or restore consistency, possibly at the cost of deleting some files or data.

A salvation program is needed after a crash if the data kept on secondary storage is not kept in a consistent state all the time and other recovery techniques to restore all the data to a consistent state are not available or cannot cope with this crash. Otherwise no salvation program is needed.

One of the reasons for the data on secondary storage to be left in an inconsistent state after a crash could be, for example, the loss of buffers kept in main storage. The reasons for having to delete some inconsistent files could be (SmH72): i) the violation of standard error checks on reading a file, ii) a conflict because the same storage appears to have been assigned to more than one file, iii) a conflict (e.g. on the file length) determined from redundant information (e.g. from a file header).

A system may use buffers for the data base (data buffers) and for audit trail tapes (audit trail buffers). After a crash there is in general no way to tell which updates recorded in the audit trail have been written to the data base and which were still in data buffers, and similarly which updates made in the data base are recorded on audit trail tape or were still in buffers (GiS76). Using the audit trail to restore the data base to what it was at the point of failure may therefore not be possible. Several systems such as IMS (IBM) or the CMIC system (GiS76) first use a salvation program which tries to rescue the contents of the buffers in main storage, after a failure, in order to close the audit trail tapes properly. However, main storage may be lost in which case restoration of the correct state is not possible.

A system in which a salvation program is of great importance is the HIVE system (Tay76) (here the program is called the recovery procedure). The system consists of a fixed number of virtual processors (VPs) which are assigned permanently to execute particular functional application programs in a cyclic manner. A processor cycle, performing such a particular function, is triggered by a message received from another VP or from outside the network of VPs.

Capabilities for the necessary code and permanent data areas are given to the VPs at system build time and the message routes between VPs are also set up permanently. Basically only the files for which permanent capabilities have been created at system build time can be restored after a crash. However,

- a) Transaction checkpoints can be made by writing the data of each transaction into a common, permanent, safeguarded checkpoint file, which can be accessed and recovered after a crash.
- b) Files may be created dynamically and capabilities for them may be put in special files called cap-files.

The recovery procedure run after a crash restores in core the read-only core image, which also contains recovery code. The main task of the recovery procedure is to perform a garbage collection in the the data base by scanning all files. Files for which capabilities are kept in cap-files are processed first. For each version of each file (several versions of each file are maintained) the check sums are evaluated to detect partially updated and corrupted pages, and where possible the appropriate updating and backtracking from other versions is carried out. (At most one version can be corrupted after a crash, and if so, the corrupted state will be detectable using the check sums. Only a catastrophe, such as an fire in the computer centre, may corrupt more than one version.) The checkpoint files are used to initialise the transactions represented in those files.

Other systems in which a salvation program is used to recover the disk contents after system failure have been described, for example, by Lockemann and Knutsen (LoK68), Daley and Neumann (DaN65) (salvage procedure), Fraser (Fra69) (start-up procedure) and in the EMAS system (EMA74). (See also the surveys in (Ton75) and (Mas73).)

2.4 Incremental dumping

Incremental dumping is used to copy updated files onto archive storage (usually tape); it checkpoints files that have been altered. Incremental dumping will normally be done after a job has finished and can also be done at regular intervals, while continued use is made of the files, in order to get checkpoints more frequently. The incremental dump tapes can be used to bring all the files to their last consistent state again after a crash has occurred. Jobs completed before the crash, will not have to be rerun. All of the updates performed by jobs running at the time of the crash will not be restored completely by the processing of the incremental dump tape after a crash; the effects of these updates may be restored partially. Fraser (Fra69) gives a very good description of the technique as used at

Cambridge, which makes complete copies of updated disk files every twenty minutes.

In the MULTICS system (DaN65), for example, all of the disk files updated or created by the user are copied when he signs off, and every N hours all newly created or modified files which have not previously been dumped, are also copied to tapes.

The EMAS system (EMA74) provides an automatic checkpointing facility for files. Files are part of the user's virtual memory and cannot be accessed through the paging mechanisms until they have been connected (i.e. the virtual memory disk address mapping has been set up). When a user process is created its virtual memory space is created, initialised and copied to disk. When the process is run the working set is in core and pages are transferred back and forth between core and drum. A page may be forced to disk, because the drum gets full or the process becomes dormant again. If a page is forced to disk then all of the updated virtual memory pages are forced to disk at the same time. This mechanism is required by the so-called consistency rule in the EMAS system. Therefore a suitable restart copy of the virtual memory of a process (which includes the files) is provided on the disk. The problem of having inconsistencies between the state of the process and the states of its associated files are avoided, because the filing system uses the resources provided by the paging system. The paging system assumes complete responsibility for maintaining a consistent backup copy of all of the state variables of the process (including files). Consequently, if the consistency rule is always obeyed then automatic checkpointing is provided.

Incremental dumping can be done as a part of an audit trail scheme (Mas71), (Ran70), (Mas73). An audit trail only gives the changes made to files from given states onwards. These states are redefined regularly for reasons of efficiency (in order that audit trail journals do not become too long). For example, in a system described by Wimbrow (Wim71) files are dumped when they have to be reorganized, because they become disorderly as a result of the operations performed. In the CMIC system (GiS76) all files are checkpointed regularly at moments when no user has the data base open.

Another scheme used in System R (Lor77), works as follows (see also Fig.2.5). Segments (which are similar to what we called files) consist of page tables with pointers to the data pages. Associated with each pointer in the page table are three bits: a shadow bit, a cumulative shadow bit and a long term shadow bit. When a segment is updated a backup and a current copy are maintained (in a way which will be described in one of the following sections). For

every page which is updated during a transaction, the shadow bit and cumulative bit are set in the relevant page table of the segment. When the current state of the segment is saved (i.e. replaces the old copy) at the end of the transaction, then the shadow bits are switched off and the old pages (of the backup version) which are replaced by the new versions of the current copy, are released. Checkpoints of all the segments are taken regularly. This involves the copying of all the page tables for which at least one cumulative shadow bit is switched on, the cumulative bits are copied into the long term bits and then switched off, and a so-called process P is started. Process P copies onto tape all of the pages for which the cumulative shadow bit is on at checkpoint time. The long term checkpoint bits are used to make sure that subsequent saves will not release the pages before P has copied them.

A special checkpoint file is used in HIVE (Tay76) to checkpoint transactions. Information put in the checkpoint files can be recovered after a crash to restart those transactions. Individual transactions can also be reprocessed selectively using this checkpoint file.

2.5 Audit trail

An audit trail records "who did what to whom, and in what sequence" (Bjo75), by keeping track of all of the operations performed. All the relevant information about the operations is registered in the audit trail, such as: the effects of the operations, the times and dates at which the operations took place and the identification codes of the user (or user program) issuing the operation.

Audit trails can be used for different purposes such as:

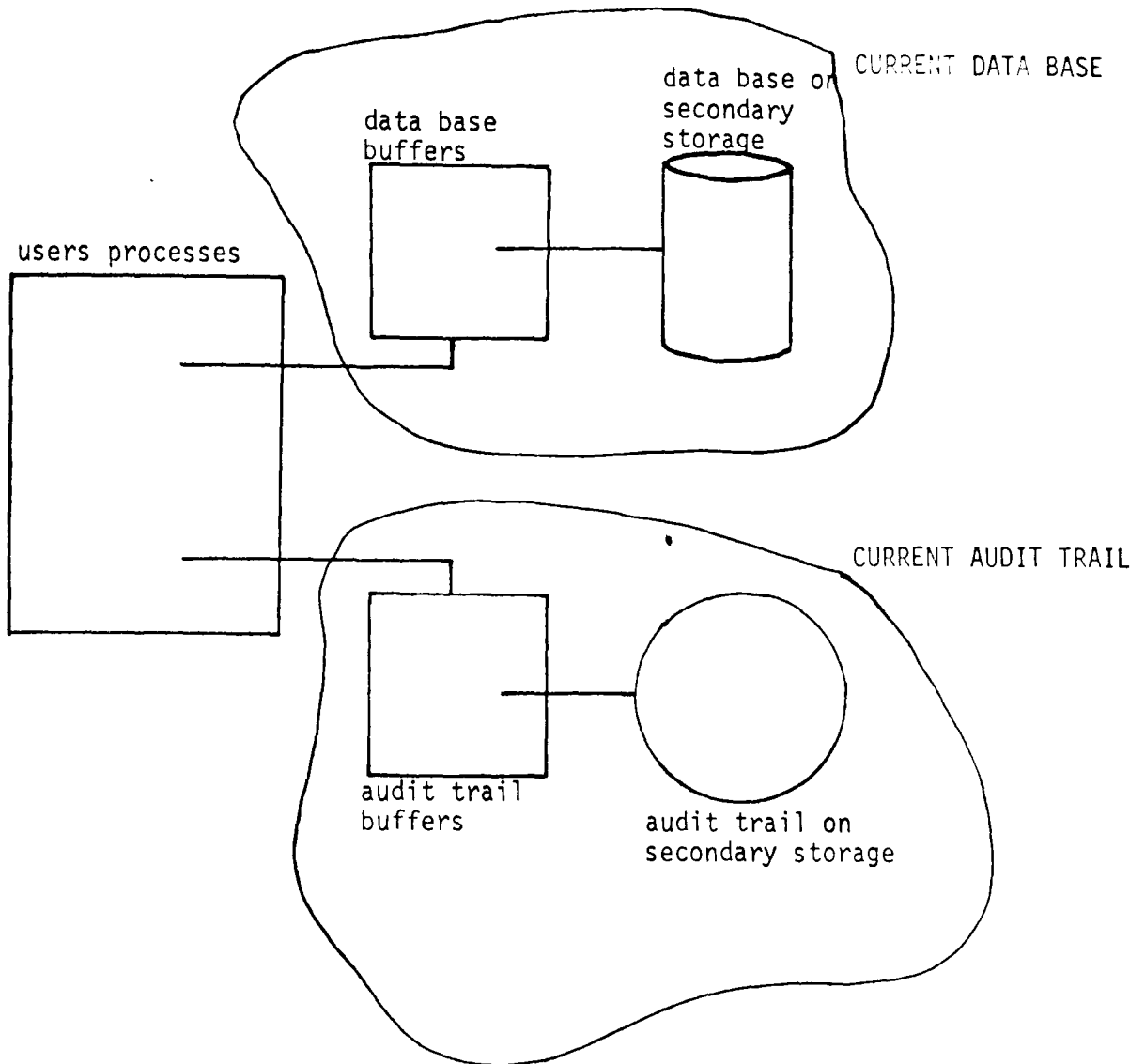
1. Crash recovery.
Backup versions of files can be re-installed and an audit trail can then be used to restore the files to what they were at the time of the crash (Cur77).
2. Backing out.
If a system crashes (without damaging secondary storage) the files affected by the processes running during the failure can be restored to what they were before those processes started. In other words the audit trail can be processed backwards for backing out.

3. Certifying the integrity of the system by providing means for verifying that rules and policies dictated by laws, business agreements, etc., are being followed by the application (Bjo75).

It is for this reason that Bjork concludes that audit trail will be the major integrity tool for shared data usage for the late 1970s and beyond. This could, however, well prove to be a rather controversial conclusion. For example, differential files combined with the incremental dumping technique could provide the same facilities, although the provision of integrity verification by this combination of techniques depends on how the differential file is implemented and on the requirements for verification.

Recovery techniques as described, for example, by Fraser (Fra69) and often used for filing systems, may, as described in a previous section, cause the loss of the effects of the most recent operations performed on the filing system. Incremental dumping, as used by Fraser, checkpoints files at regular intervals. The effects of operations performed on files since their last checkpoint was made, will be lost. This may not matter in many operating systems, because jobs can be resubmitted or operations can be redone. However, for systems where updates are made online from different sources, such as in banking or airline reservation systems (Mas71), (Ran70), (Wim71), (Ton75), this may be impracticable. One cannot afford to lose any update in the event of a failure in such systems. In systems like these an audit trail, kept say on tape, can provide a solution. Before a transaction is done on a data base an audit trail entry is written onto tape. The writing of audit trail entries must normally be done without the use of buffers (Wim71) to make sure that crash recovery is possible at any time; the use of buffers (see also Fig.2.3) may lead to inconsistencies between the data base and the audit trail (GiS76). (However, another possibility would be to salvage the buffers from main storage after a system crash, thus making possible the proper closing of the audit trail tape. This is, for example, tried, but not always successful, in IMS (IBM) and in the CMIC system (GiS76). The salvaging of the buffers is not possible in case the contents of main storage is lost after the crash.)

The audit trail can be used to back out a process. This process may have interacted with another process in such a way that that other process will have to be backed out. The audit trail can be used to back out that other process which may have interacted with again some other process, and so on. Thus using an audit trail to back out unfinished transactions performed by interacting processes, leads in general to a so-called domino effect (Ran75) (GiS76) (Cur77). A locking scheme, as used in System R (Ast76), may avoid these problems by making these interactions



The current data base is always consistent with the current audit trail, however the data base on secondary storage is, in general, not consistent with the audit trail on secondary storage.

Fig.2.3, A general data base system using audit trail.

impossible. (Practicable only when the interactions are accidental, rather than deliberate.) This will make the undoing of partially finished transactions possible using an audit trail (or a "log" as it is called in System R). Another solution would be to checkpoint all files while no user is active (GiS76). This would always stop the domino

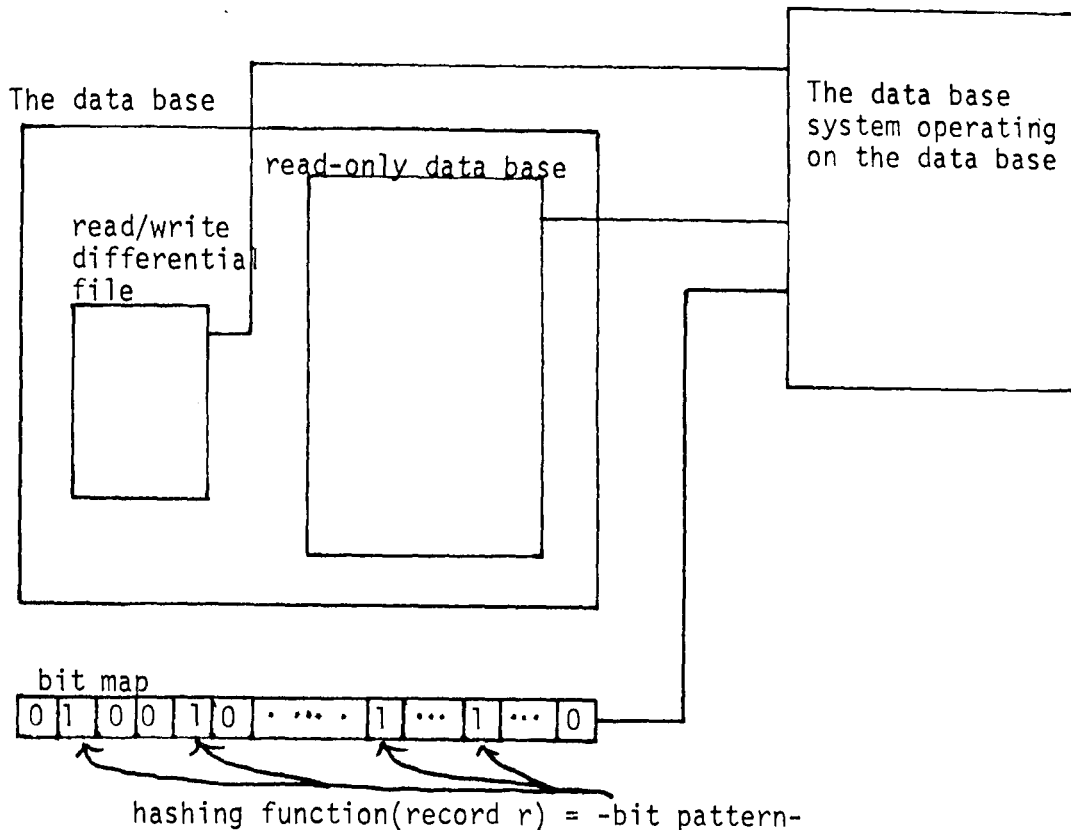
effect at that point. However, such occasions may occur too infrequently for this scheme to be much help, and frequent forcing of all users to become inactive may be impracticable.

Audit trail schemes can appear in many guises. For example, in a system described by Lampson and Sturgis (LaS76) so-called intention lists are used. An intention list specifies the operations to be performed by a processor. A processor, which is a node in a network, may receive an intention list, containing the specifications of the operations to be performed on its local data base. Intention lists, like the audit trail used in the CNIC system (GiS76), can if kept, be reprocessed if processing is interrupted without backing out the interrupted process. Intention lists, once received and accepted, cannot be lost unless a catastrophe, such as a head crash, occurs, because they are stored on disk at a fixed place known to the system, and are not altered when processed. So unless the processor breaks down and is never repaired the operations specified in the intention list will always be done. An intention list could be regarded as (an entry) in an audit trail. The difference between intention lists and audit trails is that audit trails are made as a result of issued data base updates, intention lists are created first. As far as processors processing intention lists are concerned, they could also be regarded as the programs issuing the update operations (like audit trails when they are processed, not when they are created). During crash recovery or backing out, the intention lists and audit trails are not different.

2.6 Differential files

Under the differential file (also called "change set") scheme the main files are kept unchanged until re-organisation. All changes that would be made to a main file as a result of transactions performed are registered in a differential file. A file as seen by the user is implemented as a fixed main file and a differential file. As a result of this the differential file will always be searched first in case data is to be retrieved. Data not found in the differential file is retrieved from the main data base. The most recent entry for a given record in the differential file must always be retrieved.

Severance and Lohman (SeL76) fully describe the technique and an efficient hashing method to implement it (see also Fig.2.4). A small associative memory in the form of a bit map, accessed by the hashing scheme, is used to reduce the probability of making an unnecessary search in



hashing function(record r) = -bit pattern-

The bit map suggests that record r is in the differential file, because the bits set in the bit pattern produced by the hashing function are set in the bit map.

Fig.2.4, A differential file technique using a hashing scheme.

the differential file. Severance and Lohman show analytically how to keep the probability of a filtering error (i.e. the bit map suggests that that a record is in the differential file while it is not, because the relevant bits are set to represent other records) low. They also describe the advantages of differential files for the provision of recovery, integrity, the implementation of (incremental) dumping schemes and other general advantages such as the simplicity of software. Another advantage claimed is the possibility to perform queries which do not need the exact values of all files; such queries get a

suitable (but out of date) view of the data without locking out the update transactions.

The disadvantages of the approach using differential files are (Lor77):

- * An access to a data element must first search the differential file; if the data element is not among the modifications then it must be fetched from the data base. However, Severance and Lohman show that, using a hashing scheme, this problem can be almost completely overcome. They also show how to construct a good hashing scheme for particular systems.
- * Eventually a merge of the modifications and the main data base must be performed, and this operation can be time consuming. This will certainly be a big problem if the system needs to be available without interruption.
- * Since an update can affect an element which has already been modified, the organization of the differential files must accommodate such cases. If hashing schemes described by Severance and Lohman, are used or a scheme similar to the one described by Rappaport (Rap75) in a system called VADIS, then this problem may be avoided.

Differential files are, for example, used in the VADIS system (differential files are called MODFILES). For every file in the system there is a MODFILE. The system has been developed to facilitate recovery after power failure; this has been implemented by providing resistance from crashes due to power failure. So after a power failure the system can restart as if nothing had happened. Uncompleted transactions before the failure are not undone, rather the effects of these operations are ignored using the MODFILES and a TRNSDONE file as follows:

- * Each entry in a MODFILE has a header with: record type, transaction code, pointer to previous modification of the same record, time, transaction number and some other identification codes.
- * There is a file TRNSDONE which contains the numbers of the completed transactions.
- * For every record fetched from a MODFILE the transaction number is compared with the TRNSDONE numbers and the current transaction number.

- * If the number is neither in TRNSDONE nor is the number of the current transaction, then the previous version of the record is taken (the one pointed to by the retrieved entry from the MODFILE, or in the main file), because this means that the record was put in the MODFILE by an uncompleted transaction before a failure.

Differential files are used in a system, described by Titman (Tit74), for both efficiency and reliability reasons. The way in which ordinary files are kept makes insertions or deletions very expensive. In Titmans system the files are binary relations which are stored, in a highly compressed form, in fixed length blocks. Elements are identified by a block number and the sequence number of the element in the block. An insertion or deletion requires the complete re-organization of the file giving the elements new identifiers. So, for efficiency reasons, an "add set" and a "delete set" of inserted and deleted elements are kept for each file. For reliability purposes a "change set" is also kept for each file, which is used to register the changed records. The "add set", "delete set" and "change set" together form the implementation of a differential file. The main files are kept on a separate device which is never written on except during re-organization. These files can be duplicated on tapes for recovery. Checkpointing is carried out by saving the add, delete and change sets.

2.7 Backup and current versions

Backup versions of files or data bases can be kept in order to make possible the restoration of the files to a previous state.

For example, many file-editors produce a complete new version of a file while a user is editing a file. The original file remains unchanged during the edit-session. The new version is a complete new copy; it is not achieved, for example, by using a differential file. If a user notices during the edit session that he has made a blunder, for example deleted 100 lines instead of 10, he does not replace the original copy of the file.

Incremental dumping (of current versions) can be used as a utility to maintain a backup version of a filing system or data base: altered files are copied, which can subsequently be used to update a backup version of the whole system or data base. This is done in the Cambridge filing system (Fra69), where two processes are used: one makes incremental dumps and the other creates backup versions of the system.

Similarly, complete copies of the data base can be made regularly in order to make possible the restoration of the data base to an earlier state. For example, in MULTICS (DaN65) a weekly dump is prepared of all files which have been used within the last M weeks plus all of the system files necessary to run the system.

An optimised version of this technique (see also Fig.2.5) is used in System R (Lor77) for so-called segments (synonymous to our notion of files). For each segment a page table is used to point to the data pages. There are two copies of each page table, which are identical when no transaction is in progress. If a page of a segment is altered during a transaction then its new value is put in a newly allocated page and the current version of the page table is updated to point to the new page. The backup version remains unaltered. At the end of a transaction the current version is copied into the backup version and the replaced or deleted pages are released. This releasing of pages causes the bit map used to indicate the free pages to be updated. Two copies of the bit map are maintained. A MASTER table points to the current map. The current bit map always reflects a consistent state of the system (i.e. all of the pages pointed to by the backup versions of the page tables). At the end of a transaction the MASTER table is then made to point to the up to date bit map. This scheme provides the possibility of restoring a segment to its last consistent state (held in the backup version) and of restoring consistency after a system failure (the operations of unfinished transactions, performed before the failure will be lost; these transactions will have to be restarted).

Physically completely separate backup and current versions of the page table provide logically completely different, but physically overlapping, backup and current versions of the segment under this scheme. The logically different versions of a segment overlap (physically) in their implementation where they are equal.

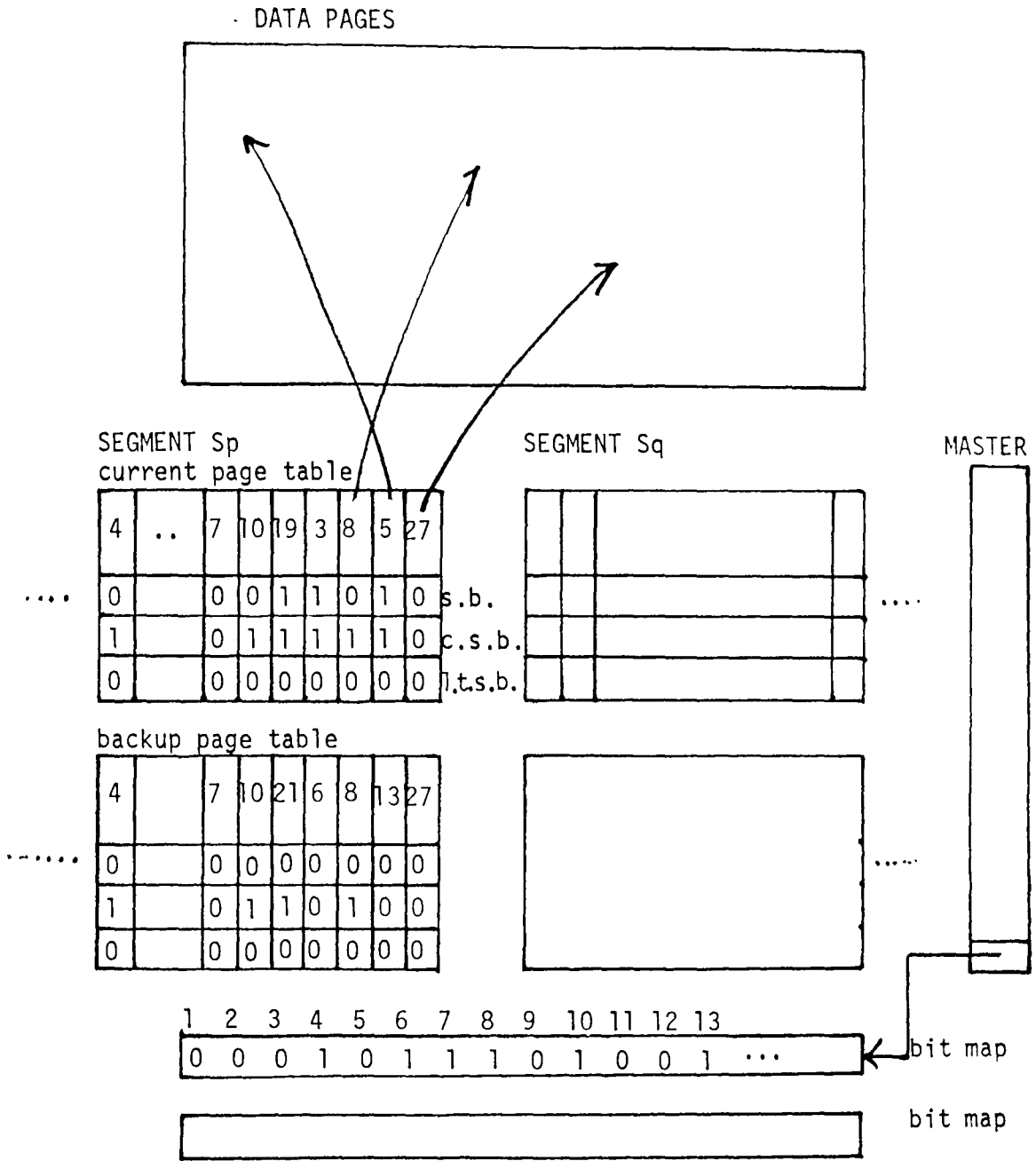


Fig.2.5, The implementation of segments in System R.

2.8 Multiple copies

The technique of multiple copies involves either:

- * Keeping more than two copies of the data so that they can be compared when needed. If a majority of the copies have the same value, then that value is taken. This technique is then called majority voting.

or,

- * Holding two copies with flags indicating "update-in-progress". An inconsistent copy (or suspicious copy) is always recognizably inconsistent, because of the flags used; if the system crashes during update the flag will still be set after the crash.

Except during update the multiple copies must always have the same value. If the different copies are updated by the same processor then a flag "update-in-progress" (sometimes called "damage flag" (Cur77)) is used if there are only two copies, in order to provide crash resistance. A consistent copy can always be retrieved after a system restart; this copy will either have the value it had before the update in progress during the crash, or the new value. The inconsistent copy can always be recognized as such and discarded. The keeping of more than two copies provides crash resistance. The use of two copies plus flags also safeguards against crashes. Majority voting may also be used to detect incorrectly performed operations, which is especially useful if different processors update the different copies. Faulty processors can then be detected and ignored or disconnected.

The important difference between the multiple copies technique and other techniques such as backup and current version, and careful replacement, is that with the multiple copies technique the different copies always have the same value, except during update. The multiple copies also exist all of the time and are implemented as physically different files which may not overlap. Schemes, for example, using different backup or archive versions, are therefore definitely not implementations of the multiple copies technique as described here.

Majority voting on data has been used extensively for space flight applications, such as in the space shuttle system where four computers are configured to receive the same input data and calculate the same outputs (Sk176).

The technique of two copies with flags is used in the provision of recovery for segments in System R (Lor77). A MASTER table is used to indicate which segments are open or

closed and which bit map (two copies are held) is the up to date one (see Fig.2.5). Two copies of the MASTER table, both containing the same information, are kept to ensure that if the system crashes while the MASTER table is being updated, always at most one copy will be left behind in an inconsistent state; the other copy either has the new state or the state the table was in before the update started. The copy that is in an inconsistent state can always be identified.

Similarly two copies of the MASTER-directory in the filing system of GEORGE 3 are maintained (New72), and two bits to make possible the distinction between a valid and invalid copy after a crash during update.

System HIVE (Tay76) maintains two read-write versions for every file. This provides one of the characteristics of a cycle (see section 3) which is that the local effects can be undone as long as the cycle has not yet finished. During a cycle one of these two versions is updated. At the end of the cycle this version is copied into the other version. The system knows which of the two versions is the one updated during a cycle. Crash resistance is therefore provided for individual files. Apart from this, a sum check is maintained for each version. This, generally, enables partially updated or corrupted files to be detected (this of course depends on the kind of crash, but it is the case for most likely kinds of crashes). In general two copies and two flags (bits) are sufficient to provide crash resistance. The flag indicates whether a copy is suspicious or not. If the two copies are updated immediately after each other, as in System R, then a copy is very likely to be inconsistent if its flag is set. In system HIVE, however, the two copies are kept on separate storage devices, so the check sum provides extra facilities: it makes the detection of incorrectly performed operations possible; in other words faults can be detected. System HIVE is one of the few existing systems in which more than one complete copy for every safeguarded file are maintained to provide crash resistance. One copy is updated during a cycle and the second copy is updated in pages which correspond to the changed pages of the first copy to reduce the overhead.

2.9 Careful replacement

This technique implies arranging that, as far as possible, no data structures are ever updated "in place". Instead updated objects (records, pages, disk blocks) that are part of a data structure, are copied into other objects and the updates are made there. The same is done for objects in the data structure which point to those objects. During

the update there will be two copies of the data structure, which overlap such that the same objects (for example pages) are used for both data structures, for those objects that have the same values in both data structures. One copy contains the pre-update value and the other copy is in a transition from that value to the new value and will only be in a valid and consistent state (from the users point of view) when the update is completed. Once the update is completed the copy containing the pre-update value is destroyed (only objects which are not shared with the other copy are released). Only during update are there two copies (which overlap in identical objects). If no update is in progress then the data structure contains the normal current value. This is different from the differential file technique where there is one copy all the time, which remains unaltered, and a differential file to register updates made to the file. (Careful replacement could be used to merge the main file with the differential file when merging is done.)

The important difference between the careful replacement technique and other techniques is that with careful replacement two "virtual" copies are held only during an update (or perhaps within a recovery scope specified by the programmer (Ver77)) to make the update or sequences of updates as safe as possible (i.e. reduce the chance of being left with a inconsistent copy or mutually inconsistent set of files, in the event of a crash).

This technique is fully explained by Gamble (Gam73) who describes a filing system in which this technique is used. Files consist of data pages pointed to by a tree of directory pages. A master directory points to each top directory page of the files. If files are updated using careful replacement then they can always be restored in their pre-update consistent states.

This approach resembles the differential approach. However, the three disadvantages of differential files (Lor77), mentioned above, are avoided. Instead the major disadvantages of careful replacement are:

1. The file or data structure must be such that the technique is feasible. For example, if a file was implemented as a list of linked pages, then this approach would be impossible (or incur prohibitive costs), because if a page is replaced then the page pointing to it is to be updated and may therefore have to be replaced, which will require the updating of the page pointing to that page, and so on. The constraints and requirements careful replacement sets for the data structures are fully described elsewhere (Ver76) (Ver77) (and in later chapters of this thesis).

2. There is a certain overhead in (disk) accesses. In GEORGE 3, by using the technique only for files, but not for the much more heavily used MASTER-directory, the measured overhead reported by Newell is surprisingly insignificant. The method is also used in the MU5 system (Gam73), so also the overhead in that system must at least be acceptable.

Files in the system described by Lampson and Sturgis (LaS76) are updated using careful replacement, during the processing of the intention lists. Segments in System R could be said to be updated using careful replacement as well; certainly the basic idea is used.

Basically the same technique is used in the CMIC system (GiS76). The storage structure used is similar to those of B-trees (Knu73). If an insertion is made in a full track then two new tracks are obtained and the contents of the full track plus the new entry are put in these two new tracks (see also Fig.2.6). The same is done for the index (the table containing pointers to the data tracks). This method of updating is combined with the use of the leaf-first rule. This rule states that copying of information to slower memory (e.g. from main storage to drum, or from drum to disk) is done such that no descriptor or pointer can ever reference a block at a faster level of the device hierarchy. In this way the following two things are ensured:

1. The (sub-)structure on mass storage is always valid, because this (sub-)structure will always be a valid and consistent tree.
2. No data on mass storage is ever removed from the structure during update. Instead replacement is used by using new tracks when necessary.

The careful replacement technique is often used in filing systems using a hierarchy of devices by employing the leaf-first rule and the root-segment rule (see also Fig.2.7). The root-segment rule states that if a data page is on a particular level of storage in the hierarchy, then every directory page between it and the root of the file is on that level or a faster level (the root is the top directory in the tree of directory pages). These two rules mean that careful replacement is used at every level in the hierarchy. So if, for example, the contents of core is lost after a crash then the drum and disk will still have two copies of the file: the disk copy contains an old value, the drum copy (of which some pages are on drum and the others on disk) contains a newer value of the file. Fig.2.7 shows how updates of a file, made in core, are subsequently made to the copies of that file held on drum and disk using these two rules, thus always maintaining a valid and consistent

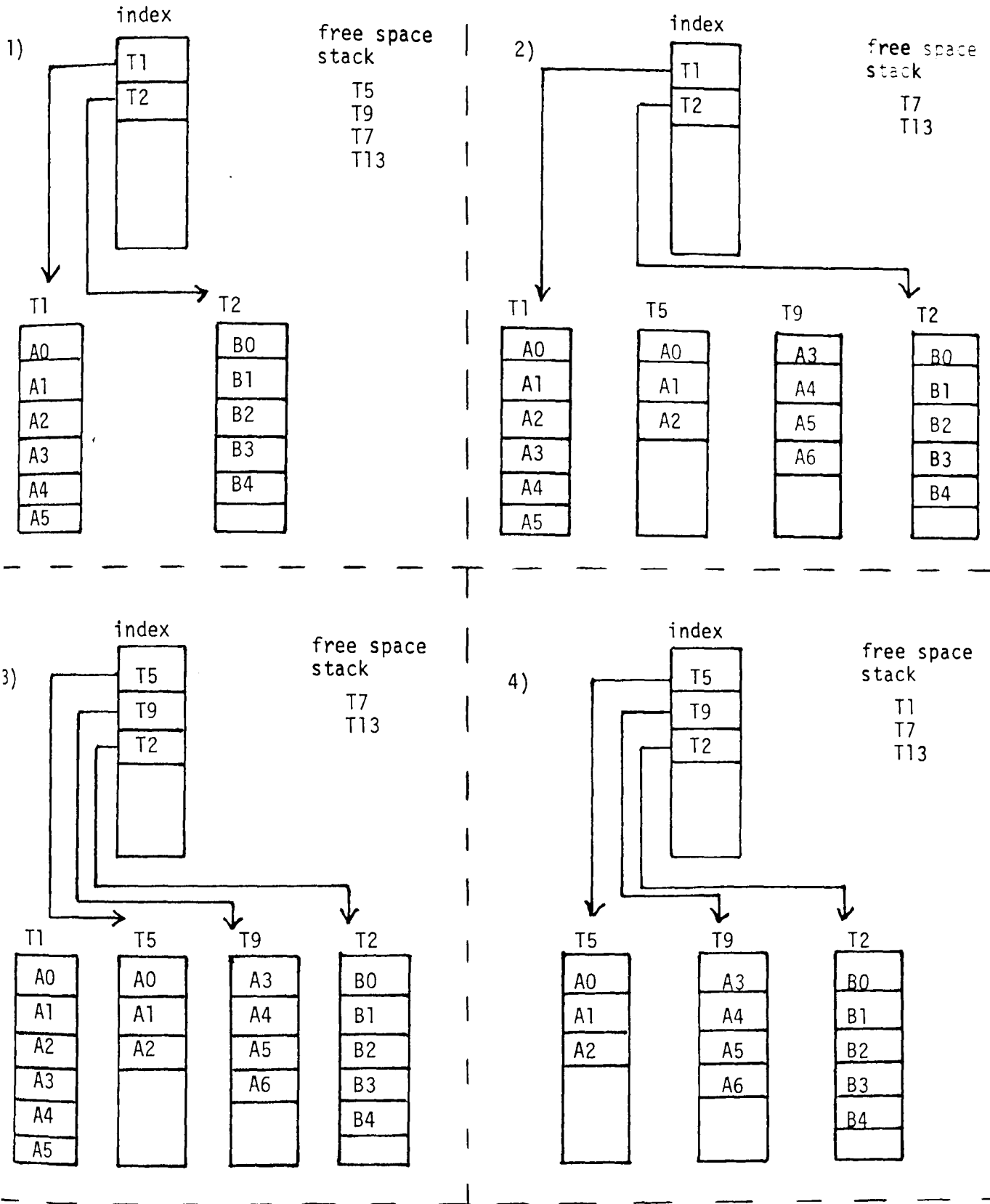
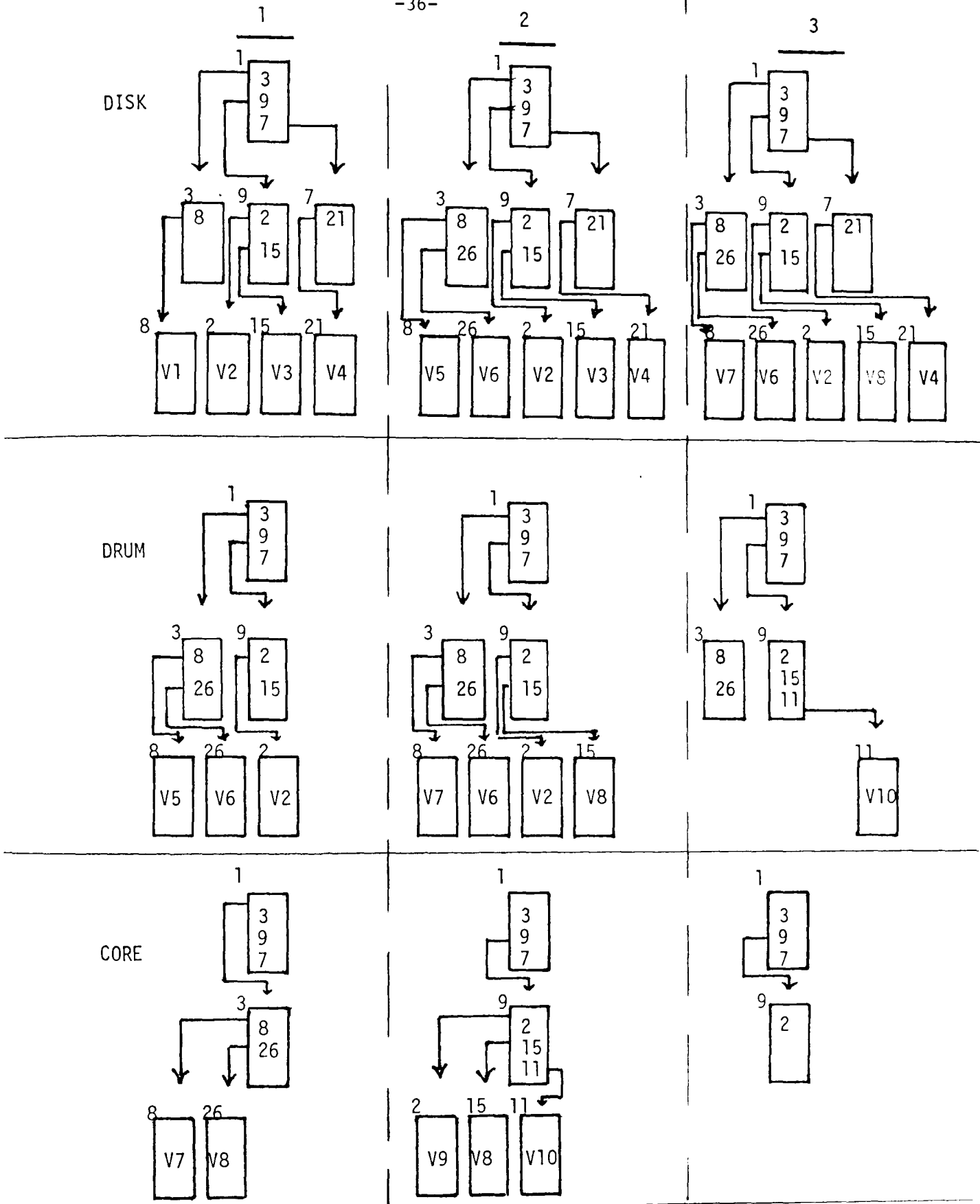


Fig.2.6, The insertion of an entry A5 in a full track in a storage structure as used in CMIC, in four steps.

copy at every level. Although pages are updated "in place" this technique can be classified as being a careful replacement technique, because of the way in which the total structures are updated (the replacement takes in fact place on other devices and updates are consolidated on the device(s) maintaining the files) and because every valid tree structure, in systems using these rules, is by definition a consistent structure. If the contents of both core and drum are lost then there will still be a consistent copy on disk. If the contents of core are lost then the updates which were reflected in the core copy (which is partially in core, partially on the drum, and partially on disk) and not in the drum copy will be lost. If the contents of the drum are lost as well then the updates which were reflected in the drum copy, but not yet in the disk copy, will be lost too. A system using these rules has been described by Schwartz (Sch73). Files are trees of pages which are either index pages (i.e. directory pages) or data pages. A file descriptor is the root of the index table in the system described by Schwartz and a directory file contains the descriptors. In that system, and in CMIC too, the directory file is updated "in place". If absolute crash resistance was to be provided in this system then the multiple copies technique could be used for the directory, as is done in GEORGE 3 (New72).

The careful replacement technique provides an easy way of restoring the state of a data structure as it was before an update or transaction started. The technique can also be used to provide crash resistance.

If the recovery is to be provided for transactions consisting of more than one update then replaced pages (or newly allocated ones in case of insertions or extensions) can be updated "in place". This technique has been used at Newcastle (Ver77) (and is fully described in subsequent chapters of this thesis) to provide recovery for files within user defined scopes, which can be nested. The nesting means that a copy of a file is maintained for every level of nesting in which the file has been operated upon (these copies overlap in identical pages). The current value of the file is in the latest copy. Recovery means restoring the copy as it was before the current recovery scope was entered. Exiting a recovery scope successfully means the replacing of the copy, as it was before the recovery scope was entered, with the latest copy.



Three possible subsequent situations.
where: V_i = a value of a data page.

Fig.2.7, The careful replacement technique in a hierarchy of devices.

2.10 Summary and conclusions

Many of the techniques to provide backing out, crash recovery and crash resistance and to maintain consistency, which are known and used at present, have been described. One of the main conclusions is that the techniques can be used in different environments or for different purposes or complement each other. None of the techniques is out of date, although some are older than others. All the techniques described are still used for different purposes and in different environments.

In order to show which combinations of techniques are used in practice (see also the introduction and Fig.2.1), a cross-reference table, showing which techniques discussed in this survey are used in particular systems, is given in Fig.2.8.

	System R	IMS	George 3	HIVE	MUL TICS	Cambridge, Fra69	EMAS	CMIC	VADIS	Newcastle Ver77
salvation program		*		*	*	*	*	*		*
incremental dumping	*	*	*	*	*	*	*	*		
audit trail	*	*						*		
differential files									*	
backup current	*	*	*	*	*	*	*	*		*
multiple copies	*		*	*						
careful replacement			*					*		*

Fig.2.8, A cross-reference table of systems and recovery techniques.

This table may not be complete for the systems it covers (e.g. System R may have some sort of salvation program), but it shows the most important features of the systems as far as recovery and crash resistance are concerned.

It appears that for filing systems, where short term losses are not considered serious, the combination of incremental dumping, having a complete backup version of the system and a salvation program suffices. This is, for example, used in MULTICS, the Cambridge system and EMAS. A salvation program may be needed for "clean up" purposes after a crash, which may lead to the loss of some data.

This combination can be improved, by using an audit trail, to guard against the loss of any updates, as is possible in IMS. The recovery facilities in IMS, however, seem very ad hoc; there is no general approach and there is no dominant technique as in the Cambridge system or VADIS. IMS provides an enormous range of facilities, 50% of the code is said to be for recovery purposes (Inf75). However, the application programmer, it seems, needs to build his own mechanisms and utilities, certainly if high integrity is required. The programmer also has to make explicit checkpoints if they are required.

The loss of any completed update can also be avoided by using careful replacement or multiple copies as in GEORGE 3, HIVE, CMIC and System R. It may also avoid the need for a salvation program (e.g. as in GEORGE 3). Also the differential files technique is very powerful and can be used to provide recovery facilities and crash recovery.

Audit trail with backup, or incremental dumping with backup, or audit trail with incremental dumping and backup, or multiple copies could be used for recovery from more serious failures which other recovery techniques cannot cope with depending on how serious short term losses are. Rappaport does not say in his paper what has been used for VADIS.

It is difficult to make a costs/overheads comparison between the various techniques, however some general statements can be made:

- * If failures do not occur often then the differential file technique and the careful replacement technique give an extra overhead, because other recovery techniques, such as incremental dumping or backup current version or a salvation program are, in general, needed anyway for failures with which these two techniques cannot cope. However, the failures these two techniques do cope with, are coped with much better (the data base is crash resistant, thus the correct state is

maintained) and more efficiently (no separate tapes, for example, need to be mounted and processed).

- * The multiple copies technique as, for example, used in HIVE could be regarded as a sledgehammer approach: the overhead is high. However, the technique does provide very high integrity such as was required in HIVE.
- * The overhead with audit trail is high, because every operation on the data base may cause an audit trail entry to be created. The audit trail technique as a recovery technique is therefore, in general, only justified if the audit trail is required for certifying the integrity of the system anyway, or if recovery must almost always (even after for example a head crash) involve the restoration of the correct state.
- * The incremental dumping and backup current version technique are, in general, the best technique for providing recovery from failures that cause extensive damage, such as a head crash on disk, if the recovery technique does not necessarily have to restore the correct state, but if instead the restoration of a consistent state is acceptable. The overhead of these techniques is not very big, because it involves checkpointing of the files or the whole data base only once every N minutes or N updates.
- * The costs of a salvation program completely depend on the number of crashes, because overhead is only incurred when the program is used, not during normal processing.

One of the techniques which is used increasingly for multi-user environments (or multi-machine environments) is the careful replacement technique, either explicitly (New72), (Gam73), (LaS76), (GiS76) (and in fact also in (Lor77)) or by the use of the root-segment rule and leaf-first rule in systems using a hierarchy of devices (Sch73). The combination of careful replacement with multiple copies has recently received much attention (GiS76), (Ast76), (Ver77). Also the differential file technique is written about more than in the past (Rap75), (SeL76).

The attention that has been paid to these techniques during the last few years makes it reasonable to assume that they will be used more widely during the next decade. The techniques ensure that data integrity is unlikely to be corrupted through failures and the extra costs weigh less heavily than they did a number of years ago. Data integrity is becoming a more important issue than efficiency, because machines are becoming much faster and cheaper and, at the same time the complexity and volume of the data maintained is increasing.

The techniques described in this survey are special to provide recovery for data bases (on secondary storage). The present thesis generalizes the basic concepts involved and describes system structures in which recovery is provided for different complex data types (not just files). The problems of incorporating general recovery techniques to provide recovery for different types in different levels of a multi-level system are discussed. An analysis of how the techniques can be used to provide "nested recovery" as provided by recovery blocks and a detailed analysis of what the notion of careful replacement implies are given. It appears, for example, that the choice of the objects to replace is of importance for the feasibility of the technique, and that there are different ways in which this careful replacement technique can be implemented.

3.0 PARTIALLY RECOVERABLE INTERFACES IN MULTI-LEVEL SYSTEMS

3.1 Introduction

This chapter presents an approach to the construction of recoverable multi-level systems and a mechanism to implement this approach. The recovery considered will, as was mentioned in chapter one, be described in terms of the particular system structure which has been developed at Newcastle upon Tyne, England, to facilitate fault tolerance, namely recovery blocks (Ran75). Recovery blocks provide a good framework within which recovery systems in general can be discussed and described.

A particular view of how to build fault tolerant multi-level systems (which is similar to a view described by Randell (Ran75)) is compared with the alternative approach described in this chapter. Some of the basic principles are compared with the principles used in a new approach being developed by Banatre and Shrivastava (BaS77).

First the definitions of the terms used in this chapter are given in section two. Then section three describes the basic principles on which the chapter is based. The most important principle of the approach taken in this chapter is to provide recoverability for types. This approach is compared with an approach whereby recoverability is provided for operations (BaS77). Section four describes a particular scheme for implementing recoverable multi-level systems built with so-called completely recoverable interfaces; a scheme which is based on that view is described by Randell (Ran75). The disadvantages of this scheme are described in section five. Section six gives an alternative approach where partially recoverable interfaces are used and shows that there are good reasons for using this approach. The problems and constraints of implementing a multi-level system using that alternative approach, and a special mechanism to overcome these problems are described in section seven. A multi-level prototype system, which shows that the ideas discussed in this chapter can be used in practice, is described in section eight. Section nine shows how the mechanism can be used to build up a system of recoverable type managers each of which adds new recoverable types, implemented in terms of unrecoverable types, to an existing interface. A full comparison between the "recovery for types" approach and the "recovery for operations" approach is made. Finally, section ten gives some general final conclusions.

3.2 Definition of multi-level system and recoverability

Before the notions "multi-level system" and "recoverability" can be defined, a number of other terms have to be specified.

A level is a set of programs which provides (to a higher level) a more abstract view of the machine on which it is running. A level is very similar to a class in SIMULA (Bir73); a program can be provided with a more abstract view of the machine by prefixing it with a class in SIMULA.

The more abstract view of the machine given by a level is provided by one or more new types implemented by that level. Some of the existing types may be hidden by a level, however the ways in which this can be done are outside the scope of this thesis. A user can write programs to run on this level. (That is the programs are effectively executed by the new, more abstract, machine.)

The new abstract view provided by a level is called an interface. A description of an interface consists of a definition of types (and/or resources), (for example: core words, arrays, files or an operator's console) and the operations that can be performed on objects of these types. A user's program specifies a sequence of operations on objects of these types provided in the interface. Issues concerning the languages in which the user has to specify his sequences of operations are irrelevant for the purpose of this thesis.

In a multi-level system programs (or program parts or procedures) can be grouped in sets L_0 to L_n where the following properties hold:

1. Every group L_i provides one or more new abstract types.
2. Every program in group L_i is the implementation of an operation on a new type provided by L_i .
3. Programs of any group L_i invoke programs of one or more of the groups L_0 to L_i , but not programs from groups L_{i+1} to L_n . In other words group L_i uses the abstractions provided by groups L_0 to L_{i-1} .
4. A group L_i may be an interpreter. L_i is an interpreter if no level L_j , $j > i$, can directly invoke any program in any level L_k , $k < i$. A group L_i could also consist of a set of programs which interpret (see definition below) only some of the instructions of the user program, the other instructions being interpreted by groups L_j , $j < i$.

A program P in a group L_k is said to invoke an

operation R (i.e. a program) in a group L_m , $m < k$; the program R invoked is said to interpret an operation for program P.

An example of such a multi-level system is given by Madnick and Alsop (MaA69), and shown in Fig.3.1.

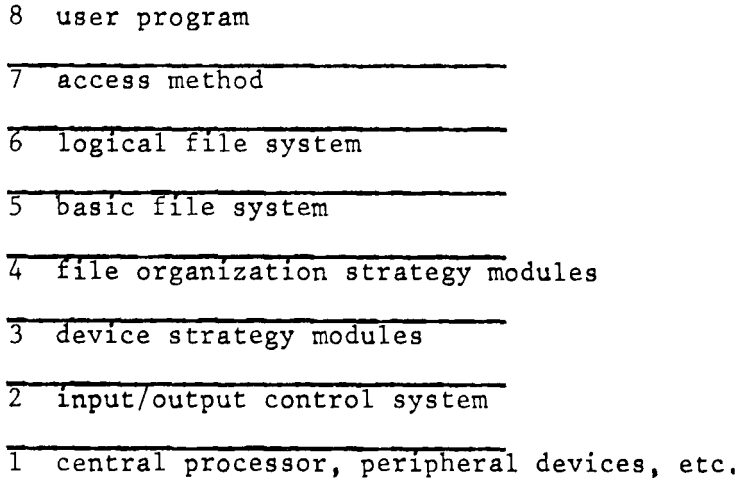


Fig.3.1 A multi-level file system, designed by Madnick and Alsop.

Every group L_i is a level in a multi-level system. Every level provides a new interface. The new types provided in an interface are mapped on (represented by) types provided in the interface of the underlying machine (i.e. the interface provided by the lower levels). The underlying machine of a level i is defined in the interface provided by level $i-1$ (see definition of interface) and is implemented by levels 0 to $i-1$. Thus in Fig.3.1 level 7, the access method, provides the different types of files, such as fixed length record files, sequential variable length record files, and so on. Level 6 maps file names on file identifiers and thus provides type: named file. Level 5 converts the file identifier into a file descriptor which gives the logical structure of the file (e.g. a tree, a list or a network, possibly with a separate header) and thus provides type: file. Level 4 provides type: file structure (it map logical virtual addresses on real addresses), level 3 provides type: physical data structure (it does clustering of file records and maintains a list of free records), and level 2 provides type: record (it performs the I/O to the devices).

A level L_i which is not an interpreter level is termed a procedural level.

The definitions of level and multi-level system are very similar to notions defined elsewhere (Dij68), (Par72b), (HoR73). There are several reasons for building multi-level systems rather than systems consisting of one level. One important reason, for example, is that it is preferable to split up programming problems into subproblems, that is to make one abstraction at a time.

For example, in an operating system levels could be devised to implement a nucleus, a paging mechanism, a scheduler and a filing system (not necessarily in that order). Similarly, in data base systems, multiple levels (or "schema") are used for purposes of providing abstractions (sometimes also called views) pertinent to different data base users. These abstractions can be effectively integrated and consistently maintained. As shown below, the need to enhance the reliability of the whole system by providing recoverability may give an additional strong incentive to implement a system in levels.

This chapter therefore considers the problem of extending the recovery block scheme to multi-level systems. A full description of recovery block structures has been given elsewhere (Ran75), (Hor74). A brief description of the working of a recovery block has been given in chapter one.

An error in a program is either:

- * a breach in the interface rules (such as a division by zero)
- * an unsuccessful acceptance test
- * a user call of ERROR.

Operation ERROR forms part of the recovery block facilities.

An error is not necessarily associated with the program "detecting" the error. One could say that the underlying machine detects a perpetration of a breach in the interface rules while the program itself detects an error if an acceptance test is unsuccessful or if ERROR is called. This chapter, however, will not be concerned with error detection; it only deals with mechanisms that can be used for the implementation of recoverable multi-level system.

When the objects of a recoverable type (i.e. a type for which recoverability is provided) are used in a recovery block and an error occurs in that recovery block then the underlying machine will restore the values of those objects to their state when the recovery block was entered. (The next alternative of the recovery block will then be invoked by the underlying machine.) If a type is not a recoverable

type then it is an unrecoverable type.

Objects of a recoverable type are termed recoverable objects and objects of an unrecoverable type are unrecoverable objects. Unrecoverable objects are not restored when a recovery block alternative is backed out. So operating on unrecoverable objects inside recovery blocks will in general lead to unpredictable effects, unless special precautions are taken.

The term "type" will be used rather loosely in this chapter, because if one object is recoverable and another object is unrecoverable then they are not of the same type. If however, the recoverability/unrecoverability is the only difference between two objects then this thesis will refer to them as being recoverable and unrecoverable objects of the same type.

If all types in each interface are recoverable in the scheme used, then this scheme will be called the Completely Recoverable Interface (CRI) scheme. This in fact is the sort of scheme described by Randell (Ran75). The scheme proposed in this thesis is a scheme whereby interfaces generally contain recoverable and unrecoverable types and will therefore be called the Partially Recoverable Interface (PRI) scheme.

A fault tolerant level is a level which provides new recoverable abstract types, thus extending the recovery block facilities. Both the type mechanisms and recovery mechanisms are provided explicitly. Ways in which this can be done are discussed in this chapter.

A fault tolerant interpreter (as defined by Randell (Ran75)) is an interpreter which provides recovery block facilities. According to the definition, all of the types in the interface provided by this interpreter are recoverable, although recoverability is not explicitly associated with types, but with "effects" (which can be undone). This strict requirement will not be imposed on the definition of "fault tolerant interpreter" in this thesis. A fault tolerant interpreter may provide an interface with both recoverable and unrecoverable types.

If the interface provided by a level i contains recoverable types (and recovery block facilities are provided to the user) then the underlying machine of level $i+1$ is said to be a fault tolerant machine. This machine is in fact the full set of levels up to level i . A recoverable multi-level system is a multi-level system of whose levels one or more are fault tolerant levels.

A fault tolerant level L implements user operations on objects of the provided recoverable and unrecoverable

abstract types. If a user operates on such objects inside a recovery block then level L must be able to undo these operations in case an error occurs in the user program. In order to be able to do this, level L will have to store information about the changing states of the objects such that undoing can be performed when necessary. A cache is a data structure local to the level (invisible to the user running on it) which is used by the level to store such information. The cache is said to be associated with the user. If the cache is used to undo operations on objects of a particular type (which may be a shared resource) then it could be said to be associated with the type. Although recovery in this thesis is associated with the types provided to the user, caches will be said to be associated with users, to indicate that the undoing is done on behalf of users; the caches are used to restore user objects in order to back out the users.

Several variant implementations of a particular fault tolerant interpreter have been described elsewhere (AnK76), (Hor74), (Ran75). In those implementations recoverability is provided for program variables. When a variable which is not a local variable in the current recovery block is updated inside an alternative for the first time, then the interpreter will record its old value in the cache.

The phrase to cache an operation will be used in this thesis to denote the maintaining of sufficient information in the cache, by the fault tolerant level, in order to be able to undo the effects of the combination of that operation and all of the previous operations on the same object in the current recovery block, should this be necessary. Processing the cache of a level is the act of restoring of the user's objects of the recoverable types provided by that level, to the (abstract) states they were in before the current recovery block was entered. This chapter will not distinguish between the various different schemes that could be used to achieve this restoring. Examples have been described elsewhere (AnK76), (Hor74), (Von76). Processing the cache in those examples means simply using the cache to restore the old values of objects that have been updated in the alternative being backed out, and subsequently removing the relevant entries (made in the current recovery block) from the cache.

The cache in a level could also consist of, for example, a list of entries as maintained in System R (Ast76), a data base system, for the backing out of transformations. For every operation performed during a transaction an entry is made, which consists of sufficient information to undo the operation performed. Processing the cache then consists of processing this ordered list of entries in reverse order, to undo the operations performed during the transactions. The cache could also be a means of

holding conventional checkpoints (i.e. copies of the entire state), or more exactly, of all objects which could be modified.

3.3 Basic principles

This chapter is based on the following principles:

1. Recovery is provided for types.

Whenever an object of a particular recoverable type is to be restored then:

- a. the abstract state of the object at the time the current recovery block was entered is known,
- b. the present state is known (which may be useful for efficient state restoration), and
- c. this (and only this) knowledge is used to restore the state of the object.

This is very different from the principles discussed and used in a scheme described by Banatre and Shrivastava (BaS77) where recovery is provided for operations rather than for types. Every operation performed may cause a corresponding reverse operation to be performed when backing out is done. (This could be said to be a "reversed-audit-trail" scheme.)

2. In general interfaces will contain recoverable and unrecoverable types.

A level implements new abstract types in terms of recoverable or unrecoverable types or a combination of both. This is different from the original view of recovery blocks and multi-level systems (Ran75), where in each interface every operation and type was recoverable. The fact that input/output operations, for example, in the implemented fault tolerant interpreter originally were unrecoverable, was regarded as a deficiency or incompleteness of the system.

3. If a program operates on objects of unrecoverable types inside a recovery block then it will be the responsibility of that program to make sure that no inconsistencies or other problems arise if an alternative of the recovery block is backed out and the program continues with the next alternative.

4. If a level L consists of a set of procedures providing new abstract types which are mapped onto recoverable types, then there is no need for this level to explicitly provide recoverability (i.e. implement a cacheing mechanism) for these new types. If the level L+1 program (which invokes procedures of level L) generates an error, then the objects used for the representations of objects of types provided by L will be automatically restored (by the underlying machine of L). (See also Fig.3.6, where level i maps objects of types T1, ... ,Tn, on recoverable objects of types t1, ... ,tm; an error in level i+1 causes the underlying machine of level i to restore objects of types t1, ... ,tm, thus restoring objects of types T1, ... ,Tn.) Procedures implementing operations on objects of types provided by level L may be invoked from inside a recovery block in level L+1. Levels are levels of abstraction so the underlying machine of level L will also restore level L+1. So the values of the objects of the abstract types used in level L+1 will be restored if the underlying machine of level L restores level L. In this case recovery is provided implicitly for the new abstract types provided by L.

There are, however, two good reasons for building multi-level systems consisting of levels that explicitly provide recoverability (i.e. by performing cacheing) for the new abstract types they implement. These reasons are:

- a. Flexibility and efficiency in restoring the state of the machine as seen by the user.
A level providing a new type can take advantage of the fact that more than one concrete representation may exist for a particular abstract state of the machine as seen by the user. For example, in a level concerned with the management of buffers it may not always be necessary to restore exactly the contents of all of the buffers in order to undo buffer management operations.
- b. Recovery for concurrent processes.
In some cases when concurrent processes share data or acquire and release resources, it may be impossible to continue usefully by restoring the concrete representations of the abstract types (except at prohibitive penalty). However, this is outside the scope of this thesis and is the topic of ongoing research discussed elsewhere (BaS77), (MeS77), (RLT77).

3.4 Completely recoverable interfaces

The multi-level systems considered in this section consist of levels that explicitly provide recoverability for all types in every interface (such levels are CRI-levels). These systems are in fact the sort of systems described by Randell (Ran75); every interface contains recoverable types only, and when an alternative of a recovery block in a program is backed out then the state of the machine as seen by that program will be restored completely to what it was just before the alternative was entered.

The recoverable multi-level systems described in this section consist of levels which are fault tolerant interpreters (all of which provide recoverable types only). It will be shown that using interpreter levels is the only way in which the CRI-scheme can be implemented if different levels provide recovery for the new types explicitly. The reasons for describing the CRI-scheme are to give the reader an introduction and a better insight into the general concepts of recoverable multi-level systems. Everything in the CRI-scheme is easy to understand and very straightforward; a description of the CRI-scheme serves as a good introduction to the (more realistic and more sophisticated) PRI-scheme.

3.4.1 Level: programs and data objects

A CRI-level provides recovery block facilities and performs cacheing, as necessary, for all of the types it supports. These new recoverable types are mapped by the level onto recoverable types of the underlying machine.

Since all types in all interfaces are recoverable, the local program data of a level L, including data structures used for the cache maintained by this level will be recoverable. The level itself is a set of programs written to run on another fault tolerant level (except perhaps a "bottom level", which can for that reason not use recovery blocks and be backed out if an error occurs). Consequently programs in this level L, which run on an underlying fault tolerant machine, can also use recovery blocks. If a program in level L (see Fig.3.2) generates an error and is backed out then all of the operations on objects used by that program will be undone by the recovery mechanism (which is termed "UNDO") in the lower level. The lower level UNDO undoes all of the operations performed on the cache (in level L) associated with the user, locally used data, and all operations on data used for the representation of user data. The program being backed out generally implements

(part of) a transformation in the next level up (the user level). When a program in level L is backed out, the next alternative of its recovery block will be invoked. This next alternative will then try to implement something on behalf of the user, in an alternative way.

This can be illustrated by the following example (see Fig.3.3):

- * Suppose that a user uses a complex number C1 which has a value 1,3 . The user then uses complex operation ADD to add the constant 5,4 to C1 inside a recovery block.
- * The level below the users level, the complex level, represents C1 by reals R1 and R2. Before the user updates C1, R1 has value 1 and R2 has value 3. The complex level interpretes the user's add-instruction by updating R1 and R2, which will be given values 6 and 7 respectively. Since the user makes the update inside a recovery block, the complex level will cache this by storing tuples R1,1 and R2,3 in its cache. The complex level uses reals R100, R101, R102 and so on, as cache. The interpretation of the user's add-instruction is done by a program called ADD COMPLEX in the complex level, which itself uses two different recovery blocks in which it updates R1 and R2 (see Fig.3.3).
- * The level below the complex level, the word level, represents reals R1 and R2 by words W1 and W2 and reals R100, R101, R102 and so on, by words W10, W11, W12 and so on. Assignments to reals performed in the complex level are interpreted in the word level, by program ASSIGN_REAL. Since the complex level updates the two reals of the complex to be updated, inside recovery blocks in ADD COMPLEX, the word level will cache the changes made to reals R1, R2, R100, R101, R102 and so on. The word level uses words W100, W101, W102 and so on, as cache.

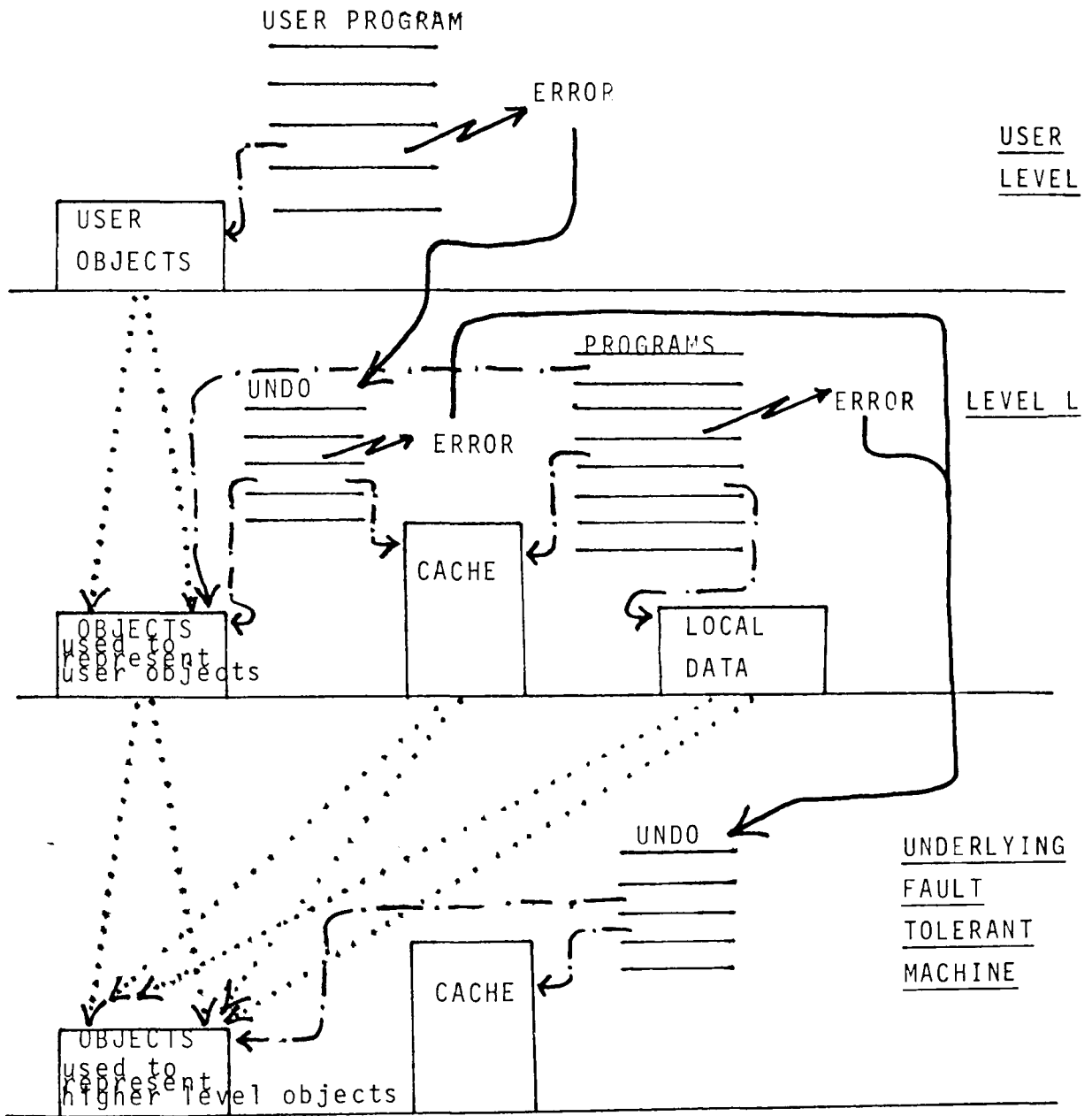


Fig.3.2, A CRI multi-level system.

Where:

- > denotes: program invokation
-> denotes: type mapping (representation)
- - - - -> denotes: program uses data pointed at

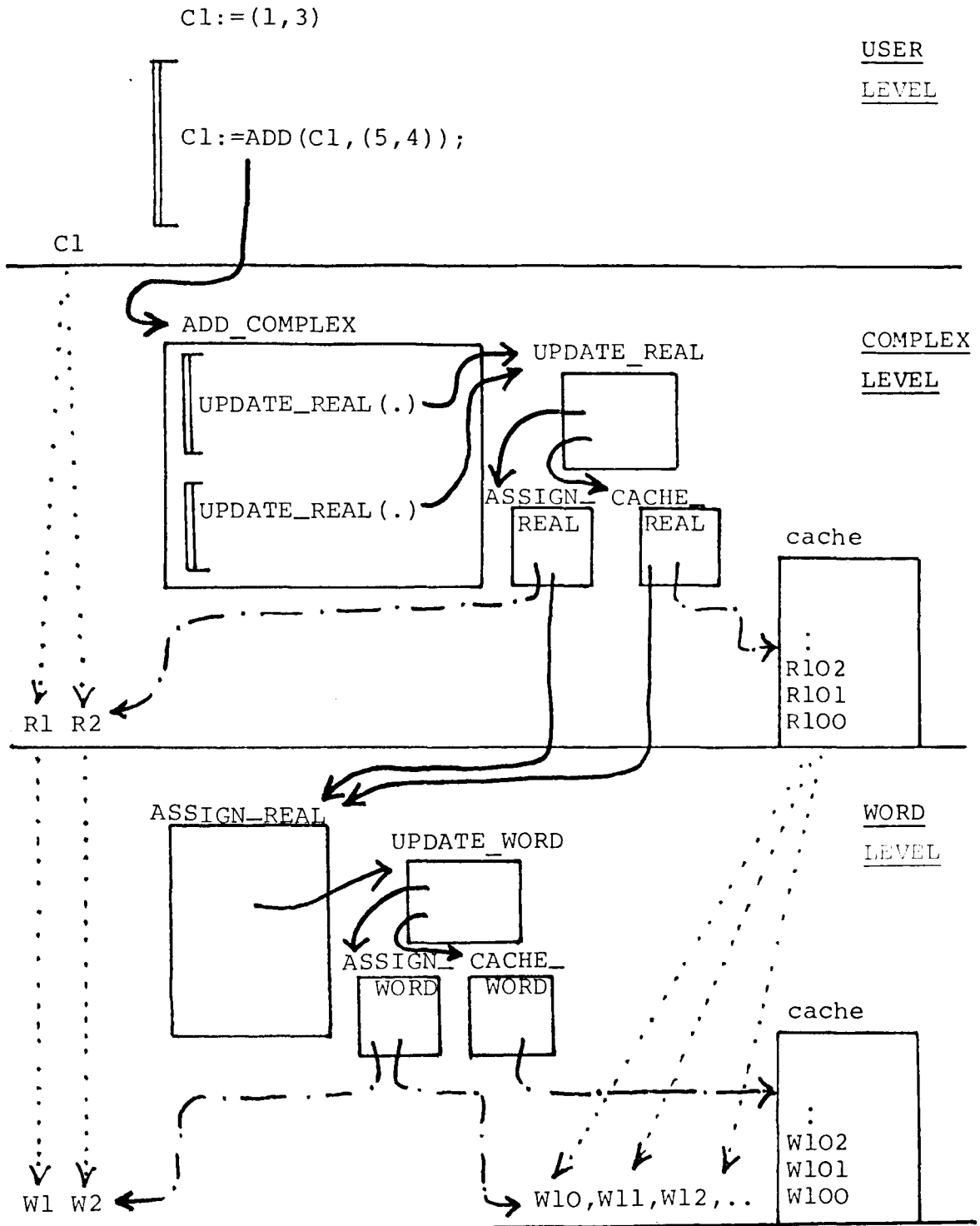


Fig.3.3, An example of a CRI-multi-level system, providing "complex" data.

Where:

- > denotes: program invokation
-> denotes: type mapping (representation)
- > denotes: program uses data pointed at

The following sequences of states are now possible if no errors occur:

	1 before ADD	2 after first ASSIGN REAL	3 after first CACHE REAL	4 out first rec. block	5 after 2nd. ASSIGN REAL	6 after 2nd. CACHE REAL	7 after ADD COMPLEX	8 after user rec. block
C1	1,3	und	und	und	und	und	6,7	6,7
R1	1	6	6	6	6	6	6	6
R2	3	3	3	3	7	7	7	7
cache	empty	empty	R1,1	R1,1	R1,1	R2,3 R1,1	R2,3 R1,1	empty
W1	1	6	6	6	6	6	6	6
W2	3	3	3	3	7	7	7	7
W10	0	0	@R1	@R1	@R1	@R1	@R1	0
W11	0	0	1	1	1	1	1	0
W12	0	0	0	0	0	@R2	@R2	0
W13	0	0	0	0	0	3	3	0
cache	empty	W1,1	W11,0 W10,0 W1,1	empty	W2,3	W11,0 W10,0 W2,3	empty	empty

where : @Ri = address of Ri
und = undefined.

Fig.3.4, A sequence of states that could occur in the system of Fig.3.3.

- * If an error occurs in the user program after the ADD instruction, but before the user recovery block is left (situation 7), then the complex level uses its cache to restore situation 1.
- * If an error occurs in the complex level in ADD COMPLEX inside the second recovery block, just after the return from CACHE REAL (situation 6) then the following sequence of states will be gone through if the second alternative of that recovery block performs the same operations and no subsequent errors occur:

	6 after 2nd. CACHE REAL	9 after error in ADD COMPLEX	10 after 2nd. ASSIGN REAL	11 after 2nd. CACHE REAL	12 after ADD COMPLEX	13 after user rec. block
C1	und	und	und	und	6,7	6,7
R1	6	6	6	6	6	6
R2	7	3	7	7	7	7
cache	R2,3 R1,1	R1,1	R1,1	R2,3 R1,1	R2,3 R1,1	empty
W1	6	6	6	6	6	6
W2	7	3	7	7	7	7
W10	@R1	@R1	@R1	@R1	@R1	0
W11	1	1	1	1	1	0
W12	@R2	0	0	@R2	@R2	0
W12	3	0	0	3	3	0
cache	W11,0 W10,0 W2,3	empty	W2,3	W11,0 W10,0 W2,3	empty	empty

Fig.3.5, A sequence of states that could occur in the system of Fig.3.3.

- * If an error occurs in the user level in situations 8 or 13 then the user program will be aborted.

Summarising, the following characteristics of a level in a CRI multi-level system are important:

- * A fault tolerant level distinguishes its data between:
 - Data used to represent the objects of the new abstract types provided.
 - A cache associated with the user.
 - Local data used in and by this level such as: work data, housekeeping data.

- * The underlying machine of a level L (see also Fig.3.2) will restore objects used inside a recovery block if an error occurs in that level. In other words an error in level L invokes the lower level UNDO which restores all data operated upon in level L (i.e. cache, local data and other objects of types t_1, \dots, t_n used to represent user objects).

- * If an error occurs inside a recovery block in a user's program interpreted by a level L, then this level will use the cache which it associates with that user to undo the operations on objects performed by the user inside that recovery block. Level L maps these user objects on lower level objects. Consequently it will change those lower level objects such that the user's view of these data is restored to what it was before he entered his current recovery block. Since it is the user's program that is to be backed out, this does not necessarily mean that the objects used for the representation of objects of the newly provided type have to be restored by this level exactly as they were when the user program entered its recovery block. In other words there may be many representations of the same state of the machine as seen by the user. A level may take advantage of this when backing out a user program. Backing out of a user program is normal progress of level L.

An example of operations on a level, implementing a recoverable buffer management system, illustrates this: Suppose that a user has available a buffer A. He then enters a recovery block and subsequently:

- claims a buffer B
- claims a buffer C
- updates buffer A
- updates buffer B
- updates buffer C
- releases buffer C

If subsequently an error occurs, these operations have to be undone. The buffer management level can undo these operations by performing the following operations:

- free buffer B
- undo the updating of buffer A.

If recovery would have been provided at a lower level of abstraction, say at the word level, then the buffers A, B and C would have been restored completely, thus requiring more recovery data (cached data) and more processing for recovery. The more abstract buffer management level can take advantage of the knowledge it has about the use of these buffers and the different states of the core areas which represent the same state of the buffers as seen by the user. So, for example, the contents of a released buffer do not matter, so buffer C can remain unaltered and B only has to be released.

If an error occurs inside a level and there is no containing recovery block then that will lead to the abandonment of this level and with it any higher levels. It is up to the next level down, which deals with the error, to decide what to do next.

If a program in a level exhausts all of its alternatives then that means that it fails to interpret (a part of) the user's program successfully. Consequently a sensible last alternative of the programs in a level, would be to back out the user of that level and see if the next alternative of the user's program will be more successful. Obviously this will only have a chance of success if that user's program happens to be inside a recovery block (and not in the last alternative of that recovery block). A similar technique has been employed in a fault tolerant interpreter which was implemented at Newcastle (AnK76).

3.4.2 Interpreters for CRI-levels

In order to show why a level providing explicit recovery for new types, under the CRI-scheme, has to be an interpreter rather than a set of procedures, consider the following situation:

Suppose that a level i is a set of procedures that do not implement all of the types in the new interface (see Fig.3.6). An error in level $i+1$ will require the processing of the cache in level i , for the restoring of values of objects of the newly provided types. This error will also require backing out by the underlying machine of level i , for the restoring of the values of objects of all the other types used in level $i+1$. In other words an error in level $i+1$ invokes the UNDO of level i and the UNDO in the underlying machine. The UNDO in level i restores values of types T_1, \dots, T_n , while the UNDO of the underlying machine restores objects of types t_1, \dots, t_n . The UNDO in level i only operates on objects of types t_1, \dots, t_m , which are used to represent objects of types T_1, \dots, T_n . The UNDO of the underlying machine will operate on objects used to represent objects of types t_1, \dots, t_m used in levels i and $i+1$, in order to restore all objects of types t_1, \dots, t_m used. This, however, means that the whole cacheing in level i has no effect with respect to the restoration of the states of objects in level $i+1$, because the underlying machine of level i will restore the level $i+1$ objects of the newly provided types anyway (by restoring the objects onto which they are mapped). Thus there is no need to do any cacheing in a procedural level in a CRI-system. Such a level cannot even provide explicit recovery and therefore level i would have to be an interpreter-level in order to make the

explicit restoring of level $i+1$ objects of the new types, by level i , possible. Explicit recovery is only performed by the interpreter levels in a CRI-system.

If level i consists of a set of procedures then the processing of the cache in level i would be useful, if level i provided recoverable types by using unrecoverable types of the underlying machine. The rest of this chapter will discuss this feature in greater detail.

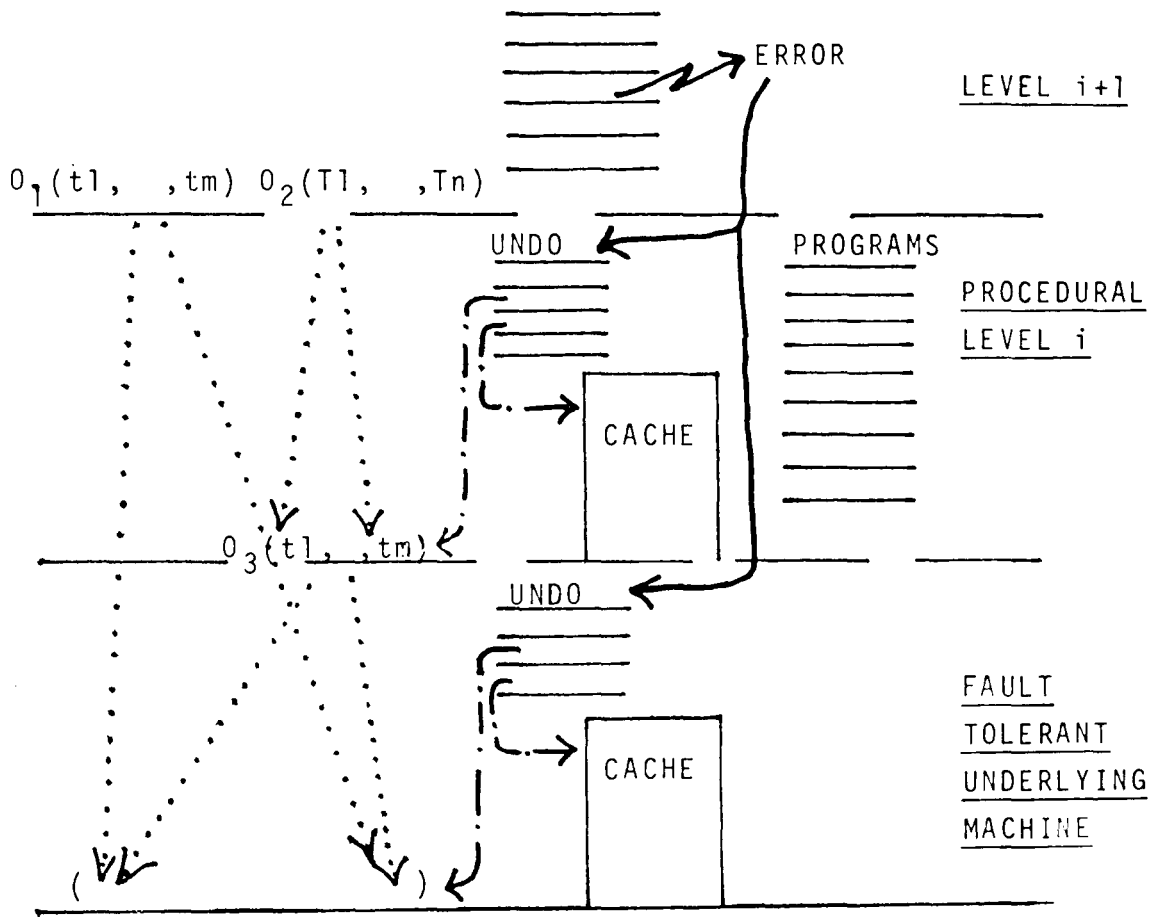


Fig.3.6, A procedural level in a CRI system.

Where:

- > denotes: program invokation
-> denotes: type mapping (representation)
- - - - -> denotes: program uses data pointed at

3.5 The disadvantages of completely recoverable interfaces

There are major disadvantages in building recoverable multi-level systems in the manner just described. In order to illustrate these disadvantages an example will be used.

Consider a level in a multi-level system, implementing a filing system and providing recoverable files which it maps onto recoverable disk pages in the underlying machine. The filing system will then use the available set of recoverable disk pages both for storing files and for cacheing purposes. Suppose that the lower level, the device access level, provides a set of recoverable disk pages which it maps onto recoverable disk blocks. This lower level then needs separate disk blocks for representing the provided recoverable disk pages and for cacheing purposes.

Now consider the following situation (see also Fig.3.7): a user has a file called FILE1, which consists of three data pages P1, P2 and P3, in which the user data are stored. The user calls , from inside a recovery block, a standard procedure CONVERT to alter all occurrences of a character A into a character B. The filing system then changes pages P1, P2 and P3 and caches these changes, which implies, in this case, that pages P4, P5 and P6 are used to keep the previous values of P1, P2 and P3. Pages P4, P5 and P6 are thus part of the cache maintained by the filing system. The device access level, which maps pages on real disk blocks, updates blocks BL1 to BL6 when the filing system updates pages P1 to P6. If the filing system updates P1 to P6 inside a recovery block then the device access level will cache the updates made to BL1 to BL6 by placing the previous values of BL1 to BL6 into, say, BL7 to BL12, which are part of the cache. So eventually 9 extra disk blocks are needed to update a file consisting of 3 disk blocks.

Thus, in general, this scheme results in a fairly substantial loss of hardware resources. A big overhead (in time) due to cacheing will also occur. If a user operation would normally cost three disk writes (in a system that does not support recoverability) then a recoverable filing system may interpret this particular disk write as six disk writes and the device access level may interpret this as twelve disk writes. Thus in order to let the user write three file pages inside a recovery block, the whole system may have to do many extra disk writes.

Thus the major disadvantages of the scheme are:

1. A loss in hardware resources: the user of the system can only use a small part of the resources.
2. A loss in efficiency: the system may spend most of its time performing cacheing rather than actually performing the operations on behalf of the user that would happen in a non-recoverable system.

So even if a multi-level system consisting of interpreters is wanted, as for example in an APL system as described by Randell (Ran75), then the extra overhead in the total system incurred by implementing fault tolerant interpreters in every level and using recovery blocks in every level, may be enormous. However, in general one would not always want to build a multi-level system using an interpreter for each level (as is required for the implementation of the CRI-scheme); so the approach is unrealistic anyway. It is clear that an alternative approach is required, if levels in a multi-level system are to provide recoverable types explicitly.

The major advantage of the CRI-scheme is that there are no problems in providing high reliability, because recovery blocks can be used in every level (except perhaps in a "bottom level"). It may, however be possible that reliability is adversely affected by the fact that much extra work is to be done.

3.5.1 An alternative CRI-scheme: "bottom level" recovery only

Because of the disadvantages of the scheme described, it may seem sensible to consider the following scheme to implement a recoverable multi-level system, consisting of interpreters, more efficiently:

Only the "bottom level" (i.e. the first software level) is a fault tolerant level providing recoverability as described in the previous section (by implementing a cache mechanism and providing a recovery block structure). Recoverability provided by all of the higher levels is based on "bottom level" recovery ("bottom level" objects are restored to restore all objects of types in higher levels.). Since the levels are interpreters, the implementation of this scheme is not a trivial exercise. The interpreter must interpret the entering of a recovery block in the next level up (i.e. the program being interpreted) by entering a recovery block itself. Similarly an error in the program being interpreted will have to cause the interpreter to

generate an error itself, thus causing the "concrete machine" to be restored and with it the program being interpreted. There is also the complication of ensuring that the user's next alternative is entered.

Although the disadvantages mentioned above are not present under this scheme, the two major advantages of providing recovery explicitly (i.e. recovery can be provided more efficiently and recovery may be possible for parallel processes by providing the recovery at a higher, more abstract, level) in different levels are not present under the scheme either.

For this reason the alternative scheme to provide recoverable multi-level systems, as described here, is not satisfactory.

3.6 Partially recoverable interfaces

A level in a partially recoverable interface (PRI) multi-level system provides a new interface with, in general, both recoverable and unrecoverable types. Any of these types can be mapped onto recoverable types, unrecoverable types or even a combination of both (although no useful application of such a mapping has yet been found).

An example is given below to show that the provision of recoverable types by mapping them on unrecoverable types can be very efficient, in time and space.

Suppose that a new level implements a filing system providing recoverable files (see Fig.3.8). These files are mapped onto unrecoverable disk pages of the underlying machine. The two disadvantages of the CRI-scheme described previously are avoided for the following obvious reason. Lower levels only provide unrecoverable disk pages so no cacheing will be done in lower levels if disk page operations are done for the implementation of file operations. Only the filing system performs cacheing for files or disk pages. Thus the loss of hardware resources and overhead in time, due to cacheing in subsequent levels, is restricted to a minimum.

If a user program generates an error (see Fig.3.8) then the UNDO of the filing system (the procedural level) will be invoked to restore files and the UNDO of the underlying fault tolerant machine will be invoked to restore all of the other recoverable user objects.

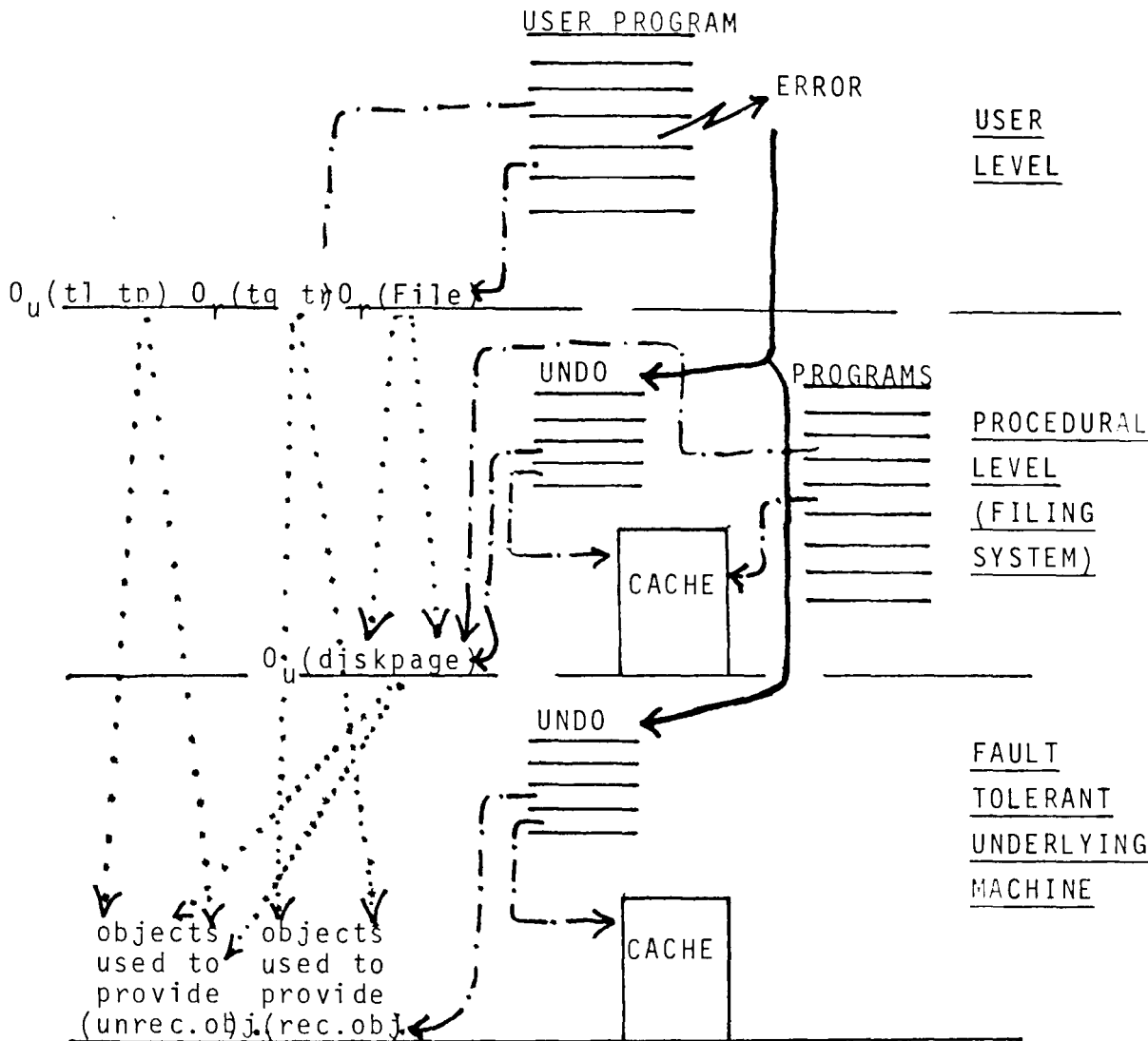


Fig.3.8, A procedural level (filing system) in a PRI system.

Where: $O_u(t1, ,tp)$ = A set of unrecoverable objects of types $t1, ,tp$.

$O_r(tq, ,tr)$ = A set of recoverable objects of types $tp, ,tr$.

- > denotes: program invocation
-> denotes: type mapping (representation)
- - - -> denotes: program uses data pointed at

Efficient state restoration, as is possible under the CRI-scheme, is also possible under the PRI-scheme. For example, the filing system above can provide recoverability for files efficiently. The filing system can take advantage of the knowledge that many representations exist for one state of the filing system as seen by the user (see also the filing system described in chapter four). The filing system may therefore have a more efficient way of restoring the user's view of the files than just restoring all of the

modified disk blocks. For example, the values of free pages do not have to be restored (as with the buffer management example).

As described previously, there is in general no point in implementing a new recoverable type (and implementing a cache to undo operations performed on objects of this type) if this type is mapped onto recoverable types by a set of procedures. Mapping a new recoverable type onto other recoverable types will only make sense if the level providing this new type actually implements all of the types in the new interface and implements a new recovery block structure, so the underlying machine will not have to do any backing out if the user generates an error. Having to map recoverable types on unrecoverable types in a procedural level is, in the view of the author, not such a severe restriction in general, because in many systems no further abstraction of types file, buffer or lineprinter, for example, are made on top of levels implementing the filing system, buffer management or lineprinter package respectively. If further abstractions are made then they can generally be made by a set of procedures without providing recovery for the new abstract type explicitly; the recovery of the new abstract type can be based on the recovery provided for the types used in its representation. If further abstraction is to be made such that recoverability for the new type is to be provided explicitly, then there are basically two solutions:

1. Use an interpreter.
This could, for example, be done in a recoverable data base query system, implemented using recoverable files.
2. Use unrecoverable objects to construct the new recoverable type.
As will be seen, it is generally necessary to be able to provide unrecoverable objects of the same types as the recoverable types provided. In that case the data base query system could be a set of procedures mapping a recoverable data base onto unrecoverable files.

Finally it is worth mentioning another good reason for providing some unrecoverable types (i.e. another good reason for PRI-interfaces) or types of which both recoverable and unrecoverable objects can be allocated, namely for debugging purposes. Suppose, for example, that the programmer of a system notices that the first alternative of a recovery block in a system fails occasionally (this can be seen from the error log that is maintained). If the programmer wants to write values of variables and messages to a special file in order to track down the error, the file must obviously be non-recoverable. In general, unrecoverable objects will be needed to monitor the software.

This section has shown that there is a case for using the PRI-scheme rather than the CRI-scheme. The next section will describe the problems and constraints of this scheme.

3.7 The problems and constraints of the PRI-scheme

If a given level is a set of procedures providing new types and mapping these types onto unrecoverable types then these procedures in this level will operate on objects of these unrecoverable types. If a user (in the next level up) generates an error then the given level will restore all the objects of the recoverable types it provides while the underlying machine will restore all the other user objects of recoverable types. However, if programs in the given level use recovery blocks then their operations on objects of unrecoverable types will not be undone when a program in the given level is backed out.

For example, a user program running on a level that implements a recoverable filing system by mapping files onto unrecoverable disk pages, may use filing system operations inside recovery blocks. If an error is subsequently generated (see also Fig.3.8) then the filing system will restore files and none of the levels below the filing system level will restore anything used for the representation of files (i.e. disk pages).

If, however, an error is generated by a program in the filing system then the operations on (unrecoverable) disk pages performed by that program cannot be undone. So if filing system programs do use operations on unrecoverable disk pages inside a recovery block, then the effects of these operations will not be undone when an alternative of a recovery block is backed out. However, if these filing system programs were not allowed to perform operations on disk pages inside recovery blocks then that would be a restriction which defeats the aim of building a reliable system by using recovery blocks in programs. There is always the possibility of errors occurring inside procedures of the filing system, so recoverability for disk pages, inside the filing system procedures, would be valuable. It is therefore reasonable for the procedures to be able to use recoverable disk pages (for their own purposes). Consequently, there seems to be a dilemma, because the underlying machine must provide unrecoverable disk pages for other reasons.

To cope with this problem a new mechanism has been developed, which is described in this section. The general problems and constraints of PRI-levels, where unrecoverable types and recoverable types are used, are described.

3.7.1 The logging mechanism

If a program operates on objects of unrecoverable types and does so inside a recovery block, then it is that program's own responsibility to make sure that it can continue usefully if it is backed out and forced into its next alternative.

Programs that use objects of unrecoverable types inside a recovery block have to restore these objects themselves if this is required, inside the next alternative. This, however, is not the way in which recovery blocks are intended to be used. Also, if the next alternative has to undo the actions of the previous alternative explicitly then this undoing may well be incomplete or incorrect. Consequently it is preferable to devise a mechanism with which it could be automated.

What is required is that whenever operations on objects of unrecoverable types are performed inside a recovery block (see also Fig.3.9) these operations have

1. to know that they are inside a recovery block,
2. to record sufficient information in a local "cache" in order to be able to undo the effects of operations on objects of unrecoverable types. This cache is local to the programs operating on the unrecoverable types and is not associated with the user, because this local cache provides recovery for these programs, rather than for the user.

This local cache is called log, to distinguish it from the cache which is used to denote the mechanism used to restore objects of recoverable types provided to the user (i.e. the next level).

A special mechanism has been designed in order to achieve these goals. The mechanism is called the logging mechanism (see Fig.3.9), its basic principles being as follows:

1. Whenever unrecoverable types are operated upon, these operations are logged (by the programs that invoke the operations) in order to make possible the restoration of the states of these types to the states they were in when the recovery block was entered. This logging is done in the same way as cacheing, only logging is on behalf of the programs themselves, while cacheing is done on behalf of the next level up. Logging is for programmed recovery, cacheing is for automatic recovery for the next level up.

2. An ENTER-procedure is provided to initialise the logging mechanism when a recovery block is entered. This could for example include the placing of a barrier in the log (similar to cache-marks (Hor74)) and perhaps the increasing of a global variable such as "current-level" to indicate the level of nesting of recovery blocks.
3. An UNDO-procedure is provided to process the log and undo operations performed on objects of unrecoverable types such that the program can continue sensibly if that program is to be backed out.
4. An ACCEPT-procedure is provided to process the log (and objects of unrecoverable types if necessary) after a recovery block in a program (using these objects of unrecoverable types) is exited successfully. (See also the filing system described in a following chapter.)

The basic idea is that programs operating on unrecoverable types can specify a new recovery block mechanism which is the original one extended with the automatic invocation of ENTER, ACCEPT and UNDO at the appropriate places. The way in which this could be implemented depends on the architecture of the underlying machine and the high level language compilers available.

Using this concept an error generated in a program (see also Fig.3,9) causes the UNDO of the underlying machine to restore all recoverable objects and the UNDO-procedure operating on the log in the same level as the program, to restore unrecoverable objects used.

One way in which this could be implemented in, for example, a SIMULA-like language is by including the ENTER, UNDO and ACCEPT procedures in classes providing new (recoverable) types, which are mapped onto unrecoverable types. The operations on the new type could be made to perform logging. The underlying machine could then provide a facility whereby it executes these three procedures when necessary as part of the recovery block processing for programs that use this class as a prefix.

A very similar concept is used for cacheing elsewhere (BaS77) to invoke reverse-procedures of previously executed operations, that are part of a resource allocation mechanism. This implementation requires a mechanism provided by the underlying machine whereby a (compiled) program can specify reverse-procedures to be executed in case of a subsequent error occurring in the current recovery block.

A second way in which this logging mechanism could be implemented is by enveloping the given alternatives of recovery blocks by these three procedures in the proper

manner. (What this involves is shown in the next section.)
 This enveloping could be done by a compiler.

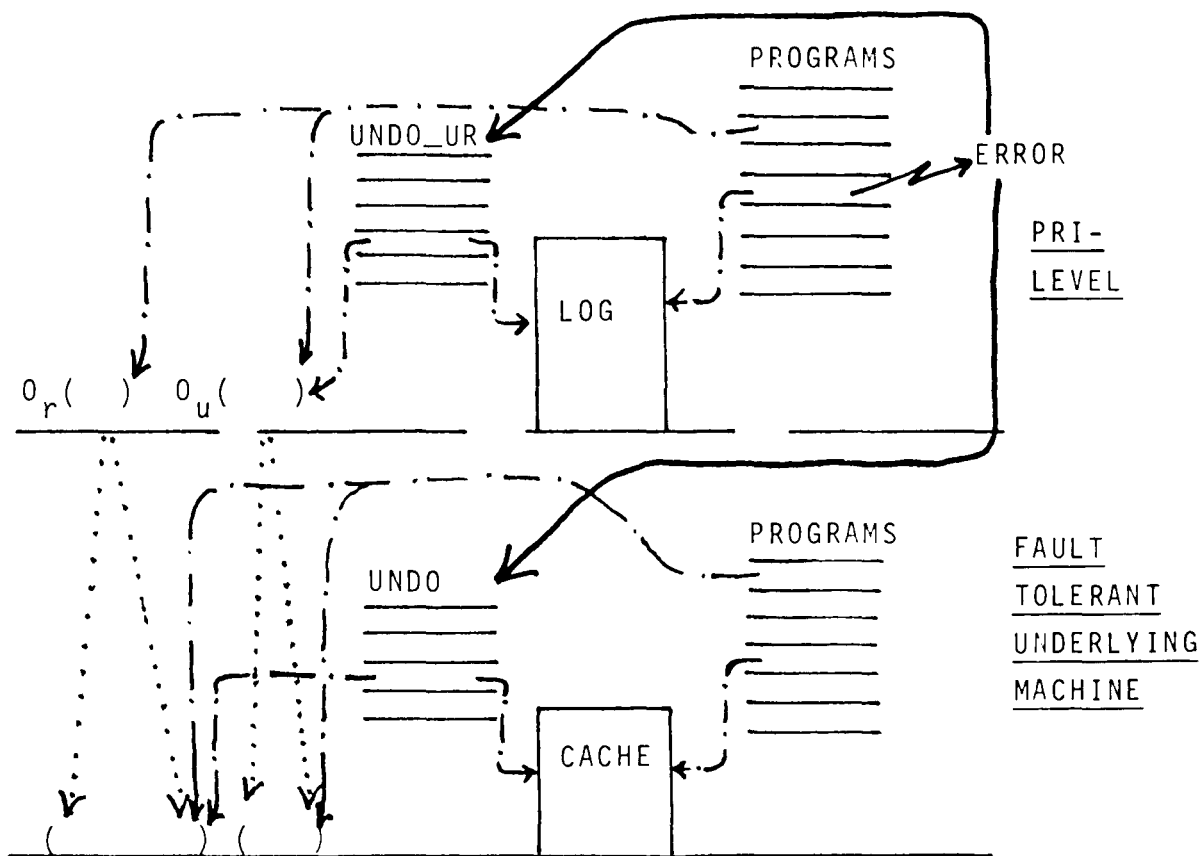


Fig.3.9, A PRI-level using the logging mechanism.

Where: $O_u(t_l, \dots, t_p)$ = A set of unrecoverable objects of types t_l, \dots, t_p .

$O_r(t_q, \dots, t_r)$ = A set of recoverable objects of types t_q, \dots, t_r .

- > denotes: program invocation
-> denotes: type mapping (representation)
- - - - -> denotes: program uses data pointed at

In principle there is very little difference between these two implementations. In the first implementation the underlying machine invokes the UNDO-procedure when a program is backed out, while in the second case the code is organised in such a way that UNDO is executed after an alternative has been backed out, and before the next alternative is invoked. Similarly ENTER and ACCEPT are invoked at the appropriate places.

The second implementation is used in this section to illustrate the logging mechanism. A multi-level system in which this implementation has been used is described in section 3.8.

3.7.1.1 An implementation of the logging mechanism

It is assumed that the underlying machine provides recovery block facilities. These facilities are used to provide an augmented recovery block mechanism (see Fig.3.10). This new recovery block mechanism envelops the n alternatives of a recovery block given by the user. In doing so the mechanism forms $(n+1)$ new alternatives out of the n user alternatives. The way in which this is done is described below (the ENTER, ACCEPT and UNDO-procedures associated with the unrecoverable types are called ENTER_{UR}, ACCEPT_{UR} and UNDO_{UR} respectively), and shown in Fig.3.10 :

1. The first operation in the second through last alternatives will be the UNDO_{UR} operation which, using the log, undoes the effects of operations on unrecoverable types performed in the previously executed alternative.
2. The first operation in the first alternative and second operation in the second through last alternative is the ENTER_{UR} operation which tells this PRI-level that a new alternative is entered.
3. The next operation in each new alternative is the invocation of the user defined alternative.
4. The next operation in each new alternative is the invocation of the user defined acceptance test. The result is stored in the local variable b .
5. The last operation in each new alternative is the execution of ACCEPT_{UR} if and only if b is true. ACCEPT_{UR} takes the appropriate actions on the log and unrecoverable data after a successful acceptance test.

6. An extra new alternative, alternative (n+1), is constructed consisting of UNDO_UR followed by an operation that sets b to false.

The new recovery block structure passes the variable b and the new (n+1) alternatives as the acceptance test and the alternatives to the original recovery block structure provided by the underlying machine. The new recovery block structure is thereby provided. In order to show how the mechanism works the structure of the code at run time providing the new recovery block structure is shown in Fig.3.10 using a BCPL-like (Ric69) notation.

A compiler for a SIMULA-like language could produce "object" code which would basically look like the code shown in Fig.3.10. The notation in a SIMULA-like language that could be used to implement a level providing a recoverable filing system mapping recoverable files onto unrecoverable disk pages provided by a class "disk_system" is shown in Fig.3.11. The recovery class, as the level is called, would be used as a prefix by programs wanting to use the recoverable filing system. The compiler would envelop the users recovery block alternatives with the enter, accept and undo procedure as shown in Fig.3.10. User programs could use the filing system operations such as create_file and read_file.

If all of the programs operating on the unrecoverable types now use this new recovery block structure instead of the one provided by the underlying machine, then the undoing of the effects of operations on unrecoverable types is provided as necessary.

When programs operate on unrecoverable types they must make entries in their log if these operations are performed inside a recovery block. Whether or not an operation is performed inside a recovery block can be checked by testing a variable "current_level" which is maintained by ENTER_UR, UNDO_UR and ACCEPT_UR.

```
let RECOVERY_BLOCK_NEW(origAT,origalt1, ... , origaltn) be  
  f( let b = undefined;  
    let newalt1 be  
      f( ENTER_UR;  
        origalt1;  
        b := origAT;  
        if b then ACCEPT_UR;  
      f)  
    let newalt2 be  
      f( UNDO_UR;  
        ENTER_UR;  
        origalt2;  
        b := origAT;  
        if b then ACCEPT_UR;  
      f)  
    .  
    .  
    .  
    let newaltn be  
      f( UNDO_UR;  
        ENTER_UR;  
        origaltn;  
        b := origAT;  
        if b then ACCEPT_UR;  
      f)  
  
    let newaltn_1 be  
      f( UNDO_UR;  
        b := false  
      f)  
  
    let newat = valof f( resultis b f)  
  
    RECOVERY_BLOCK(newat,newalt1, ... ,newaltn,newaltn_1)  
  f)
```

where: RECOVERY_BLOCK(AT,alt1, ... ,altn) is the recovery block structure provided by the underlying machine. AT is the acceptance test and alt1 to altn are the alternatives.

Fig.3.10: The new recovery block structure used for the logging mechanism.

```
disk_system recovery class filing system;
  begin integer array cache(1:length);
    integer current_level;
    enter procedure
      begin
        ...
      end
    accept procedure
      begin
        ...
      end
    undo procedure
      begin
        ...
      end
    procedure create_file(f, ... );
      begin
        .
        .
        .
      end
    procedure read_file(f, ... );
      begin
        .
        .
        .
      end
    .
    .
    .
  end
```

Fig.3.11: A "recovery class" in a SIMULA-like notation, as could be used to implement a level providing a new recoverable type.

3.7.1.2 The relation between the logging and the caching mechanism

The logging performed in a level is different from cacheing, which is done on behalf of the user, in the sense that entries are put in the cache associated with the user if the user is inside a recovery block, while entries are put in the log if the level itself is inside a recovery block. Logging in a level has nothing at all to do with a user. Logging is for the PRI-level program's own benefit since it is knowingly using unrecoverable objects within a recovery block.

If a fault tolerant level maps a newly provided recoverable type T onto unrecoverable types (t1, ..., tn) then a cache is maintained in order to be able to undo user operations on objects of type T (see also Fig.3.12). This cacheing will be done in terms of objects of types (t1, ..., tn). In the same way in which information is stored in the cache, to enable undoing on behalf of the user, information can be stored in the log, to enable undoing on behalf of the level using unrecoverable types itself. A fault tolerant level will check before it performs an assignment (on objects used for the representation of recoverable user types) whether cacheing is required; in the same way it can check whether logging is required (when operating on unrecoverable objects, which may or may not be used for the representation of user objects). Also, similar procedures ENTER, ACCEPT and UNDO can be used by this level when a user enters a recovery block or a program inside this level enters a recovery block, when a user passes an acceptance test or a program inside this level does, or when a user fails an acceptance test or a program inside this level does, respectively. For cacheing purposes these procedures operate on the cache, for logging purposes they operate on the log.

When a user (see Fig.3.12) generates an error, the UNDO_{cache} in the PRI-level is invoked to restore objects of type T and the UNDO of the underlying machine is invoked to restore the rest of the recoverable user objects. When a program in the PRI-level generates an error, the UNDO_{log} in the same PRI-level is invoked to restore the unrecoverable objects operated upon and the UNDO of the underlying fault tolerant machine is invoked to restore the recoverable objects operated upon by that program.

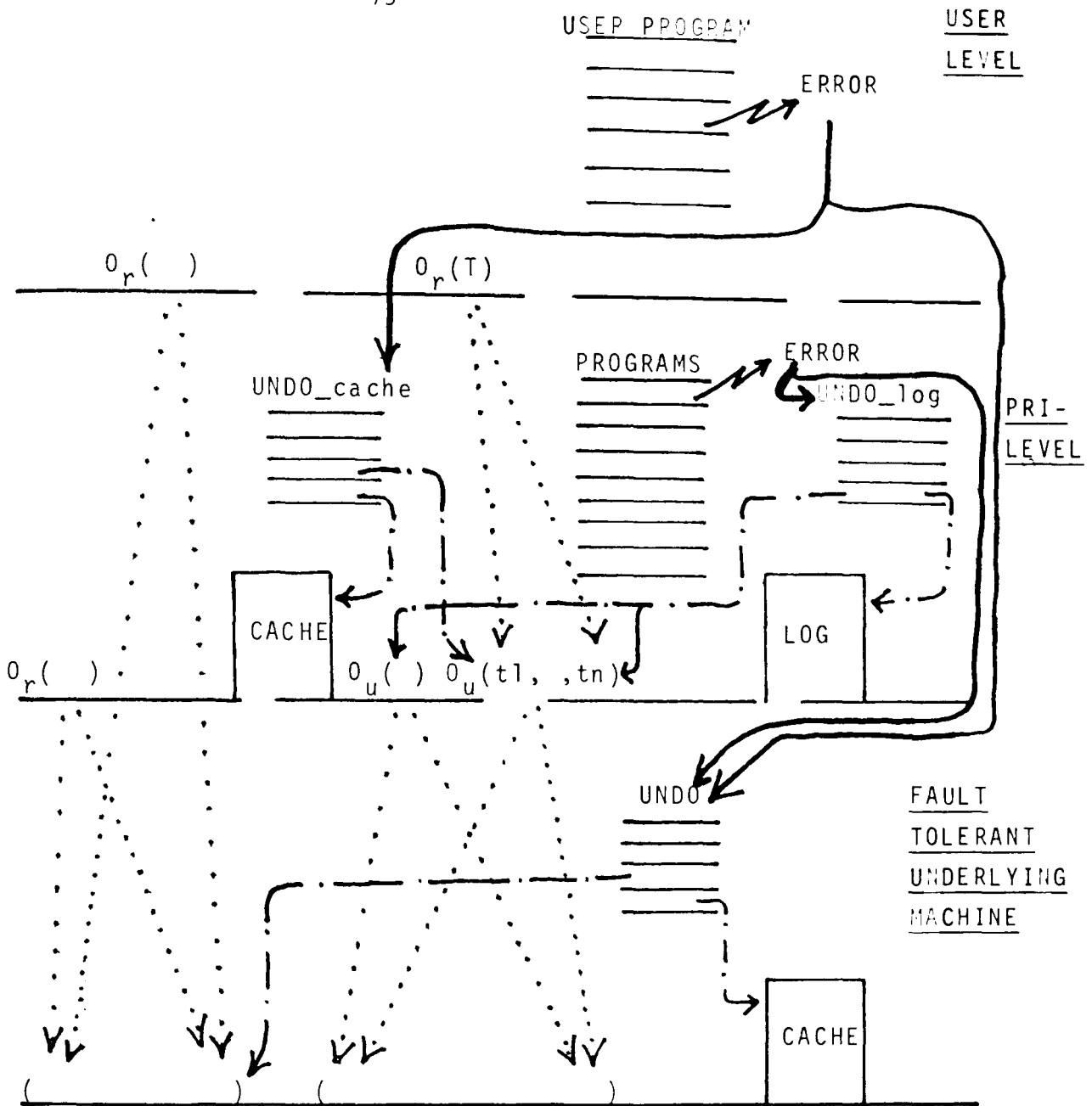


Fig.3.12, A general PRI-level.

Where: $O_u(t1, \dots, tp)$ = A set of unrecoverable objects of types $t1, \dots, tp$.

$O_r(tq, \dots, tr)$ = A set of recoverable objects of types tp, \dots, tr .

—————> denotes: program invocation

.....> denotes: type mapping (representation)

— · — · —> denotes: program uses data pointed at

In the special case where a level satisfies the following three conditions: 1) the level is a set of procedures providing new recoverable types for which it provides recovery explicitly, 2) these types are mapped onto unrecoverable types of the underlying machine and 3) the level does not use unrecoverable types for any other purpose but to represent this new recoverable type, then the logging and caching mechanisms can be combined into one mechanism which serves both purposes. In that case the new recovery block structure provided by the logging mechanism will also be used by the next level as the recovery block structure (see also Fig.3.13). Such a level is called a recoverable type manager. So a recoverable type manager provides recoverable types which it maps on unrecoverable types and also uses the recovery mechanisms that are used to provide the recovery for the user types, for its own purposes. (A recoverable type manager can provide more than one type.) Recoverable type managers are, because of the combining of logging and caching, much simpler than general PRI-levels. The minimum necessary to provide recovery is done while the maximal possible advantage is taken from this recovery mechanism; both the user and the programs providing the new recoverable types can use this recovery mechanism.

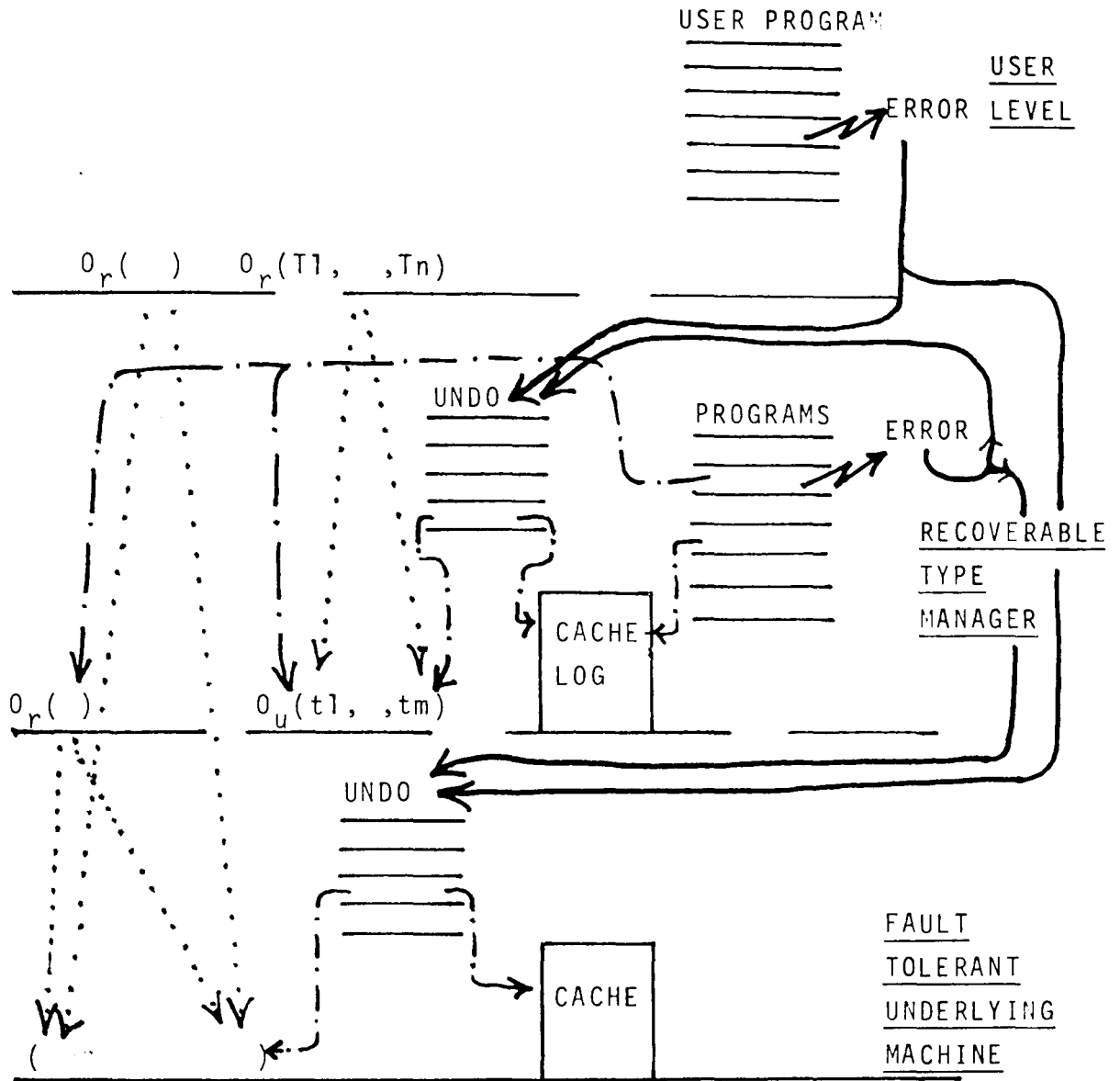


Fig.3.13, A recoverable type manager.

Where: $O_u(t1, ,tp)$ = A set of unrecoverable objects of types $t1, ,tp$.

$O_r(tq, ,tr)$ = A set of recoverable objects of types $tp, ,tr$.

—————> denotes: program invocation

.....> denotes: type mapping (representation)

-.-.-> denotes: program uses data pointed at

3.7.1.3 The properties and constraints of PRI-levels

The logging mechanism (see also Fig.3.10) imposes some requirements and constraints and has some interesting properties which are worthwhile mentioning again explicitly:

1. The UNDO UR operation in the newly formed alternatives has to be the first operation in all but the first of the new alternatives. The UNDO UR operation can not be placed at the end of the new alternatives (as: if b then ACCEPT UR else UNDO UR), because an error occurring within the alternative will cause the alternative to be backed out immediately and the next alternative to be invoked, leaving the operations on objects of unrecoverable types still to be undone.
2. Procedures ENTER UR, UNDO UR and ACCEPT UR, used to provide a new recovery block structure, are ordinary programs in the same level as the programs that use that new recovery block structure. If an error occurs during the execution of one of these three procedures then the underlying machine will restore all of the recoverable types provided and the next (enveloped) alternative will be invoked. The first operation to be executed will then be UNDO UR. If the error in the previous alternative occurred during the execution of UNDO UR then this next alternative may again generate an error during its execution of UNDO UR, or may not be able to function properly. In other words if an error occurs during the log processing then there is likely to be serious trouble.
3. As a consequence of the previous point, one can say that in general the ENTER UR, UNDO UR and ACCEPT UR operations can not use recovery blocks, because they are operating on unrecoverable objects for which no implicit nor explicit recovery is provided when they operate on them. (So if UNDO log in Fig.3.12 or UNDO in Fig.3.13 generates an error then the underlying fault tolerant machine which deals with this error, may have to abandon all higher levels on the machine.)

A solution to this problem could be to organise the log (and perhaps the unrecoverable data) in such a way that the ENTER UR, ACCEPT UR and UNDO UR can simply be retried after failure. An example of this is given by Lampson and Sturgis (LaS76) who describe a system using "intention lists". If the system crashes during the processing of an intention list then the system can simply restart by processing the list from the beginning again. Reprocessing the intention list corresponds with an ACCEPT UR or UNDO UR procedure being retried using the same log.

4. This mechanism copes with all of the problems of PRI-levels described in previous sections. All that is involved is:
 - a. The provision of a new recovery block structure, which, as shown above, is very simple.
 - b. The maintenance of a log, which means making entries in a local data structure plus providing an ENTER_UR, ACCEPT_UR and UNDO_UR operation. If the unrecoverable types operated upon are used to represent a newly provided recoverable type, then these three operations are necessary anyway (but operating on a cache).

5. This mechanism does not introduce a new intermediate level between the underlying machine and the PRI-level using the logging mechanism. This mechanism does not introduce new types either. The log is a data structure used by both the procedures ENTER_UR, ACCEPT_UR, UNDO_UR and the level using the unrecoverable types. The mechanism processes the log and possibly restores objects of unrecoverable types. It does so with a built-in knowledge of what this level does with these types. For example, if the buffer system mentioned previously uses unrecoverable arrays to provide recoverable buffers in the new interface then the logging mechanism will have to provide recovery for those unrecoverable arrays. Since the level knows what the unrecoverable arrays are used for, it can optimise recovery. In other words the mechanism can take advantage of the fact that many representations may exist for one abstract state as seen by the user, in the same way as is done for caching on behalf of the user. Thus if a buffer was free before the buffer management level entered a recovery block then it may not have to restore the contents of the unrecoverable array if it fails an acceptance test after having used that array inside the recovery block, because it knows what the array is used for (namely for the representation of a buffer) and that the concrete state is not important.

The basis of the mechanism is that when this level is backed out and the next alternative of its current recovery block is entered, then the underlying machine will have restored the recoverable types to exactly the states they were in when the previous alternative was entered. The logging mechanism is used to undo the effects of operations on unrecoverable types (see also Fig.3.9). However, it may possibly not restore the states of the unrecoverable types exactly as they were when the previous alternative was entered. (UNDO_UR may be "cleverer" than that.)

3.7.2 The data structure of the log

A PRI-level keeps track of its operations on objects of unrecoverable types inside recovery blocks by using a log as shown above. When a level is forced into an alternative then it uses this log to undo the effects of these operations.

If such a log were constructed from recoverable data structures then an undo action by the underlying machine, which is performed when the level under consideration is backed out and forced into a next alternative, would also undo all of the assignments performed on the log. In other words, the log would be identical at the beginning of each alternative of a recovery block. Thus all the information entered in the log would be lost by the time this level wants to undo its operations on unrecoverable types and is therefore unable to do so. Consequently, the log used by a PRI-level must be constructed from unrecoverable data.

A possible solution avoiding the need for an unrecoverable log is to use the log in a check-point fashion. This means that the values of the objects that may be operated upon are copied just before the recovery block is entered. Obviously this may lead to gross inefficiency, because it may not be known in advance which objects will be operated upon and which not. This scheme is therefore not considered any further.

Another solution is to extend the recovery block features to include the facility to declare UNDO-procedures (and ACCEPT- and ENTER- procedures). If the UNDO_{UR}-procedure could be invoked by the underlying machine before any objects are restored to their previous states, then the log would not have to consist of unrecoverable data objects. The programmer would then have to specify the enter, accept and undo procedures which would have to be executed as part of the recovery block processing when necessary. The fault tolerant machine on which the (compiled) code is to run must provide (virtual) instructions that make it possible to "declare" enter, accept and undo procedures. The underlying machine would place the entry points of these procedures in a cache area and invoke them when necessary (similar to the use of recoverable procedures as described elsewhere (Hor74)). The compiler would produce code with the special (virtual) instructions provided to "declare" the enter, accept and undo procedures to the underlying machine.

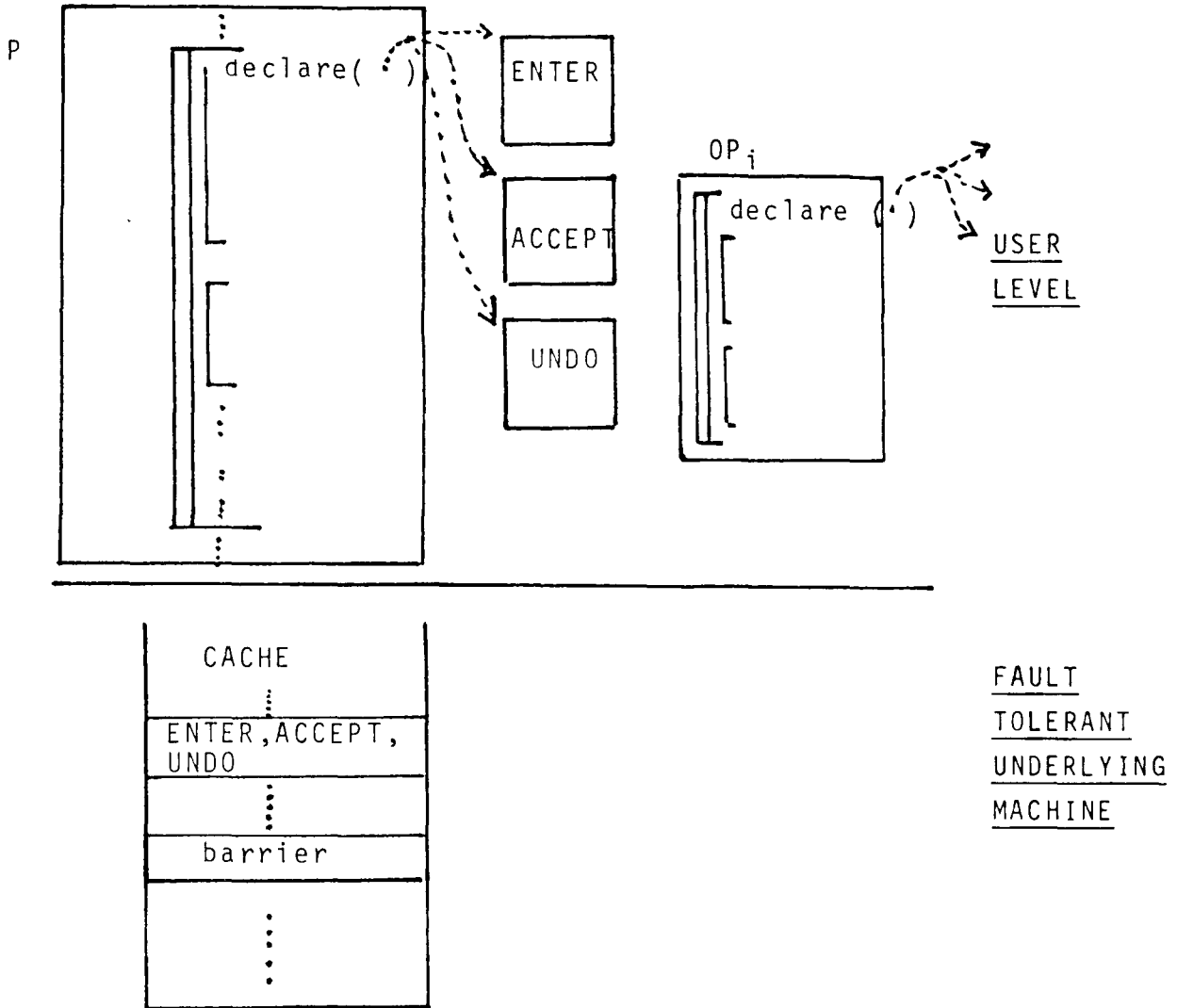


Fig.3.14: An example to show a possible implementation of a fault tolerant machine providing the possibility to "declare" enter, accept and undo procedures.

An implementation of this scheme could be as described below and illustrated in Fig.3.14.

Suppose that program P specified ENTER, ACCEPT and UNDO as enter, accept and undo procedures. The object code of P could, for example, contain the necessary "declare" instructions at the start of each recovery block, which causes the fault tolerant underlying machine to put the entry points of ENTER, ACCEPT and UNDO in the cache. The underlying machine starts executing P and if P generates an error inside a recovery block then the underlying machine will first execute UNDO and then process the relevant cache area and invoke the next alternative.

Suppose that operations OP1 to OPn are not declared inside P. The object code of recovery blocks in OP1 to OPn could, for example, "declare" other enter, accept and undo procedures. Some appropriate scope rules are needed to make sure that the correct enter, accept and undo procedures are called during execution of recovery blocks in P, OP1 to OPn and other (lower level) operations used in OP1 to OPn. These problems, however, are not the subject of this thesis, but are involved in the scheme described by Banatre and Shrivastava (BaS77).

The fact that the log used by a PRI-level must consist of unrecoverable data has the following important consequences:

1. If a PRI-level corrupts its own log then no recovery is possible at this level, because it is unrecoverable. As mentioned in the previous subsection this will most likely give serious trouble. Consequently it is important to protect these logs from corruption and to ensure that log operations are very reliable. A lot more protection is needed for logs in general than for caches (associated with users) in interpreter-levels, because these caches can be recoverable (if the level implementing them runs on a fault tolerant machine). If such a level corrupts a cache then it can simply fail its acceptance test and the underlying machine will undo the corruption and force the level into its next alternative as described in section four.
2. It is not possible for a PRI-level to know which types are required by a higher level for the use of a log. Because of this it would appear to be preferable to let that PRI-level provide unrecoverable objects of the same types as the recoverable types that it implements. An example of this was described in section 6. It is clear that this is an important point which will have to be taken into account when building recoverable multi-level systems.

Another yet unmentioned problem in PRI-levels is that updating of the log and operations on unrecoverable objects always have to be organised such that an alternative can be backed out at any time. For example, if an object is updated and subsequently the log entry is made and an error occurs in between these two events, then the operation on that object will not be undone. The problems of providing complex new types out of unrecoverable types are addressed in chapter four.

It is clear that a cache in a procedural PRI-level has to be constructed from unrecoverable data for the same reasons as logs do (see Fig.3.12).

3.7.3 Systems consisting of PRI-levels

Fig.3.10 showed how a new recovery block structure can be constructed from an existing recovery block structure, such that the recovery block facilities of the existing recovery block structure are extended. Here it is shown how different recovery block structures can be used and constructed in different kinds of PRI-levels.

I) A general PRI-level.

To understand the recovery block structures that have to be implemented by a general PRI-level, as shown in Fig.3.12, the following two things have to be considered:

1. Programs may operate on unrecoverable types that are not used for the representation of newly provided recoverable types. A logging mechanism is therefore required.
2. Logging the changes made to the unrecoverable types operated upon by a PRI-level, has in principle nothing to do with the recovery of the newly provided types. So the cacheing mechanism has to be used.

Therefore two different recovery block structures are implemented by a general PRI-level as shown in Fig.3.15: RECOVERY BLOCK NEW and RECOVERY BLOCK USER. RECOVERY - BLOCK NEW is the recovery block structure provided by the logging mechanism, and used by the programs internal to the PRI-level. RECOVERY BLOCK USER is the recovery block structure provided to the user. The ENTER, ACCEPT and UNDO procedures are the procedures that process the cache and the recoverable types provided to the user when the user enters a recovery block, passes an acceptance test or generates an error, respectively. The ENTER UR, ACCEPT UR and UNDO UR procedures similarly operate on the log for programs in the general PRI-level.

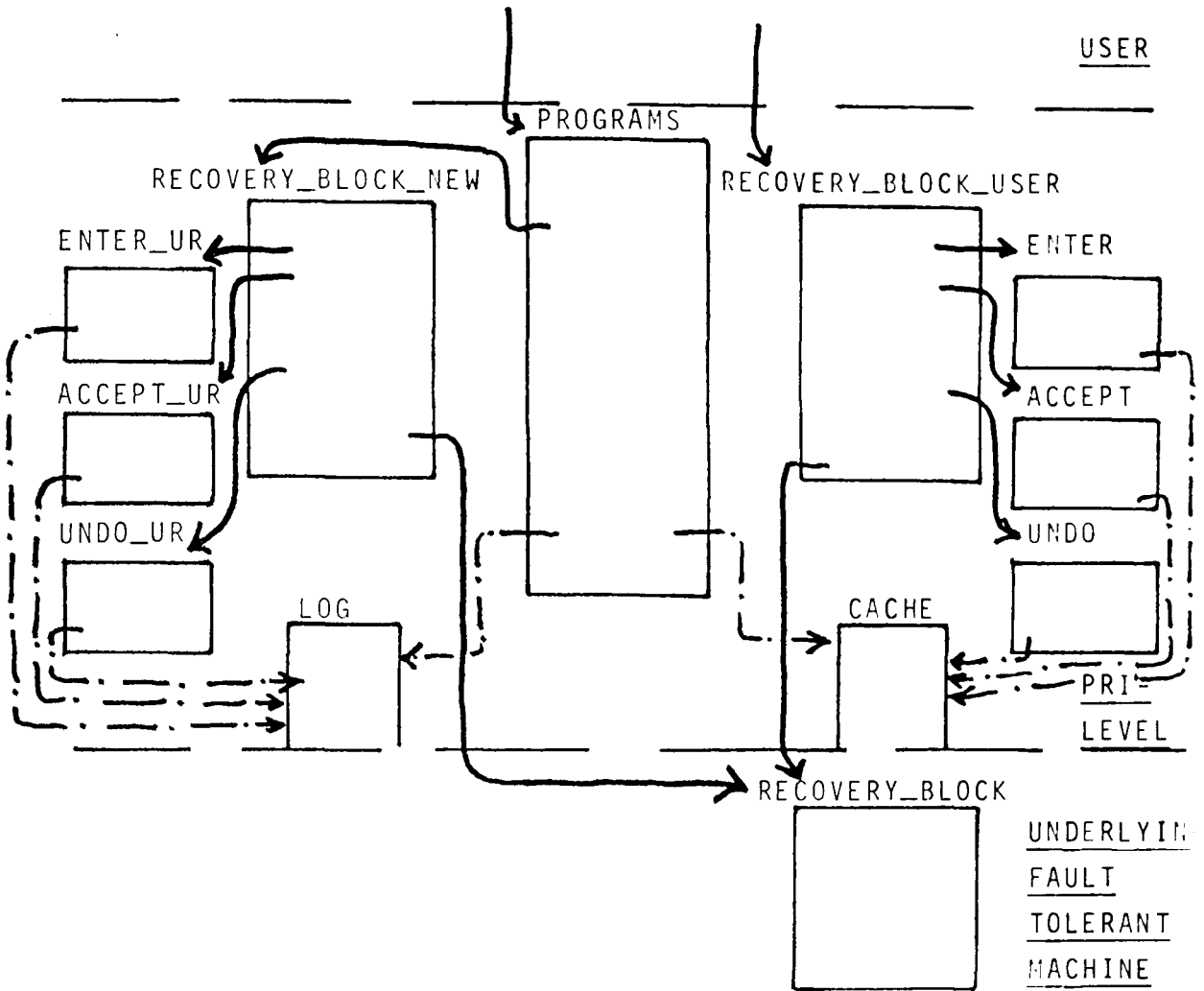


Fig.3.15, A general PRI-level.

Where:

- > denotes: program invokation
--> denotes: type mapping (representation)
- .-.-> denotes: program uses data pointed at

II) A recoverable type manager.

As described before and shown in Fig.3.13, a recoverable type manager maps the newly provided types on

unrecoverable types only. So the logging mechanisms is sufficient to restore values of objects of the new types provided to the user. If no other objects of unrecoverable types are used by the PRI-level then no further logging will be needed, so the log and the cache can be combined to serve a dual purpose. The same mechanism can, in that case, be used by the PRI-level and be provided to the user, as shown in Fig.3.16. (One data structure is now used as both log and cache and should therefore be called, for example, a cache-log. However, the term log will also be used for these "cache-logs" in this chapter.)

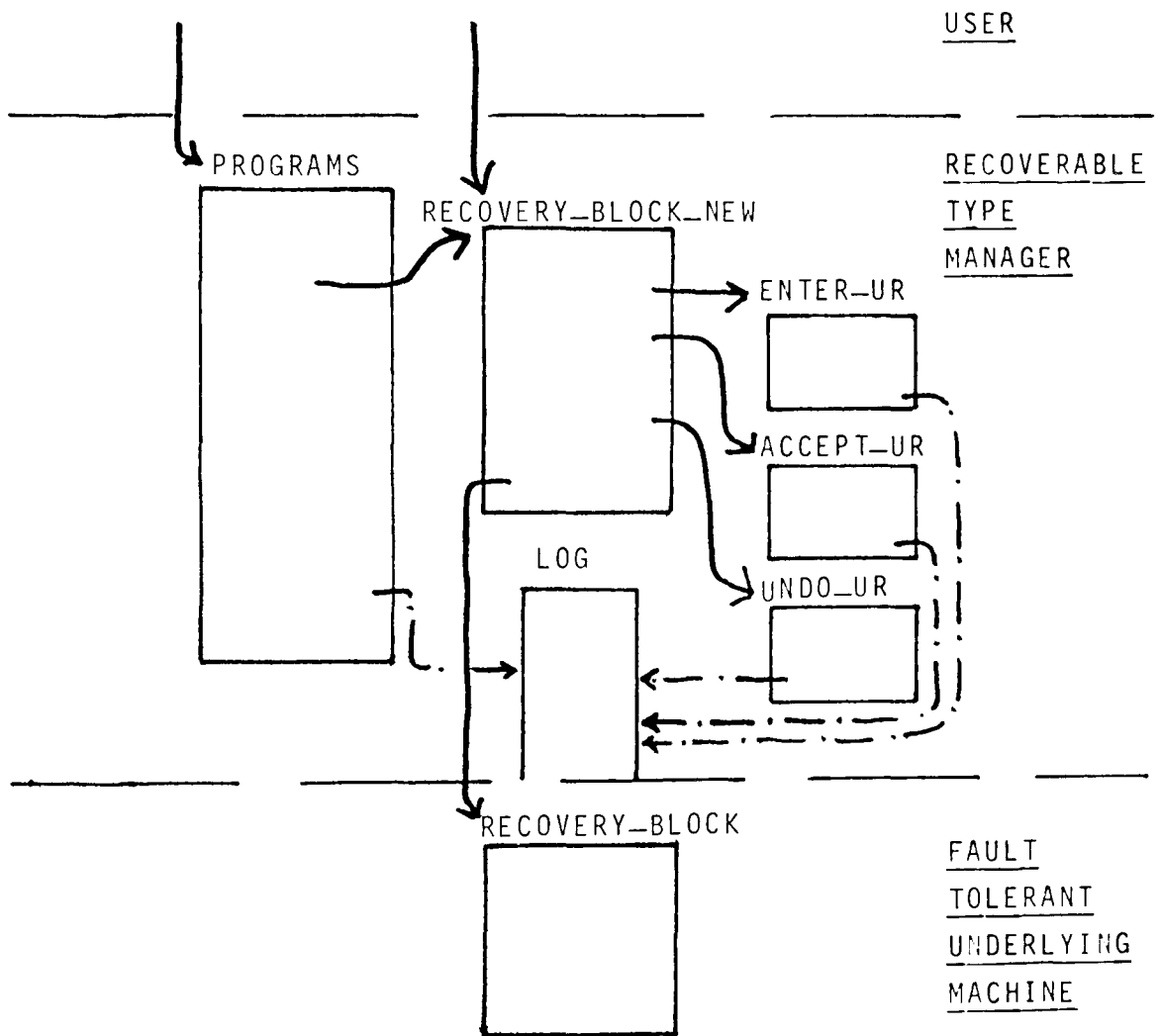


Fig.3.16, A recoverable type manager.

Where:

- > denotes: program invocation
-> denotes: type mapping (representation)
- .-.-.-> denotes: program uses data pointed at

The recovery mechanism in a recoverable type manager will only have to undo the damage done to objects of the types provided to the user. So internal and user recovery both restore user types. Restoring user types is sufficient for internal recovery, because the unrecoverable objects operated upon are used only for the representation of objects of the new user types. An interesting example showing how this can be done is described in chapter four.

Programs have no way of knowing whether the recovery block structure used is provided by a logging mechanism in a recoverable type manager or whether they are running on a fault tolerant interpreter.

III) A generalization of a recoverable type manager.

As mentioned before, it may be desirable to provide recoverable and unrecoverable objects of a type (see, for example, section 6 under 2.). If a recoverable type manager implements new type T1 to Tn, which it maps onto unrecoverable types, and it can provide recoverable and unrecoverable objects of these types then the following structure of the level, as shown in Fig.3.17, is required:

- * A recovery block structure RECOVERY_BLOCK_UT, is needed to provide local recovery in the recoverable type manager for unrecoverable objects used for the representation of unrecoverable user objects.
- * A recovery block structure RECOVERY_BLOCK_USER is provided to the user to provide recovery for recoverable objects of types T1 to Tn.
- * This level (the recoverable type manager), however, needs both of the previous recovery mechanisms. So the structure required is one that provides recovery for the unrecoverable objects used to represent recoverable user objects and for the unrecoverable objects used to represent unrecoverable user objects. Another recovery block structure (RECOVERY_BLOCK_RT) is therefore built to extend the recovery provided by RECOVERY_BLOCK_UT with recovery for the unrecoverable objects used to represent recoverable user objects.
- * The same recovery mechanism for providing recovery for recoverable user objects and for providing local recovery for the objects on which they are mapped can be used (because the level is a recoverable type manager). The log, ENTER, ACCEPT and UNDO used to build RECOVERY_BLOCK_RT are exactly the same as those used for RECOVERY_BLOCK_USER. RECOVERY_BLOCK_RT is the recovery block structure used by programs in the level.

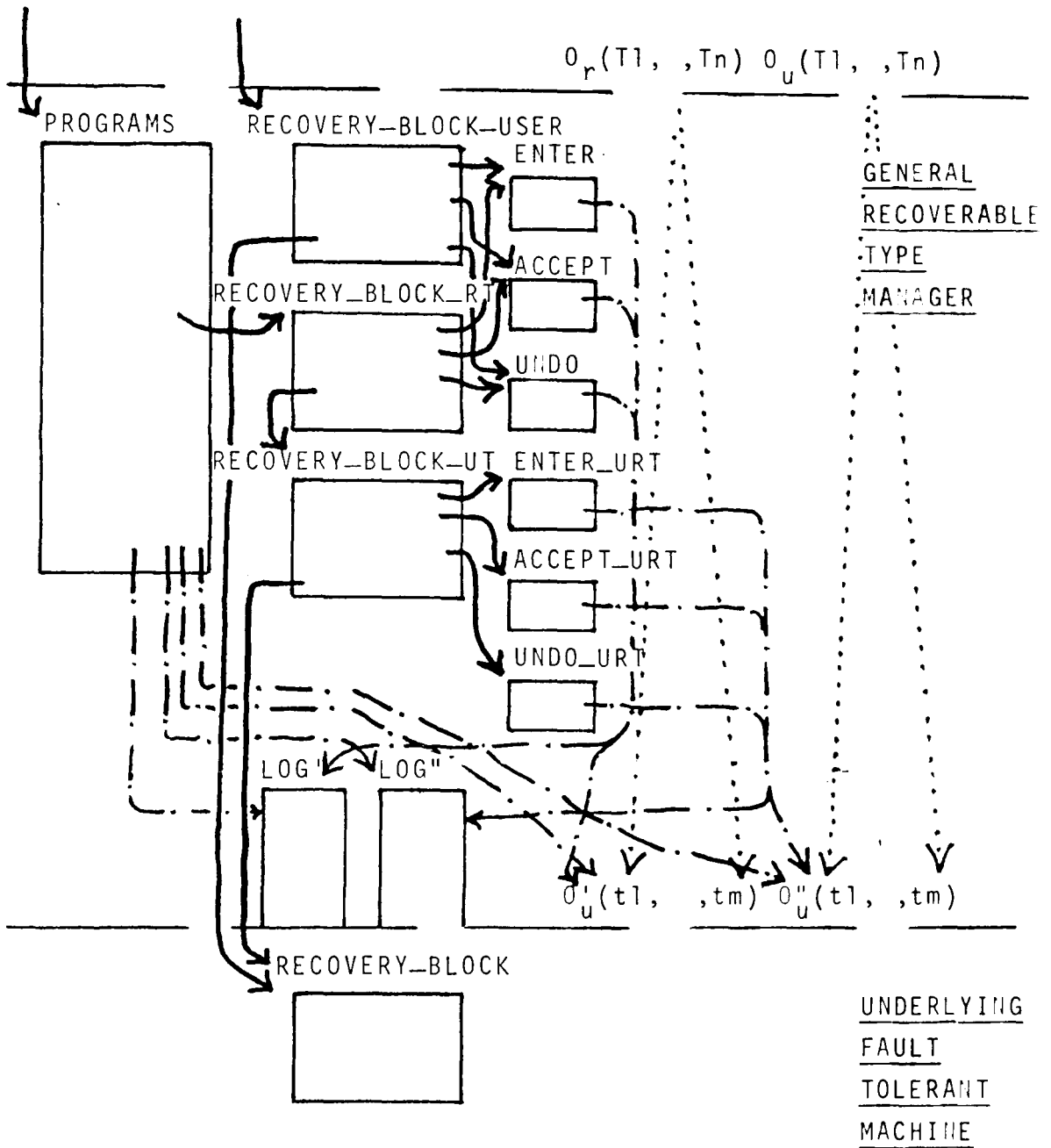


Fig.3.17, A recoverable type manager providing recoverable and unrecoverable objects of the same type.

Where:

- > denotes: program invocation
-> denotes: type mapping (representation)
- - - -> denotes: program uses data pointed at

The schemes described here do not require complex scope rules or a complex cacheing mechanism in the interpreter providing the basic recovery block structure onto which higher levels built new mechanisms. Once a designer or programmer understands the basic principles described in this chapter, then levels are very easy to construct.

3.8 A two-level prototype system

In order to show that the ideas put forward in this chapter can be implemented, a two-level prototype system consisting of two PRI-levels has been designed and implemented successfully.

The first level of this system consists of a fault tolerant interpreter for recoverable basic OCODE (a description of the OCODE machine has been given elsewhere (Ric71)). This interpreter runs on a Burroughs B1700 machine and is written in the microprogramming language BML (DeW73).

The interpreter provides recoverability for OCODE variables. All of these variables are mapped onto (virtual) machine words. The cache records changes made to the machine words used for the representation of recoverable OCODE variables. Input/output operations provided by this first level are unrecoverable. Consequently the type disk page, which is also provided, is unrecoverable.

The interpreter provides an instruction:

```
RECOVERY_BLOCK_VARS(AT,alt1, ... ,altn)
```

```
Where: AT = the start address of the acceptance  
test
```

```
alti= the start address of alternative i
```

This instruction provides the recovery block structure which corresponds to the notation (Ran75):

```
ensure AT;  
by alt1;  
.  
.  
.  
else by altn;  
else error ;
```

A full description of the implementation of this interpreter is given elsewhere (Ver76).

The interpreter also provides the instructions ERROR, and PRIOR. The PRIOR operation takes one argument which is the address of a variable; the value the variable had at the time the current recovery block was entered is returned.

This instruction can be particularly useful in acceptance tests.

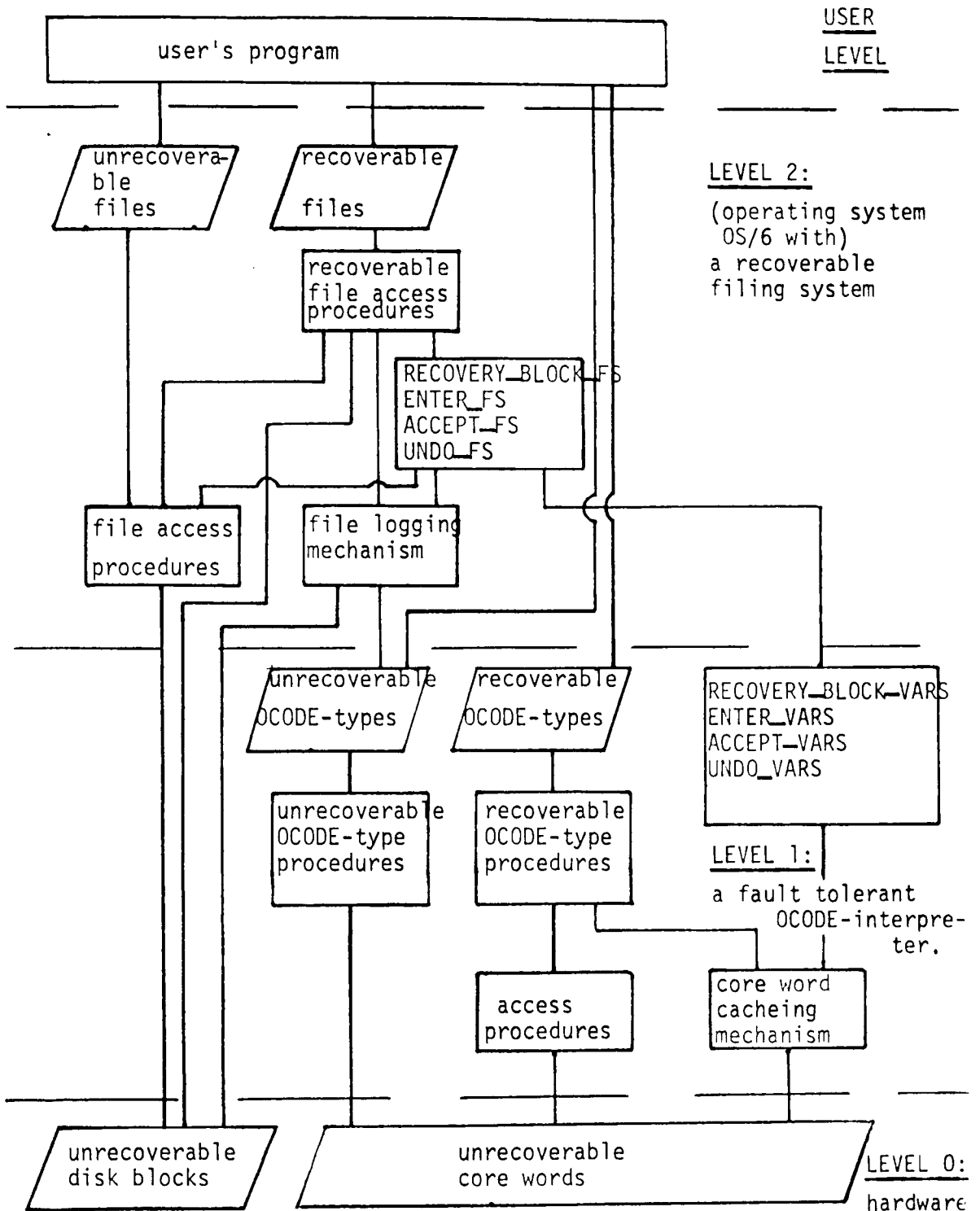
The second level of the prototype consists of BCPL-programs that implement a recoverable filing system. (OCODE is the result of a BCPL-program compilation.) This level provides recoverable files which it maps onto unrecoverable disk pages, provided by the first level. This second level uses a logging mechanism as described in the previous section and shown in Fig.3.10. A full description of the filing system and the way in which recoverability has been provided for files is given in chapter four.

The logging mechanism provides a new recovery block structure: RECOVERY_BLOCK_FS(AT,alt1, ... ,altn) in the manner described above. Programs of the filing system use this new recovery block structure. The filing system is a recoverable type manager, so users use the same recovery block structure as the filing system, namely RECOVERY_BLOCK_FS.

The filing system consists of the following parts:

1. Procedures ENTER_FS, ACCEPT_FS, and UNDO_FS, which take the necessary actions for the filing system if a recovery block in a filing system program or user program is entered, an acceptance test has been successful or unsuccessful respectively (these procedures are particular examples of the ENTER_UR, ACCEPT_UR and UNDO_UR in Fig.3.10).
2. File operations for recoverable files. These operations are log-oriented, which means that they maintain sufficient information in the log to be able to restore the states of the files used.
3. RECOVERY_BLOCK_FS providing the new recovery block control flow structure.

The total structure of the two-level machine is shown schematically in Fig.3.18.



Where:- parallelograms indicate data types occurring in interfaces between programs.
 - rectangles indicate program modules performing a mapping.
 - solid lines indicate invocations of programs by other programs (rectangles may occur on those solid lines between two programs to indicate the types used in the interface between those two programs).
 - dashed lines are used to separate levels.

Fig.3.18, The structure of the two level prototype system.

3.9 Recoverable types as a basic concept for recoverable multi-level systems

Recoverable type managers can easily be added to a system. In other words the machine can be extended with new levels each adding new recoverable types to an existing interface by mapping them onto unrecoverable types of the underlying machine (so long as the underlying machine provides the recovery block structure).

In order to show the flexibility with which recoverable type managers can be used to build up a recoverable multi-level system, another example is given below.

3.9.1 A recoverable lineprinter manager on top of the recoverable file manager

Suppose that the two-level system described in the previous section provides an unrecoverable lineprinter with one (unrecoverable) operation: 'send_char_to_printer'. In order to provide a recoverable and more abstract printer a set of programs providing operations such as 'print_integer', 'print_file', 'new_line', and so on, are written. When these operations are invoked the output is not printed immediately, because the operations would then be unrecoverable. Instead the output is spooled to a recoverable file. If the outermost recovery block is successfully exited then the spoolfile will be printed.

In this way recovery is not provided by performing operations and undoing their effects later if necessary. Instead the effects these operations would have are recorded and this record will be discarded when necessary. When no further undoing can follow (that is when the outermost recovery block is exited) then the file used to register the effects of lineprinter operations is used to actually print the output. The file used is recoverable, so the effects of user lineprinter operations are undone automatically if a program is backed out.

A recoverable lineprinter manager providing this recoverable lineprinter will consist of the following parts (programs are given in a BCPL-like notation).

1. A procedure 'send_char_to_recoverable_printer' is used to replace 'send_char_to_printer'. This procedure is shown below:

```
let send_char_to_recoverable_printer(char) be  
  f( if pr_level=0 then send_char_to_printer(char)
```

```
        else add_to_file(printer_file,char)
    f)
```

where: pr_level is a recoverable variable the use of which will be explained below.

printer_file is the name of the recoverable spoolfile.

add_to_file is a procedure that appends a character string to a given file.

2. The procedures such as: 'print_integer', 'print_hex', 'print_file', 'skip_page', 'new_line'.

These procedures use 'send_char_to_recoverable_printer' to map information to be printed on a stream of characters for the line printer.

3. The procedures ENTER_PR, UNDO_PR and ACCEPT_PR, which are given below:

```
let enter_pr be f( if pr_level=0 then
                    empty(printer_file)
                    pr_level:=pr_level+1
                    f)
```

```
let undo_pr be f( f) // a null body.
```

```
let accept_pr be f( pr_level:=pr_level-1
                    if pr_level=0 then
                    printfil(printer_file)
                    f)
```

where: empty is a procedure to empty a file,

printfil is a procedure that prints a given file.

The variable pr_level is used to indicate the recovery block level of nesting.

If an error occurs during the execution of a recovery block alternative then 'pr_level:=pr_level-1' will be done automatically, because 'pr_level:=pr_level+1' of ENTER_PR will be undone by the underlying machine. UNDO_PR can be a null procedure, because of the recoverable spoolfile and the recoverable variable 'pr_level' used.

4. The new recovery block structure is provided by 'RECOVERY_BLOCK_PR' which is constructed from 'RECOVERY_BLOCK_NEW' in exactly the same way as 'RECOVERY_BLOCK_NEW' is constructed from 'RECOVERY_BLOCK' as shown in Fig.3.10. The ENTER, UNDO and ACCEPT procedures are given above.

The user can now use the recovery block structure 'RECOVERY_BLOCK_PR' and the recoverable printer. The interface provided by the recoverable file manager is thus

extended with a recoverable printer. The new recovery block structure can also be used by the print procedures in the recoverable lineprinter manager.

As mentioned above, the user of the new system is not aware of the construction of the multi-level system used. The user cannot know whether subsequent levels perform caching and logging in order to provide recoverable types, or whether all of the recoverable types are provided by one fault tolerant interpreter.

3.9.2 Recoverable types versus recoverable operations in a multi-level system

The use of recoverable type managers gives a system structure as shown in Fig.3.19. Just before or after the fault tolerant interpreter providing the basic recovery block structure has restored the values of the recoverable types it provides, the UNDO-procedures of all of the recoverable type managers will be invoked.

Similarly all of the ENTER-procedures will be invoked when an alternative is started and ACCEPT-procedures will be invoked when an alternative finishes successfully. Consequently after an alternative has been executed and, say, failed its acceptance test, then all of the recoverable type managers will check to see if any objects of the recoverable types they provide have been altered. If so they will restore the values of those objects (see Fig.3.19). Consequently, any recoverable type manager that has not been invoked by the alternative will have its UNDO-procedure invoked unnecessarily. However, if any objects of types provided by the recoverable type manager have been altered then the restoring of the states of the objects is independent of the sequences of operations performed; the type manager knows the present state (which may be useful for optimisation of the restoring of the old state) and the state to which to go back to. The original states will be restored using this knowledge only. Methods of performing recovery for complex data structures based on this principle, are discussed in chapter four.

For example, consider a recoverable stack manager. Suppose the user had invoked the operation PUSH ten times and subsequently invoked the operation POP ten times and the program is subsequently to be backed out (and the stack is to be restored to its original state by the recoverable stack manager). The UNDO of the recoverable stack manager will in that case not do anything, because the present state of the stack is identical to the state to which the stack is to be restored.

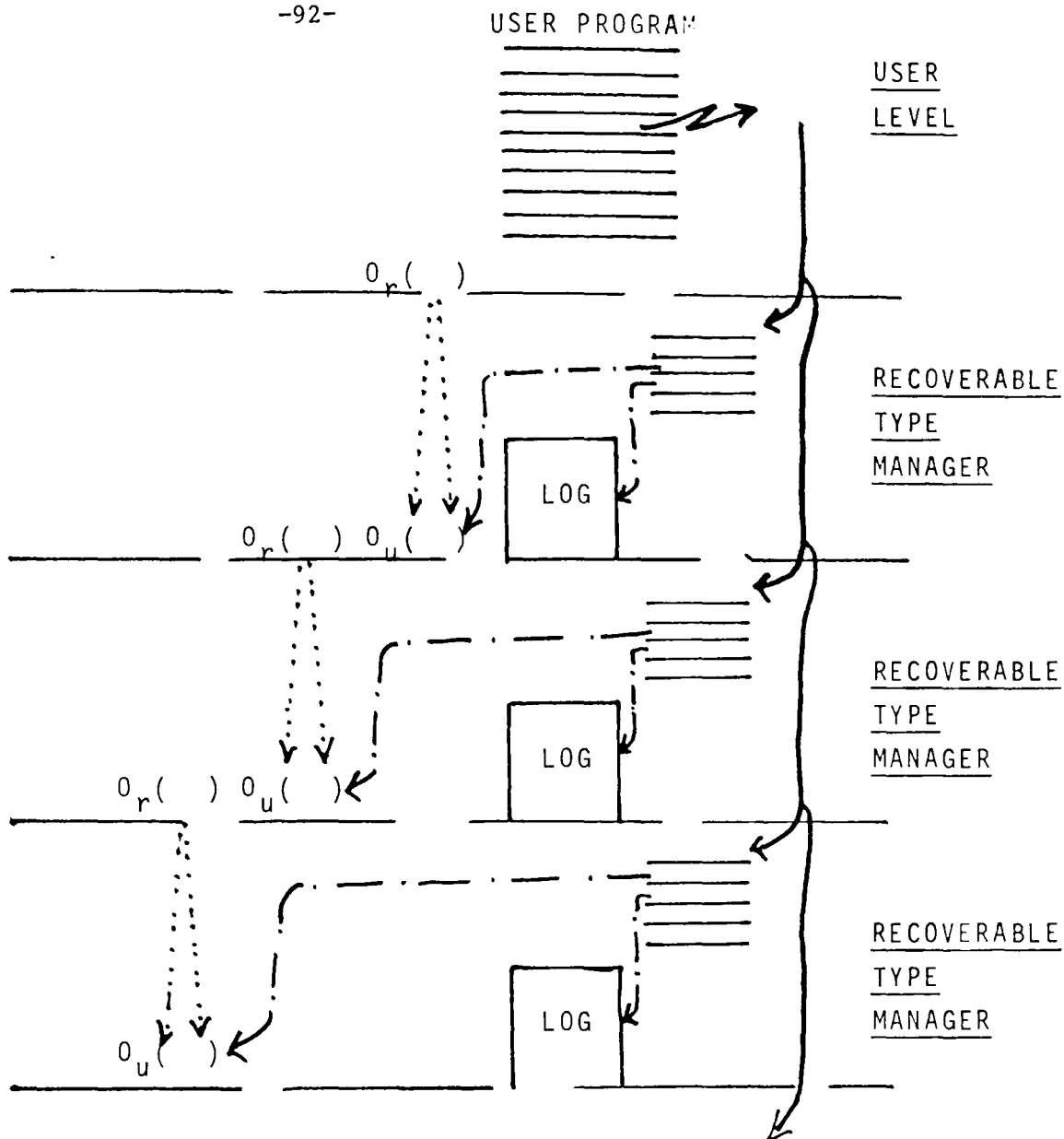


Fig.3.19, A multi-level system consisting of recoverable type managers.

Where: $O_u(t_1, ,tp)$ = A set of unrecoverable objects of types $t_1, ,tp$.

$O_r(t_q, ,tr)$ = A set of recoverable objects of types $t_q, ,tr$.

- denotes: program invocation
-→ denotes: type mapping (representation)
- .-.-→ denotes: program uses data pointed at

This is very different from the approach presented by Banatre and Shrivastava (BaS77) where the recovery block structure is never redefined. In this approach, every operation is made recoverable, so the action on an acceptance test failure is to call the "undo"s of all the operations performed in that alternative. Taking the stack example above, it is possible, in this scheme, to make POP and PUSH operations independently recoverable. The UNDO of the recoverable stack manager would then cause the invocation of operation POP REVERSE ten times and operation PUSH REVERSE ten times. However, if the user had not used the stack at all then the UNDO of the recoverable stack manager would not be invoked at all. The scheme presented by Banatre and Shrivastava is, however, sufficiently flexible so that an optimised recovery scheme for the stack, similar to the one described in the previous paragraph, can be programmed.

Admittedly the stack example is certainly not a yardstick to compare both approaches. The example is only given to illustrate the differences in the approaches. There may well be many environments where the recoverable type manager approach is more efficient than the reverse-operation approach, and vice versa. It is impossible to make a general comparison of the efficiencies of these methods.

It is interesting to note that System R (Ast76) employs the basic principles of both techniques for two different recovery mechanisms, although no nesting of scopes of recovery is possible. In order to provide recovery for segments (collections of logical address spaces used to store the data) two page maps, called the current and backup, are maintained. When a page is updated for the first time in a transaction, its new value will be directed to a new page pointed to by the current page map, while the backup page map and the original pages are left intact. This could be classified as type recovery.

The recovery for transactions is supported through the maintenance of time ordered lists of entries, which record information about each change to the recoverable data. During transaction recovery, the listed entries for the transaction are read in last-in-first-out order. Special routines are employed to undo all the listed modifications back to the recorded save point. This could be classified as reverse procedure recovery.

The recovery for segments is used to create system checkpoints. The listed entries can be used as an audit trail to restore files to their original state, in case of a system crash. The recovery for transactions basically provides a scope within which the user can undo all his operations (a very simple recovery block with one

alternative). The use of "inverse updates" in System R has been justified elsewhere (EsC75). Eswaran and Chamberlin define an integrity subsystem as a subsystem which permits users to make assertions which define the "correctness" of the data base, and to specify the actions to be taken when assertions are not satisfied. An integrity subsystem depends on a logging and recovery subsystem which can selectively back out a given update request. This may be carried out either by performing "inverse-updates" or by keeping a copy of the data base that exists at the start of the execution of the update request and resurrecting the copy.

It is, however, the author's personal view that programs (or systems) should be built based on the data types used (see for example a method described by Jackson (Jac75)). Much research has been done in the area of proving correctness of programs (see chapter one). However, the results so far fall short of a tool for routine use and different approaches have been sought. Much of the recent research has been aimed at proving correctness of data types and data structures (Gut76), (WLS76). Most recent research in protection and security (Den76), (Lin76) has also been based on data types (rather than on programs or program structures and properties). Similarly it is the author's view that recoverability should be provided for types, rather than for pieces of programs executed, for which "reverse-pieces of programs" exist.

3.10 Conclusions and relation to other areas

This chapter has described how recoverability can be linked with types. The presence of both recoverable and unrecoverable types in the interfaces of a multi-level system can be efficiently used to construct a recoverable multi-level system. The use of recoverable type managers, which provide new recoverable types by mapping them onto unrecoverable types in the interfaces provided by the lower level, has been shown to be a flexible, reliable and efficient way to implement levels in a recoverable multi-level system. Programs in such a system operate on both recoverable and unrecoverable types, which leads to non-trivial recovery and consistency problems. A mechanism devised to solve these problems and an implementation of this mechanism in a recoverable two-level system have been described in this chapter.

The scheme proposed has been described as part of a possible way to construct recoverable multi-level systems. The scheme could, as was mentioned already, be used as a part of a mechanism to provide extended types. It was shown how, for example, a recovery class could be specified. This

could be done for other purposes than just the implementation of fault tolerant systems as described in this chapter. If a programmer could also specify invariants or assertions, as for example in ALPHARD (WLS76), then the enter accept and undo procedures, specified as part of a ALPHARD "form", could be invoked before an object of that form was updated, after it was updated successfully (i.e. the assertions and invariants were still valid) or unsuccessfully respectively. If the accept procedure was usually empty and most operations were performed successfully, then overheads would be fairly small. Similarly there are other ways in which the scheme could fit in languages providing type extension facilities. Programs written in these languages will need to be compiled and this chapter has dealt with the concepts and mechanisms needed at run time. Language issues have not been addressed, but are the subject of ongoing work at Newcastle (BaS77).

4.0 RECOVERY FOR COMPLEX DATA STRUCTURES

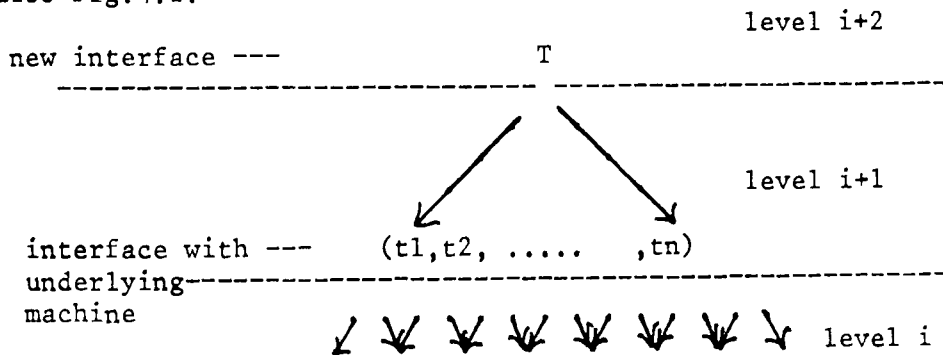
This chapter investigates mechanisms that can be used to provide recovery and consistency for global data structures. Consequently this chapter is basically concerned with the problems of providing recoverability in data base systems and filing systems, or more generally systems providing complex data structures, that remain in existence after the running of a job.

Several mechanisms are described and their advantages and disadvantages are discussed. A recoverable filing system, which has been implemented, is used to illustrate the problems. This chapter also describes how consistency in global data can be maintained at any time, even when recovery is not possible such as when a system crash occurs. Several mechanisms are described and compared.

4.1 Definitions

The terms defined in the previous chapters will also be used in this chapter. Some new definitions necessary for the purpose of this chapter are given.

A multi-level data structure is an abstract data type, provided by levels in a multi-level system, by mapping objects of this type onto objects of one or more types provided by the underlying machines of those levels. See also Fig.4.1.



where: T,t1,t2, ... ,tn are types.

Fig.4.1, A multi-level data structure representation.

The mapping done by level i+1 defines the representation of the new type, T in level i+2.

A multi-level data structure could be represented as a

type T for which $T=Q_1(Q_2(\dots Q_n(t_1,\dots,t_m), \dots))$ where every Q_i is a mappings function providing a new abstract type by representing it by a (possibly composite) data structure. In the multi-level systems envisaged, mappings Q_1 to Q_n could be done in one level. For example, a level implementing a filing system may map type `file` onto `<file_index, file_structure>`. Type `file_structure` may be mapped onto `<file_header, file_body>`. Type `file_body` may be mapped onto type `file_page`. (The filing system of Madnick and Alsop (MaA69), described in chapter three, is an example of this.) The term complex data structure will be used instead of the term "multi-level data structures", in order to avoid confusion with the notion of level as used in the present thesis, because these data structures can be provided by one level in a multi-level system as discussed in chapter three.

The notions of recoverable type and recoverability provided for a type will be used in this chapter to mean that such types can be reset to values held earlier, such as needed for recovery blocks. So in terms of recovery blocks: if a recovery block is used in a program then operations on recoverable types performed inside an alternative will be undone if the acceptance test fails, while operations on unrecoverable types will not be undone. For convenience, recovery is discussed in terms of recovery blocks, although the mechanisms discussed are general recovery mechanisms which need not be part of a system providing recovery facilities in the (syntactic) form of recovery blocks.

Commitment of modifications made to global objects (global to the program performing the modifications) inside nested recovery blocks, occurs when the outermost recovery block is left. The notion of outermost recovery block is used, in general, with respect to data to denote the recovery block outside which no recovery for those data can be done. (Either because it is the outermost recovery block used and the data are globals, or the data objects are local to that (outermost) recovery block.)

The term to cache is used to include the storing of any data to aid later recovery. These data used to aid later recovery form the cache. (Differences between a cache and a log are irrelevant for the purpose of this chapter.) A barrier is placed in the cache each time a new recovery block is entered. At the end of a recovery block (after the acceptance test has been evaluated) the information subsequent to the latest barrier will be processed and the barrier removed.

4.2 An example of a complex data structure: a filing system

A recoverable filing system has been implemented on a B1700 computer in the Computing Laboratory in Newcastle upon Tyne, in order to test the ideas and mechanisms described in this chapter. This filing system was part of the prototype two-level system described in chapter three. The filing system and basic principles on which the filing system and recovery mechanisms are based, are fully described in this section.

In order to implement the described recoverable filing system, the filing system of OS6 (StS72a,b,c,d) has been redesigned. The user interface has been kept unchanged. OS6 was chosen for several reasons. First of all the Computing Laboratory at the University of Newcastle upon Tyne possesses a version of OS6, which is running on a B1700 computer. Furthermore, OS6 is written in the high level language BCPL (Ric73) and is very modular: it is easy to replace parts of it. Finally the system is a single user system (more users exist but only one can be logged on at one time). This allows the problems of protection and recoverability in parallel systems (Ran75), (RLT77) to be avoided. This example of a recoverable filing will give the reader a better understanding of the notions and mechanisms used, such as cache, recovery and abstract data, and it is used in the rest of the chapter to illustrate the general problems and mechanisms discussed.

This section will not discuss the reasons behind the choice of the particular mechanisms chosen. The following sections will do this in a more general context, whereby the filing system described in this section is referred back to for illustration purposes.

4.2.1 The structure of the filing system

Files are regarded as sequences of objects of type word. User programs running under OS6 may create and destroy files. Files can also be freely assigned to variables, and be passed as parameters or be the result of function calls within a single program. A bottom-up description of a file is given in this subsection.

* The file body.

At the conceptually lowest level a file consists of a file body consisting of one or more directory pages and data pages.

A directory page is used to point to the data

pages. Both a directory page and a data page occupy one disk block each. When there are more data pages than entries in the directory page then a new level with a new directory page (the new top directory) is created which points to the directory pages which point to the data pages, etc. Examples of possible file body structures are shown in Fig.4.2, shaded areas indicate unused words.

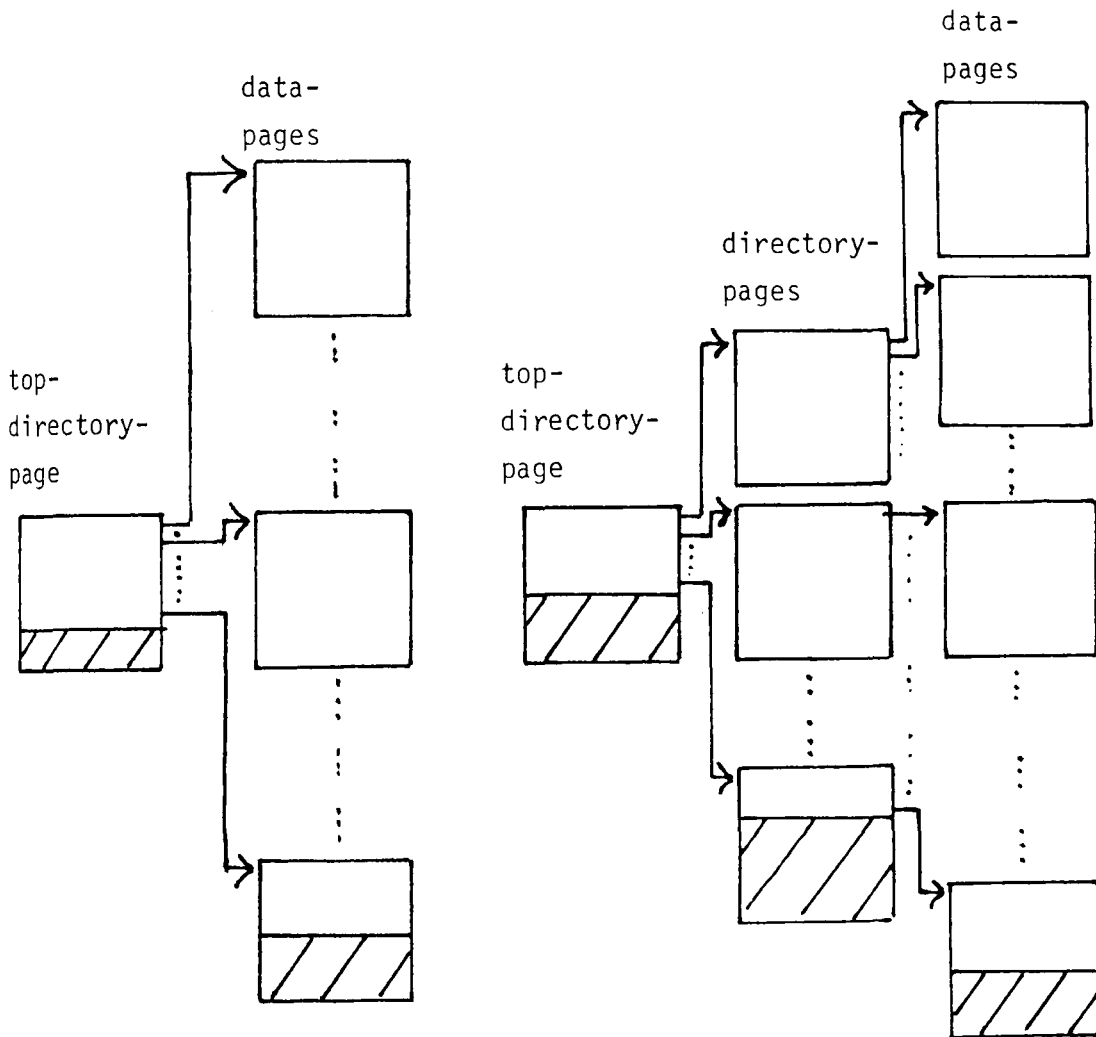


Fig.4.2 Two examples of possible file body structures

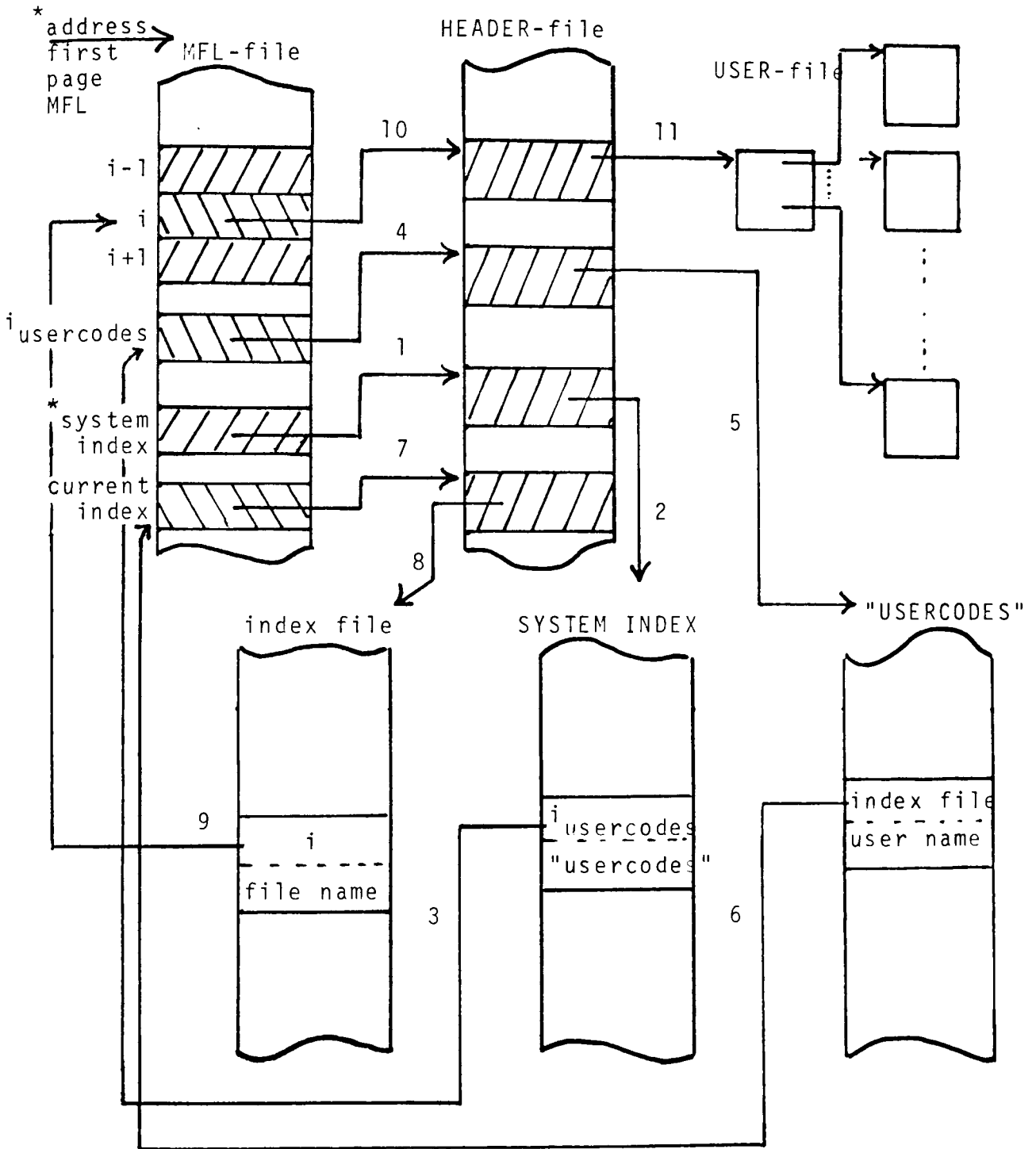
- * The header.
For each file there is a unique header which contains general information about the file, such as: the address of the top directory page, the address of the last page, the date-last-accessed, the date-last-written and the type of the file. These headers are kept in a special file: the Header file.

- * The file index.
Each file has a unique index. This index is the index in a table which contains the addresses of the headers for all the files in the system (an address is a tuple: page in Header file, offset). This table is kept in a special file: the Master File List (MFL). The disk address of the top-directory page of the MFL file is known by the filing system (a constant which is initialised when the system is set up).

- * The file name.
Index files are used to associate names with file indexes, in order to be able to use files in other programs. An index file contains entries: file index, file name. The System Index is the index file which contains all the entries for all the system files. The file index of this file (the system index) is known by the filing system (it is a constant which is initialised when the system is set up).

The data pages of the file body are the objects in which the user data is stored (the user "sees" file, but doesn't know about data pages). The directory pages, headers, file indices, and file names map higher level descriptions of a file onto these data pages, thus providing subsequent abstractions.

The MFL file, the Header file and the System Index are special system files. There are two other special system files. The first one is file "usercodes" which is an index file containing as entries the tuples: userindex, username. Each user has his own index file (a file directory) and when a user logs on, the system will look up his user index and set the system variable "current index" to the user index of that user. The other special system file is the Free Store File (FSF) which is used to contain the addresses of the free pages on disk. If a disk page is free then it is marked "free" and its address is in the FSF.



The two variables marked by a '*' are known to the system and set during system initialization. The value 'current index' is set when the user logs on, so after that the search for a user file starts in the MFL-file from 'current index'.
 Accesses 1-6 are made when a user logs on in order to set up the current index.
 Accesses 7-11 are made when the current user wishes to access one of his files.

Fig.4.3, The structure of the prototype filing system.

If a user wants to access a file with a given name the following steps are involved:

1. With the current index, the address of the header of the user index file is looked up in the MFL file.
2. The header is read and the first page of the user index file can be read.
3. The user index file is searched to find the index of the file with the given name.
4. With the index of the file the address of the header of that file can be found in the MFL file.
5. The header of the file can be read and used to access the file itself.

The total structure of the filing system is shown in Fig.4.3.

4.2.2 The mechanisms to provide recoverability for files

Files are regarded as globals for each program, even if they are created inside a recovery block. So only when the acceptance test of the outermost recovery block used is successful, final commitment of the filing system operations performed in the recovery block (which may contain nested recovery blocks) occurs. The updates for the filing system are made in such a way that the state of the filing system can be restored to what it was at a certain programmer-determined point (e.g. at the point of entering the current recovery block). The general principles upon which the filing system and recovery mechanisms are based are:

1. A minimum of information is to be kept in order to restore the state of the filing system (as seen by the user) to the state it was in at recovery block entry. The information must be sufficient to restore the state no matter which operations have been performed in the mean time. Since, for example, an audit trail scheme or "reverse audit trail" scheme keeps track of all of the operations performed and executes the reverse operations in the reversed order in which the original operations were performed, recovery may involve the accessing and restoring of previous values of one object, many times. The final value is then the value to which the object was to be restored. In the interest of storage usage recovery is to be linked with data structures and values rather than with operations,

and an audit trail scheme or reverse audit trail scheme is therefore unsuitable.

2. The choice of the level of abstraction at which cacheing is to be done, (i.e. the logical units for which recovery is provided) is of extreme importance for the efficiency with which the recovery is provided. For example for file bodies, disk pages are cached rather than disk words.
3. Updating of objects "in place" is avoided. The original values are left in the original objects and the new values are copied into new objects the first time these objects are assigned to inside a recovery block. A cache is used to maintain pointers to old and new copies of the objects. Crash resistance is provided, because new values are cached while original objects are left unchanged. Subsequent assignments are redirected to affect the new objects (cached values) rather than the original ones. Using this scheme, reads as well as writes trigger a search through the cache. The original cache scheme updates all objects "in place" and old values, as they were at the checkpoint (at the point of entering the current recovery block), of those objects are cached (Hor74). This scheme is appropriate for simple program variables, but not for complex global data types which will remain in existence after the running of the program, because it may leave the data in an inconsistent state if a failure occurs during the running of the program.

4.2.2.1 The mechanisms for updating and cacheing of files.

Updating and cacheing of files is done using a technique which is very similar to the "careful replacement" technique (see chapter two) which can be used to update files as safely as possible, i.e. minimising the chance of being left with an inconsistent filing system in the event of anything going wrong. Using this technique two versions of a file are kept when it is operated upon inside a recovery block. The versions overlap in sharing unchanged disk pages. A table is kept by the system to keep track of which pages belong to the new version only, and which pages they replace in the original version (if any; see below). This table is called the page cache. For reasons, which are described in subsequent sections, the MFL file and Header file are treated differently. Whenever an MFL entry is created, destroyed or updated an entry is placed in the MFL cache. Similarly a header cache is used to store entries when headers are created, destroyed or updated. These two "typed" caches contain all the cached information about

operations on MFL entries and headers. The pages of original versions of files, which are not shared with new versions, plus the three caches, form all the cached information in the filing system.

The way in which the first operation on each file is done after a (possibly nested) recovery block has been entered is described in detail below, in order to show how this cached information (recovery data) is formed and maintained to support the nested recovery. When such an operation is performed then the relevant data page to be updated will be copied into a new disk page and the change will be made in this new disk page. (This is for the case when a data page is updated; the file can also be extended by a data page in which case that new page is the newly created one. When a data page is deleted from a file, that data page will remain unaltered and not be copied into a new page.) If the directory page pointing to the original data page has not yet been changed inside the current recovery block alternative, then the contents of that directory page will be copied into another new page, and the pointer to the original data page is replaced by a pointer to the new data page. (This is for the case when a data page is updated; when a data page is added to the file a new entry will be placed into the new version of the directory, and when a data page is deleted from the file then the entry will be deleted from the new version of the directory page.) The same is done for possible higher level directories. Thus the new value of the file is defined by the new top-directory, and the original value, i.e. the cached value, is defined by the original top-directory. Thus inside a recovery block two forms of the file are kept from the point when a file is first changed inside that block. These two forms are the file as it was before entering the recovery block and the up-to-date version. When recovery blocks are nested then several forms of the file can be kept. In general: suppose that there are m nested recovery blocks and a file has already been changed in k of the m levels of nesting, then $(k+1)$ versions of the file are kept in the innermost recovery block.

In order to keep track of which page is replaced by which other page a tuple $\langle \text{old page } i, \text{ new page } j \rangle$ is put into the page cache for each replaced page. A tuple $\langle \text{old page } i, \text{ new page } j \rangle$ in the page cache will be combined with another such tuple after the latest barrier if possible. Pairs of tuples are combined according to rules T1-T4:

- T1) $\langle \text{page } 1, \text{ page } 2 \rangle$ & $\langle \text{page } 2, \text{ page } 3 \rangle \implies$
page 1, page 3, page 2 is freed.
- T2) $\langle \text{"new"}, \text{page } 1 \rangle$ & $\langle \text{page } 1, \text{ page } 2 \rangle \implies$
"new", page 2, page 1 is freed.
- T3) $\langle \text{page } 1, \text{ page } 2 \rangle$ & $\langle \text{page } 2, \text{"deleted"} \rangle \implies$
page 1, "deleted", page 2 is freed.

T4) <"new", page 1> & <page 1, "deleted"> ==>
page 1 is freed.

(Note: rules T1 and T2 will never be applicable here, because page 2 in T1 and page 1 in T2 will not be replaced, but overwritten. Those two rules will be used, however, for processing of the cache after an acceptance test.)

When the top-directory of a file is updated then the header has to be updated as well. The header of the file is then copied into a header cache, which in the prototype filing system is kept in core, and the new value of the address of the top-directory is placed in that header in the cache. A few other fields in the header may have to be updated too, such as "date last accessed". However, this is not of importance for the mechanism. Similarly, a tuple <file index, tag> is put into the MFL cache when a file is changed. The tag field can have the values "changed", "deleted" or "new", to indicate that the file has been changed, deleted or newly created respectively, within the current recovery block. A tuple <file index i, tag1> will be combined with a previously stored tuple <file index i, tag2> after the last barrier, if possible. Pairs of tuples are combined according to rules T5-T8:

T5) <file index i, "new"> & <file index i, "changed">
==> <file index i, "new">
T6) <file index i, "new"> & <file index i, "deleted">
==> nothing.
T7) <file index i, "changed"> & <file index i, "changed">
==> <file index i, "changed">
T8) <file index i, "changed"> & <file index i, "deleted">
==> <file index i, "deleted">

(A new tuple <file index i, "deleted"> causes a cached tuple <file index i, "new"> (i.e. cached in the same recovery block level) to be erased from the cache and the "deleted" tuple will be discarded. This is done, because all it means is that a file has been created and subsequently destroyed inside the same recovery block alternative. Consequently no further processing in order to undo or accept these operations will be necessary after the acceptance test.)

Subsequent changes in the same disk page or file header, made within the same recovery block level, are done in the newest version of the page or header without any further cacheing. As far as the MFL cache is concerned, only a deletion of a file can cause an already cached tuple to be overwritten; other subsequent changes in the file in the same recovery block level do not affect the MFL cache. The same also holds for the page cache.

Each time a page, header or MFL entry is changed, an entry is sent to the respective caches. The caches with

their routines decide what to do with the received entry. Original pages, headers and MFL entries do not carry any indication to show whether or not there is an updated version of them (which is in the cache for headers and MFL entries). Therefore, the definition of access paths has to be changed to include the caches. In other words, the search order for an MFL entry or header is changed to start the search in the caches. The fact that three separate caches are used instead of one cache is irrelevant. Because there are logically three kinds of cache entries it is convenient to speak of three caches. However, these three caches could well be implemented as one cache in which each entry has a type descriptor to distinguish its kind.

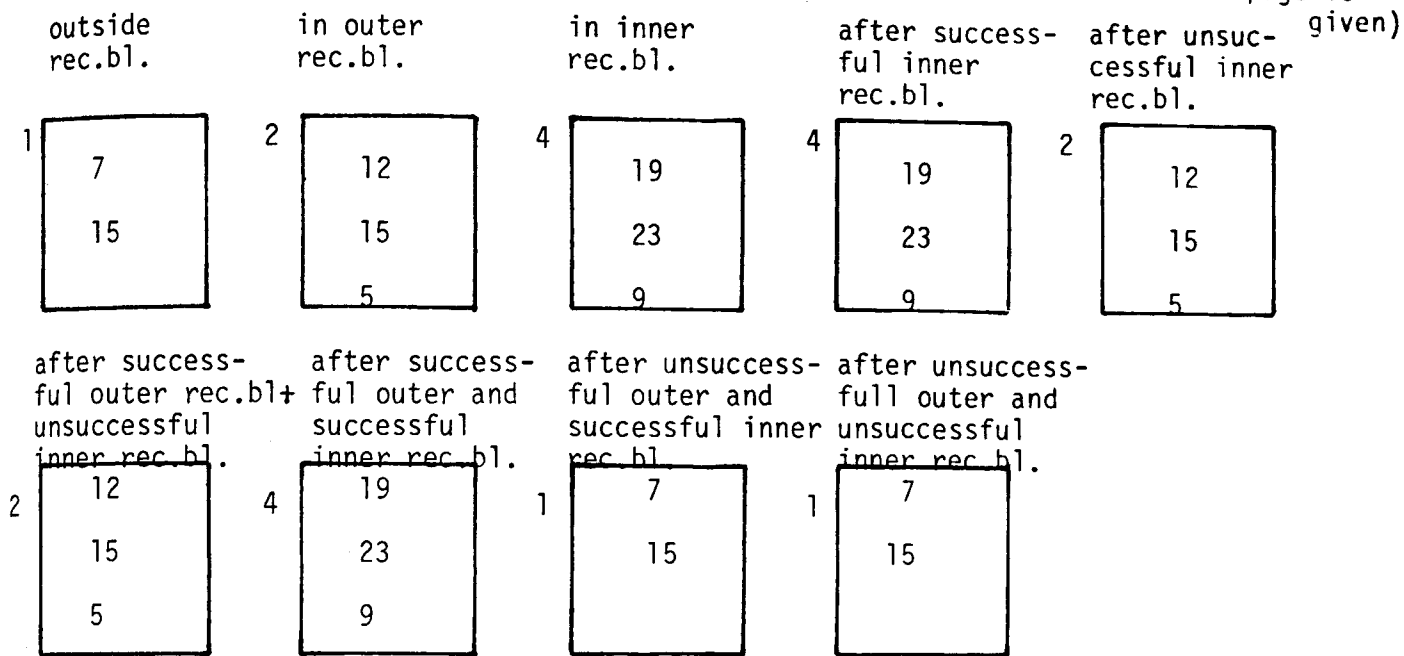
The given design and definition of files is well structured in the sense that many versions of a file can be kept without any unnecessary duplication of information. Each version has the same structure and is defined by its index (and header). No complicated merging (as with differential files; see chapter two) after acceptance of the operations, nor restoring of individual disk pages (as an audit trail would require) after a failure, are needed. Another advantage of the technique used is that by cacheing new values rather than old values of MFL entries and headers, the original files are kept on disk and will remain undamaged in case of a system crash.

4.2.2.2 Processing of caches and files after the acceptance test

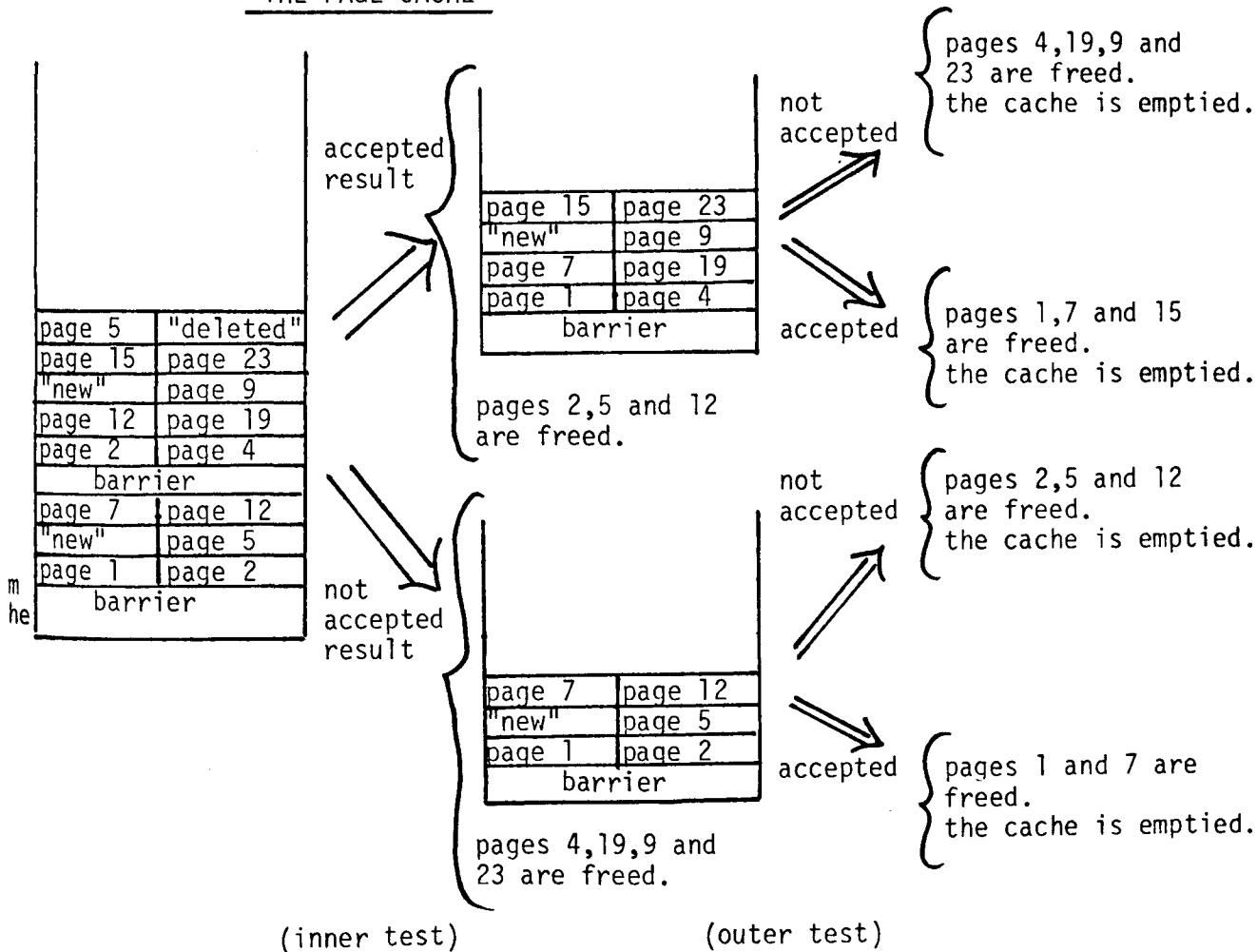
At the end of a recovery block alternative the page cache is used to update the FSF (Free Store File) and to free pages. It is important to note that changes in FSF are not cached. The algorithms which process the page cache, header cache and MFL cache after a recovery block acceptance test are described here and an example of the page cache processing is shown in Fig.4.4.

In the case that the acceptance test of a recovery block has failed, all the "new pages" in the tuples <old page, new page> in the page cache up to the latest barrier in the cache, are freed. The cache is cleared up to the barrier. If the acceptance test was successful then there are two possible cases. The first possible case is that the outermost recovery block has been successful, in which case all the "old pages" in the tuples <old page, new page> in the cache are freed, and the whole cache is emptied. The other possible case is that an inner recovery block acceptance test has been successful, in which case the latest barrier is removed and all the tuples are moved up. Pairs of tuples up to the next barrier are combined

THE TOP-DIRECTORY PAGE AT DIFFERENT STAGES (the address of the current page is given)



THE PAGE CACHE



Assumed is that no caching is done between the execution of the inner and outer test.

Fig.4.4, Page cache processing: an example with a file with one directory page.

according to rules T1 to T4.

When an inner recovery block acceptance test is successful then the headers in the header cache after the latest barrier are merged with the headers between the latest barrier and the next latest barrier. If the same header appears both before and after the latest barrier, but after the next latest barrier, the earlier header in the cache is replaced by the most recent one. The latest barrier is then removed and other headers move up to fill up empty spaces in the cache. If an inner recovery block acceptance test is not successful then the cache is emptied upto and including the barrier. The MFL cache is processed like the header cache. Pairs of tuples are combined according to rules T5 to T8 in case an acceptance test has been successful. When an acceptance test of an outermost recovery block in a program is successful then the caches are used to update the MFL file and Header file.

4.3 Cacheing for complex data structures

A generalization of the approach that is exemplified in the filing system is given in this section. The problems and constraints in designing a cacheing scheme for recoverable complex data structures, and possible solutions for those problems, are discussed.

The representation of an abstract object is a data structure which can logically always be distinguished into data providing an abstraction and information carrying data. Information carrying data is used to store the user's data (values) whenever he makes assignments to the abstract object. These data are addressable from the higher level through the system (a level providing the complex data structures).

In order to provide a concrete representation of the abstract structures, the relations between the components of the structure must be represented in some way, and the abstract operations interpreted in terms of these relations. The extra data used to describe the representation will be called "data providing an abstraction". In other words, this data is additionally required to support the abstract view of the data structure being provided. In effect, it is part of the mechanism structure rather than the data structure.

For example in the prototype filing system described in the previous section the directories, MFL entries and headers are data that is used to describe the relationships between the data pages and the file index. Directories, MFL entries and headers are therefore referred to as "data providing an abstraction", which in this case provide an

abstraction that is called "file".

The general logical structure of a complex data structure is shown in Fig.4.5.

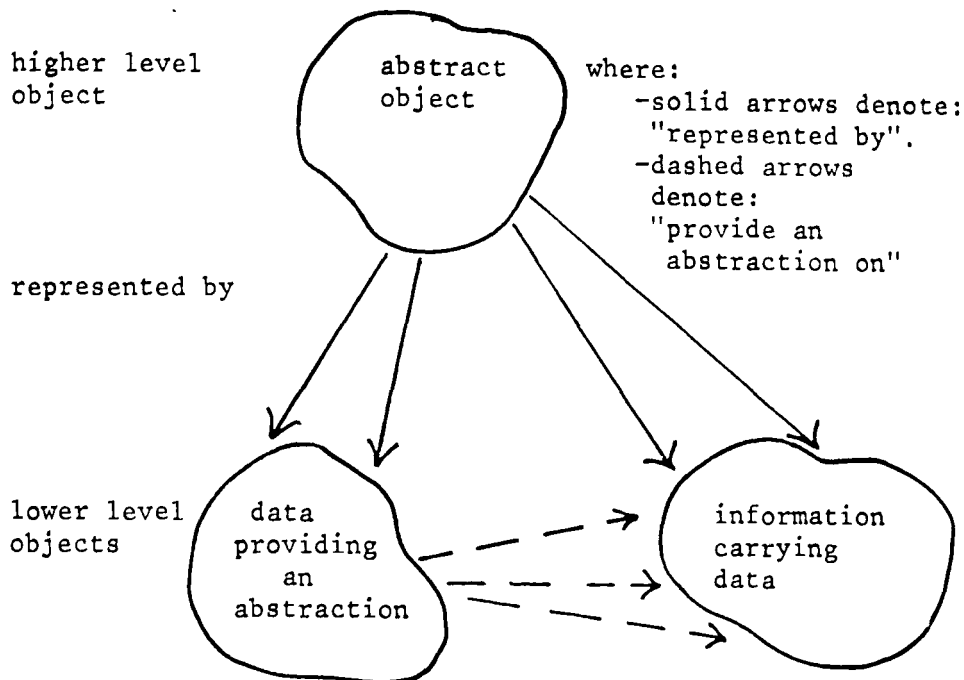


Fig.4.5, The logical structure of a complex data structure.

The provision of recoverability for a type represented by a complex data structure as shown in Fig.4.5, involves:

1. When an object of such a type is updated then the information carrying objects have to be updated and cached.
2. Depending on the way in which updating and cacheing is done, the data providing an abstraction may have to be updated.

4.3.1 Separating data providing an abstraction and information carrying data

The approach which is nowadays becoming more important as data base complexity grows, is to store the data providing an abstraction (data that define the "relationships" as defined by Martin (Mar75)) separately from the information carrying data. The major objectives of

this approach are to make possible faster data retrieval and to provide more complete data independence (Mar75).

It will be shown that not separating data providing an abstraction and information carrying data in any data structure may make recoverability prohibitively expensive. This can be illustrated by an example:

Consider the example of the filing system described in the previous section. Suppose that instead of using directory pages to record the relationships between data pages, pointers in the data pages themselves are used to link the pages. If a data page is updated then this means, under the scheme used, that the given page is replaced by another page. Consequently the link in the page pointing to the given page must be changed to point to the new page. The page containing the link must then be replaced as well, requiring an update in the link of the page pointing to it, and so on. Thus a whole chain of pages will have to be updated and cached. This chaining of pages is therefore obviously totally impracticable if recoverability is to be provided on a disk page basis as described. There are of course ways to get around this such as only replacing a page if the data part were changed and cacheing the changes to links in a special link cache. Although it is difficult to say anything about efficiency, it is clear that the definition of a file becomes very messy. It will be shown that also safely updating the file once the operations performed are accepted becomes more difficult. For those reasons these kinds of schemes are rejected.

In general it can be said that, whatever the structure of the complex data structure is, for example a chain, ring, tree, plex (Mar75) or any other, if the information carrying data is not separated, both physically and logically, from the data providing an abstraction then the problems of updating and providing recoverability may become prohibitively expensive or impracticable as data structure complexity grows.

Redundancy, such as redundant pointers to make linear searching a little easier, or sumchecks, are part of the data providing an abstraction, i.e. part of the mechanisms, unless this redundancy is used as "hints" (Lam75). Hints are, as described in the first chapter, always checked against some "absolutes" and are used to optimise the efficiency of operations on the data structures. It is, therefore, unnecessary to maintain correctness of hints all the time. Absolutes are data providing an abstraction, which therefore must be always correct. If redundant information items are not used as hints then they are part of the data providing an abstraction. Consequently, it is important to be careful with the choice and use of redundancy in data structures.

4.3.2 The cacheing problems for complex data structures

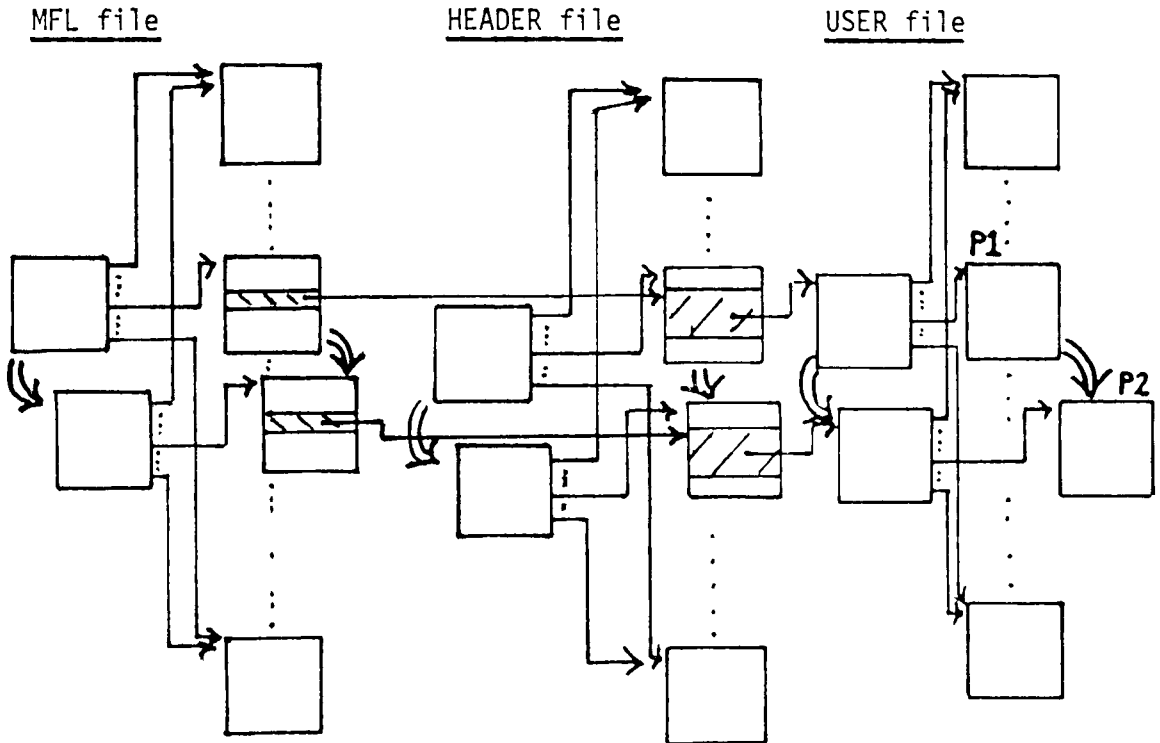
In general, if a level (in a multi-level system) provides a type T, then it may use some specially reserved objects of type T to store data necessary to support the abstraction being provided (i.e. to store data that is part of the mechanism used to operate on the complex data structures). So these special system objects contain data providing an abstraction. For example, the MFL file and Header file in the filing system of the previous section.

If these special objects of type T are treated as ordinary data structures, like all other objects of type T, as far as updating and cacheing are concerned, then this may cause the same sort of linked update problems as described in the previous subsection. The problems now occur at a higher level and at a larger scale:

Suppose that a data structure D1 is to be updated, and this update has to be cached. As a result of this the data providing an abstraction may also have to be updated. Again this update may have to be cached. Suppose that the data supporting an abstraction is stored in data structures which have the same type as data structure D1. Also suppose that the data supporting the abstraction of the updated data structure D1 is stored in data structure D2. Consequently data structure D2 may have to be updated. As a result of this the data providing the abstraction for this particular data structure D2 may have to be updated and cached as well. So the data supporting the abstraction of data structure D2 may have to be updated, and so on.

A particular example of this problem is (see Fig.4.6): Consider the filing system described in the previous section. A change made in a file within a recovery block causes its top-directory to be changed. Consequently the header has to be changed. If the header file is to be treated as an ordinary file then this means that the disk page in which the header to be updated is stored (in the Header file) will have to be replaced. As a result of this the addresses of the updated header and all the other headers in the replaced disk page of the header file change. The header of the header file also changes as a result of this, and consequently several other headers may get a new address. This means that for all these headers the corresponding MFL entries will have to be changed. If the MFL file is treated as an ordinary file then this implies that all pages in MFL, in which an entry has to be changed, have to be replaced. Obviously the header of the MFL file also has to be changed. Again the Header file must be changed. Thus each first change on the filing system within a recovery block may cause a large part of the filing system to be replaced (cached).

Summarising it can be said that data providing an abstraction may use the same data structure as those of which an abstraction is provided, but recovery for them must be provided separately.



Where: a double arrow is used to indicate "replace for update".

The user updates page P1, which causes also the data page of the Header file containing the header of the user file to be updated. This causes the data page of the MFL file that contains the MFL-entry pointing to that header to be updated as well. All the relevant directory pages are also updated.

Other headers and MFL-entries are updated as well (see text), causing more page replacements. However, these updates are not shown in this diagram.

Fig.4.6, The explosion of updates that may be caused by a first update of a file inside a recovery block when careful replacement is used and the Header file and MFL file are treated as ordinary files.

4.3.2.1 A general solution and two specific implementations

The distinction between information carrying data and data providing an abstraction has already been stressed, and will be necessary in this subsection for a solution to the caching problem described above. The general solution to this caching problem is as follows:

The caching for the information carrying data is done by storing the original values of the objects in which this data is stored. When the value of an object used to keep information carrying data is changed then the data providing an abstraction for that object may have to change, because of the way in which updating and recoverability are implemented. The caching for data providing an abstraction has to be done differently from caching for information carrying data. Data providing an abstraction consists of logical units, for example, headers, descriptor, addresses, mapping tables or pointers. Rather than providing recoverability for the objects in which the data providing an abstraction is stored, recoverability will be provided for the logically independent units.

For example in the filing system of the previous section, data pages are the data objects used as information carrying data. Data pages are cached when updated. The data providing an abstraction consists of the headers and MFL entries. This data providing an abstraction is stored in two files, but recovery is provided for headers and MFL entries, as units of recovery rather than data blocks as for ordinary files.

According to the definitions given in this chapter the directories, used in the filing system described in section 4.2, are data providing an abstraction. Yet in the implementation recoverability for directory pages is provided as for information carrying data, i.e. for each object used to contain the data (the disk page). In the implementation of the previous section this was possible for two reasons:

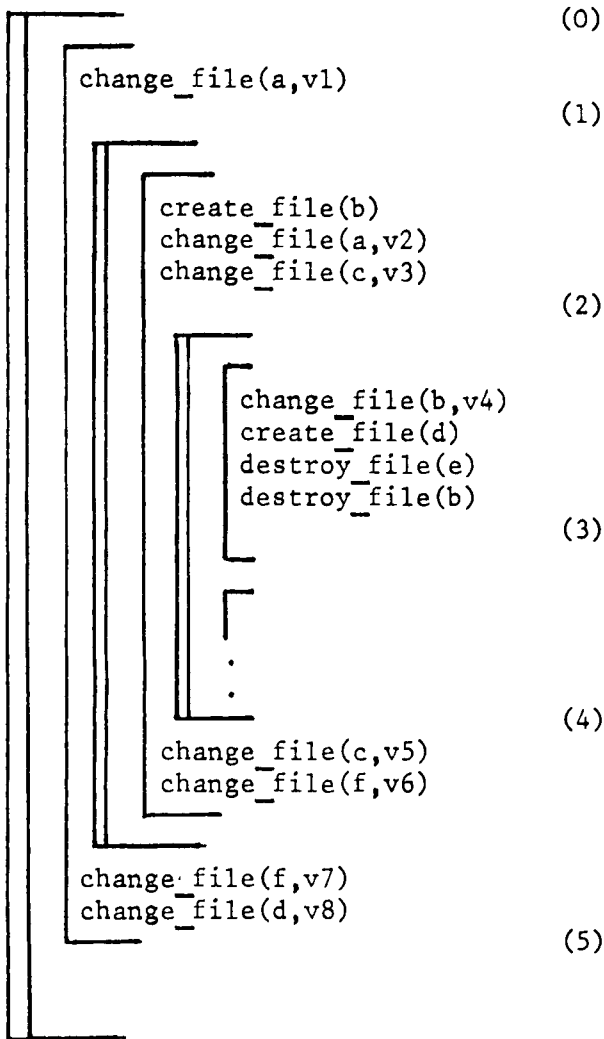
1. One data or directory page always completely belongs to one file. It is not possible that half the page is used to store data of one file and the other half used to store data of another file. In other words these data and directory pages form the bricks out of which data structures called file bodies, are constructed. Obviously a directory cache could be implemented to contain directory pointers, like the MFL cache and header cache are used to contain MFL entries and headers respectively. Using such a directory cache may even be more efficient under certain circumstances, although it is impossible to make general statements

about this.


2. Directory pages are part of the data structure, called file body, and not data providing an abstraction that are stored in another data structure of the same type as the one it is providing an abstraction of (like headers and MFL entries are stored in files). A distinction between these two kinds of data providing an abstraction could be made. Directory pages could be said to be used to define the "relationships" (as defined by Martin (Mar75)) of data objects in the data structure. Headers and MFL entries provide an abstraction of these data structures. However, it is not felt to be necessary for the purpose of this thesis to make this kind of distinction.


The main point is that from some level of abstraction onwards, in a complex data structure, recoverability for data providing the abstraction must be provided for each type rather than for each physical object used. This general solution breaks the vicious circle described above, because objects containing data providing an abstraction are treated as special objects for which the provision of recoverability is independent of the type of data objects used.

Two different ways in which this scheme can be implemented are considered. The two methods look very similar, but are in fact fundamentally different. The first method (from now on referred to as method I) is to change the data providing an abstraction when necessary and cache the original values of the changed units. The second method (from now on referred to as method II) is to leave the original data providing an abstraction unaltered and cache the new values of the units. This is the same distinction as discussed earlier (in section two) with respect to update "in place" and cacheing original values as opposed to cacheing of new values. (See Fig.4.7, Fig.4.8, Fig.4.9.) In order to show the differences, suppose that an update is made to an object causing the data providing an abstraction to be updated too. Also suppose that the object is a global (either because it is declared outside the outermost recovery block, or because it is a type which is always global even if it is created inside an inner block, for example a file on disk or some other resource). If the update is made inside a nested recovery block then method I will work according to the following principles (Fig.4.7 shows an example program and Fig.4.8 and Fig.4.9 show how methods I and II, respectively, update objects and cache values.):



where: change_file(f1,v1) : gives file f1 value v1
destroy_file(f2) : destroys file f2
create_file(f3) : creates file f3

 is used to mark the scope of a recovery block

 is used to mark the scope of an alternative

All files are global objects, even when they are created inside a recovery block.

Fig.4.7, An example program operating on global objects.

- * The data providing an abstraction will be updated and the previous value will be cached with a tag to indicate the kind of update that has been done, e.g. change, delete or create.
- * Up-to-date versions of the data providing an abstraction are now kept while previous values are cached.
- * The entries in the cache can only be thrown away when the outermost recovery block acceptance test has been successful, because the objects are globals to recovery blocks.
- * If there are several abstractions each providing a more abstract view, thus implementing a complex data structure, then a different cache will be associated with each set of data providing an abstraction. This may not be necessary in the case that one level implements them, but it is irrelevant in that case whether there is one cache with type-flags for each entry or different caches. However, it is convenient to regard a separate cache as being associated with each set of data providing an abstraction. The processing of these caches can now be done completely independently. No information from other caches will be needed to process a cache in order to accept or undo the operations performed on a set of data providing an abstraction.

Method II will work according to the following principles:

- * The objects in which data providing an abstraction are stored will remain unaltered and new values will be stored in a cache. The definition of the data providing an abstraction is completely changed. The data providing an abstraction now consists of the caches and the original data providing an abstraction. So the mapping functions are different.
- * If several sets of data providing an abstraction are used then there may be a problem in constructing new values of data providing an abstraction if an abstract data type is operated upon inside a recovery block. This problem can be shown in an example:

In the filing system described in the previous section, the MFL file and Header file are not updated immediately when a file is operated upon inside a recovery block. Instead the new value of the header is cached and an entry to indicate that the file has been operated upon is put in the MFL cache. An MFL entry (in the MFL file) contains a pointer to the corresponding header in the Header file. However, if a new header is cached then the MFL entry can not be constructed yet.

place	cache	values of files					
		a	b	c	d	e	f
0	empty	va	NE	vc	NE	ve	vf
1	a=va	v1	NE	vc	NE	ve	vf
2	a=va ----- b=new a=v1 c=vc	v2	empty	v3	NE	ve	vf
3	a=va ----- b=new a=v1 c=vc ----- b=empty d=new e=ve	v2	NE	v3	empty	NE	vf
4	a=va ----- a=v1 c=vc d=new e=ve	v2	NE	v3	empty	NE	vf
5	a=va ----- a=v1 c=vc d=new e=vew f=vf	v2	NE	v5	v8	NE	v7
6	a=va c=vc d=new e=ve f=vf	v2	NE	v5	v8	NE	v7
7	empty	v2	NE	v5	v8	NE	v7

Where: NE means non-existent.

Fig.4.8, The cache contents and values of files at various places in the program of Fig.4.7, while cacheing method I is used and no errors occur.

Instead an MFL entry is made to indicate that a new file has been created.

- * The entries in the caches can only be processed (i.e. put in the sets of data providing an abstraction in order to update the original sets) when the outermost recovery block acceptance test has been successful.
- * As a consequence of these last two points the caches associated with the different sets of data providing an abstraction cannot be processed independently. When the outermost recovery block acceptance test has been successful, the data providing the lowest level abstraction has to be updated first. Then the data providing the next level abstraction can be updated, and so on. So some communication or simultaneous processing will be necessary between the procedures that process these caches.

The last point is a characteristic difference between method I, i.e. doing the actions immediately and caching the previous states, and method II, i.e. delaying the actions and recording the new states.

4.3.2.2 A comparison of the two methods

Both methods described above have their advantages. Neither of the methods can be said to be superior under most circumstances.

Method I has the advantage that caches can be processed independently. Another advantage of method I is that the definitions of data providing an abstraction are the same for an unrecoverable system and for the recoverable system with the same structure.

Method II has the advantage that the original values of data providing an abstraction are kept so they remain unchanged in the event of a system crash (unless the crash occurs during the cache processing after a successful outermost recovery block acceptance test). This is of importance if data has a longer lifetime than the runtime of the program, and is kept outside core. Another advantage of method II arises when more than one user exists and there is an access strategy providing multiple reads but exclusive updates for the data structures. While an update proceeds, and until it satisfies its acceptance test, other users can use the previous version. The update program must exclude other readers only during the actual cache processing and updating of the original version when the outermost recovery

place	cache	values of files					
		a	b	c	d	e	f
0	empty	va	NE	vc	NE	ve	vf
1	a=v1	va	NE	vc	NE	ve	vf
2	a=v1 ----- b=new a=v2 c=v3	va	NE	vc	NE	ve	vf
3	a=v1 ----- b=new a=v2 c=v3 ----- c=destr. d=new e=destr.	va	NE	vc	NE	ve	vf
4	a=v1 ----- a=v2 c=v3 d=new e=destr.	va	NE	vc	NE	ve	vf
5	a=v1 ----- a=v2 c=v5 d=v8 e=destr. f=v7	va	NE	vc	NE	ve	vf
6	a=v1 ----- a=v2 c=v5 d=v8 e=destr. f=v7	va	NE	vc	NE	ve	vf
7	empty	v2	NE	v5	v8	NE	v7

Where: NE means non-existent.

Fig.4.9, The cache contents and values of files at various places in the program of Fig.4.7, while cacheing method II is used and no errors occur.

block acceptance test has been successful. If the test fails other readers are completely unaffected. Even if the test succeeds other readers could continue using the old version. This involves constructing new mapping tables for new files (new file directories), letting other readers still use old mapping tables. However, these issues are outside the scope of this thesis.

Summarizing:

- * If an existing system is to be made recoverable then method I is probably the most practicable one, because mapping functions do not change. In general, however, making an existing system recoverable will almost certainly give many problems.
- * It is expected that method II is more feasible in a multi-user system with multiple reads but exclusive update access strategy for the data structures.
- * With method II the caches associated with the different sets of data providing an abstraction cannot be processed independently, while with method I they can. This may complicate the cache processing with method II.
- * Method II may have advantages in the event of a system crash and has been chosen for this reason in the filing system described in section 4.2.

4.3.3 Alternative solutions to the cacheing problem

It could be argued that cacheing file words rather than whole blocks would solve the problems described in this section. However, there are several other problems with this scheme.

In the first place, if method II is used for cacheing then a linear scan through a file could be very complicated and inefficient, because a stream of words rather than disk pages is to be constructed from the original version of the file and the cache. The cache will have to be implemented on disk, because core store may safely be assumed to be too small to contain such a cache. Consequently the method of cacheing will have to be method I, which may be a disadvantage in itself.

Secondly, if method I is used for cacheing then the undoing after an unsuccessful acceptance test will be very cumbersome and inefficient. However, this could be acceptable since acceptance tests are not expected to fail frequently.

Finally, the overhead in administration data in the cache will be fairly big e.g. a page address plus offset for each cached word. This would mean, for example, that only 33% of the cache would consist of cached words if the page address and offset occupy a word each. The overhead can under certain circumstances be justified such as in the special representations used for sparse matrices (i.e. if on average only a few words per data page are updated). Cacheing will in any case be complicated and probably inefficient, because writing one cache entry to disk every time a file word is changed would obviously be too big an overhead. Consequently this will have to be optimised, which may be complicated, because cache entries must be written before the file is updated, in order to be able to cope with errors occuring between these events, or system crashes.

The scheme looks in fact rather like one where bits are cached to provide recoverability for integers. However, it is not that this particular scheme is in all environments in all cases a bad scheme, but rather that the problems discussed in this section are real problems and the solutions sought are feasible ones.

4.3.4 Main conclusions

The main conclusion drawn from the discussion on recoverability for complex data structures is that data providing an abstraction and information carrying data have to be separated in different physical objects whenever possible. They also have to be treated separately when recoverability is to be provided for them.

The information carrying data is preferably treated as physical objects for which recoverability is provided, while for data providing an abstraction the recoverability is preferably provided explicitly for the logical units (types) from which the data is composed.

In the filing system of section 4.2 the types that could be distinguished at different levels of abstractions were: word, page, directory pointer, header and MFL entry. It was shown that recovery is to be provided for types rather than for the physical objects used to store the data, from some level onwards. Below that level recovery is to be provided for those physical objects used. In the filing system the recovery for types was provided for headers and MFL entries: it could have been provided for directory pointers and even words in data pages as well. However, that would have been less efficient in the particular environment.

4.4 Maintaining consistency in recoverable complex data structures

So far facilities to make state restoration possible have been discussed. Another aspect of recovery is crash resistance, which is, as described in chapter two, of importance for data structures that remain in existence after the running of the program has finished. It will be shown that the same mechanisms that have been discussed in sections 4.2 and 4.3 for providing recovery, can be used to provide crash resistance.

The notion of recovery block provides a convenient way of talking about the problems and will therefore be used in relation to the techniques and mechanisms used for recoverability and crash resistant systems. Recovery blocks may be used to define the scope of recovery from crashes. The scope of recovery from crashes is taken to be the time between entering and leaving the outermost recovery block. (Only recovery for sequential processes is considered here.) The use of a recovery block with a trivial acceptance test and one alternative must assure the writer of the program that unless a catastrophe occurs, so that data are lost, everything he wants to do will be done or nothing will be done at all. The kind of consistency considered is explained by an example:

If a program inside a recovery block wants to update a file header on disk (the header is a variable length block which may be divided over more than one disk page), then that header must either be updated completely, or must not have been modified at all. Thus suppose that a header is partially in disk page x and partially in disk page y . Then a power failure could occur when x has been updated and y has not been updated yet. This would leave the header neither in its original nor in the new state and, therefore, must be impossible.

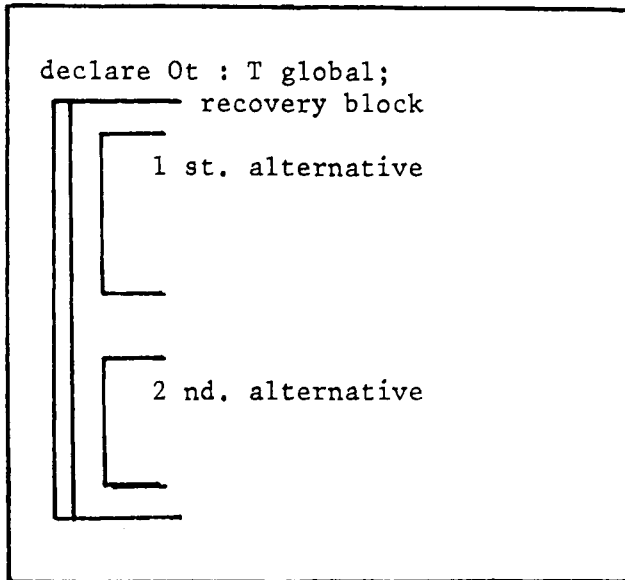
4.4.1 Crash and crash resistance

The notions of crash and crash resistance have been defined in chapter two of this thesis. These notions, however, will be defined more precisely here for the purpose of this chapter.

Program P , shown in Fig.4.10, may have to be abandoned, because a level $j \leq i$ fails and cannot continue. Level i performs operations on objects obj_1, \dots, obj_n whenever it interprets an operation on O_t in the user program P . If level i has to abandon program P (or the interpretation of

an operation on O_t), because a level j ($j \leq i$) has failed, then this will be called a crash. It is assumed that object O_t does not disappear with program P , but rather is a global which can be used again by a following program after P has finished.

Program P



level $i+1$

A level mapping O_t onto obj_1, \dots, obj_n
of the underlying machine.

level i

underlying machine

Fig.4.10, A diagram of a level on which a program using an abstract type T provided by the level, is running.

In order to show which sorts of crashes the system is required to be able to cope with, all crashes are classified in either one of the following classes:

1. None of the objects obj_1, \dots, obj_n is corrupted, because of the crash.

2. One of the objects obj_1, \dots, obj_n is corrupted, because the crash occurred while a correct state transition for O_i was in progress. Thus that object is left in an invalid state.
3. k of the n objects obj_1, \dots, obj_n are corrupted, with $2 \leq k \leq n$, because of some catastrophe.

For example, in the filing system described earlier in this chapter a crash of class 1 occurs if the system crashes while no disk write was in progress, or the disk write is not affected once initiated successfully. A crash of class 2 occurs if a disk write is interrupted or unsuccessful. A crash of class 3 occurs if a head crash occurs, or the operating system fails to perform normally and goes on writing to disk for a while, or the disk head does not position itself correctly before a disk write.

A class 3 crash may corrupt the whole data base such that repairing is impossible and built-in defense mechanisms are destroyed or have not worked properly. So, as described in chapter two, other recovery mechanisms are required to cope with such failures. Recovery from a crash of class 3 is possible is by keeping complete backup versions of all the objects obj_1, \dots, obj_n on different physical storage devices. (However, other failures may of course corrupt these backup versions.) From now on the term "crash" is used in this section to mean just one of the first two sorts of crash, unless indicated differently. A class 3 crash is called a "catastrophe". Crashes of class 2 are distinguished separately, because in many systems there are many kinds of failures that may cause the (so far correct) execution of a program to be stopped abruptly, even when an update of an object obj_i is in progress. It is therefore useful to be able to cope with this particular class of crashes.

The consistency requirement described means that after a crash occurs while the system is in a recovery block, the system as seen by the user must either be in the state it was in before the recovery block was entered, or the new state which is the one it would be in if the recovery block had been left normally after a successful acceptance test. The only concession that will be made is that some objects on which the user objects are mapped, may have to be "freed" by some sort of garbage collector, for reasons which will become clear in the following subsection. If the system can cope with crashes of class 1 and class 2 (in the way described here) without requiring special recovery actions after the crash, then it is called crash resistant.

4.4.2 The provision of crash resistance with recovery

There are several ways in which crash resistance can be provided. Chapter two showed that the careful replacement technique is one of the best techniques to provide crash resistance. There will however, using careful replacement, always be some information which has to be kept in a fixed place, because the software must have a "grip" on the data structures (e.g.: the address of the first page of a system directory in a filing system). This information will therefore have to be updated "in place".

There are of course other strategies that could be used. A complete survey of techniques has been given in chapter two. However, the careful replacement technique is one of the most efficient ones and fits in nicely with the recovery strategy discussed in sections 4.2 and 4.3.

It was argued that updates to an object O_t have to be done as follows:

1. Cache the information carrying objects (see Fig.4.5) which have to be changed and make the necessary updates.
2. Cache new values for data providing an abstraction (see Fig.4.5) and process the cache and the data providing an abstraction after passing the recovery block acceptance test.

Step 1 can be implemented by the careful replacement scheme. The data providing an abstraction of step 2 then comprise the information that has to be updated "in place". Thus the careful replacement technique for providing crash resistance fits in nicely with the recovery strategy discussed. The original versions of objects that are replaced (i.e. updated) are now the cached objects. Consequently they are not thrown away after the operation on O_t inside a recovery block has been completed, but are kept until the appropriate recovery block acceptance test has been successful. If this acceptance test fails then the new versions of the objects have to be thrown away.

For example, in the filing system described in this chapter, file bodies, i.e. directory and data pages, are updated using the careful replacement technique. New values of data providing an abstraction, i.e. headers and MFL entries, are kept in caches. Consequently whenever a user operation on a file is performed inside a recovery block and a crash occurs, the file will keep its original value. The critical moment is when the data providing an abstraction have to be updated "in place" after a successful acceptance test.

4.4.2.1 The critical updates

The strategy followed leads to the following situations in the event of a crash occurring at different stages of the processing of user object Ot, using recovery blocks:

- * If a crash occurs inside a recovery block before either an "UNDO" or an "ACCEPT" has been executed for that block then the user object Ot (see Fig.4.10) will not yet have been changed, because the data providing an abstraction still map onto the original version. (The "ACCEPT" is the accept procedure after a successful acceptance test. The "UNDO" is the backup procedure after an unsuccessful acceptance test.) Only new objects obji (see Fig.4.10) will possibly have been allocated by the level providing object Ot.
- * The "UNDO" frees newly allocated objects obji and undoes the caching of new values of data providing an abstraction. "UNDO" does not affect the objects obji of the original representation of Ot. Thus if a crash occurs during the execution of the "UNDO" procedure then Ot will remain unaltered.
- * The "ACCEPT" for an inner recovery block processes caches only and possibly frees some objects obji that were newly allocated (for a detailed example see the filing system described earlier). So the same holds for such "ACCEPT" procedure as for the "UNDO".
- * The "ACCEPT" procedure of the outermost recovery block does update the data providing an abstraction "in place". Other objects obji remain unchanged during the "ACCEPT", because there is already an old and a new version for every changed object. So only some objects obji need to be freed. If the crash occurs during the updating of the data providing an abstraction then this could leave an inconsistent data structure behind (i.e. the user object is in an invalid state).

Consequently the problem is now isolated to the "ACCEPT" of the outermost recovery block, during which information is updated "in place". This updating is called the critical update.

Several approaches to this problem are possible, some of these are:

1. The principle of multiple copies could be used. This concept provides absolute crash resistance at any time.
2. By using extra levels of indirection the amount of information to be updated "in place" could always be reduced to one object (see also (New72)). For example, rather than updating the whole system catalogue "in place", a new version of the catalogue could be made and a fixed disk page could be used to contain the address of the first page of the catalogue, thus reducing the amount of information to be updated "in place" to one word. The updating of the one object "in place" could be done using one of the other approaches described.
3. The probability of a crash occurring during the updating of the data providing an abstraction in "ACCEPT" can be so small that we can afford to update them "in place" and still be very safe. If a crash occurs anyway then it is to be treated as a class 3 crash.
4. If there is only one object to be updated "in place" (or just a few) then an external device (or the operator) could be used as a backup of the value of this object (for example write it to a mini-tape or type it on the operator's console and let the operator type it in again in case a crash occurred before the "OK-message" is printed). In fact this could be regarded as an implementation of the multiple copies technique.

Many other schemes, possibly involving audit trail, differential files, checkpoint/restart for the whole process, or other techniques, could be devised. However, the techniques described here are the most commonly used ones and seem the most obvious ones.

For example, in the filing system described in this chapter, strategy 3 has been chosen, because 1 and 2 are both too cumbersome, and critical updates are not likely to happen very often (only during the "ACCEPT" of an outermost recovery block).

The advantage of strategies 1,4 and 2 (if method 1 or 4 is used for the single object to be updated "in place") is that after a crash that occurs within any recovery block (even the outermost) the system can usually just restart as if nothing has happened. With strategies 3 and 2 (if method 3 is used for the single object to be updated "in place"), however, some way of finding out whether the crash occurred during an update of data providing an abstraction "in place" or not, is needed. If it did then the crash is to be treated as a class 3 crash, if it did not then all is still well. Two ways in which the system can find out when the

crash occurred are using a salvation program and using a flag to indicate "critical update in progress". A condition for the use of a salvation program is that it must be possible to validate the data structures (and data providing an abstraction). The same program could be used for other purposes, for example if after a hardware failure it has to be established whether a crash was a class three crash or not. An implementation of a system with such a salvation program (called a "verification procedure") is described by Fraser (Fra69). A vector of the length of the number of available disk blocks is kept. The directory of a file points to an entry in this vector, which indicates the first block of the file. Each entry points to another entry which indicates the next block of the file. The validation program checks the length of the file, which is kept in the directory, with the number of disk blocks belonging to the file. If there is any inconsistency then the file involved is deleted. Many systems also use a flag "update in progress" or, as it is sometimes called, a "damage flag" (Cur77).

4.4.3 Conclusions

This section has discussed how it can be ensured that data structures are updated completely and correctly, or not altered at all. Thus a crash occurring inside a recovery block appears to the user as having happened just before the outermost recovery block with respect to the values of the global data structures. It is always possible to guarantee such consistency in data structures at any time, even after a system crash which may occur at any time.

Several strategies and solutions have been discussed and in many cases there is a trade-off between efficiency and reliability. However, it looks as if one may have to pay a lot in terms of efficiency to get a system which is crash resistance at any time during the processing. The mechanisms used to update objects and cache modifications may provide crash resistance while inside any recovery block. Programs may be inside recovery blocks for 99% of the time. Only during the "critical updates", which may be done during 1% of the time, no crash resistance could be provided. Implementing multiple copies or extra levels of indirection to overcome this problem, may be very expensive.

Trade-offs are very difficult to make in general. All that can be said is that the overhead with certain strategies will be bigger than with some other strategies. The costs of a crash of class 3 are completely dependent on the specific application.

5.0 A COST ANALYSIS OF THE IMPLEMENTED RECOVERABLE PROTOTYPE SYSTEM

5.1 Introduction

This chapter analyses the costs of the provision and use of the recoverability in the recoverable (two-level) system described in this thesis. In this two-level system the first level (i.e. lower level) consists of a fault tolerant OCODE interpreter, and the second level consists of (an operating system with) a recoverable filing system.

The fault tolerant OCODE interpreter used in the prototype system was built only to provide a fault tolerant underlying machine, to the next level, with a partially recoverable interface. The fault tolerant OCODE interpreter provides an interface with recoverable OCODE types (24 bit words), unrecoverable OCODE types and unrecoverable disk blocks, as described in an earlier chapter. All of the work for this thesis has concentrated on subsequent levels (PRI-levels on a fault tolerant machine). The efficiency of the recoverable OCODE interpreter implemented was therefore not important; so an implementation which was very efficient in space (there was not much core space to spare) and could be written with little effort, was chosen. The implemented fault tolerant OCODE interpreter is, however, not very efficient in run time in cases where many different variables are updated inside recovery blocks. Details of the run time overhead and a description of the implementation details of the fault tolerant OCODE interpreter have been given elsewhere (Ver76), (Ver77a).

Comparisons and cost analysis of different ways in which such fault tolerant interpreters can be implemented have been made before (Hor74), (Ran75), (Ker74), (Von76) and will therefore not be repeated in this thesis. The cost analysis given in this chapter will concentrate on the recoverable filing system.

The cost analysis given, consists of a formal performance analysis (i.e. execution times of programs using the recoverability provided) and an analysis of the extra data space and program space required by the filing system for the provision of recoverability. A comparison is made with some alternative techniques that could have been used and some experiments have been performed to measure the overhead incurred by the use of recovery blocks.

5.2 Basic principles of the cost analysis

The cost analysis of this chapter is based on:

1. The times necessary for the execution of the operations provided.
The extra time used by the filing system for the provision of recoverability for files, is estimated by considering the extra number of disk accesses necessary to provide this recoverability.
2. System program sizes.
The total size of the programs providing a recoverable filing system is compared with the total size of the programs providing a filing system with an identical structure and identical operations, but for which no recoverability is provided.
3. The extra data space needed to keep redundant information for potential backing up.
The filing system maintains three caches (in core) and uses extra disk space, needed to keep cached pages. The total amount of cache space needed by the system is difficult to estimate in general. Some initial work in this area has been reported elsewhere (Wye73). Only some rather general statements can be made about the sizes of these caches, since the amount of cache space needed entirely depends on the use made of the system.
4. A formal analysis of the overhead incurred by the use of recovery blocks.
The overhead in the number of disk accesses needed during normal operations on files to provide recoverability, is compared with the overhead that would be required when alternative techniques were used to provide the recoverability. Other advantages and disadvantages of those techniques are also discussed in order to make a reasonable comparison.
5. Some experiments measuring the overhead incurred by the use of recovery blocks.
A number of experiments were done to measure the overhead in the number of disk accesses needed during normal processing, in order to provide recoverability.

These five aspects form a fairly comprehensive cost analysis of the recoverability provided by the system.

The only way in which the benefits of the recoverability provided could be determined would be by

measuring the availability and reliability of the system, with and without the use of recovery blocks in real life situations. For practical reasons this was not possible within the framework of the work done. It would have involved the monitoring of a system while used by a number of users. The system built is a prototype and there are no users on the system. The running of a general and arbitrarily chosen set of user programs taken from similar systems was impossible or impracticable. Such experiments would require an appreciable effort and almost make a project in itself. For these reasons the experiments were restricted to the monitoring of a little test program and a real life utility program.

A very important factor for the provision of high reliability based on recoverability is the use of a good error detection scheme. The costs of the tests involved must, for every particular application, be taken into account if the costs of reliability is to be determined, rather than just the costs of recoverability. The costs of such tests is completely ignored in the cost analysis given here.

5.3 Execution times: an analysis of the overhead in disk accesses

The execution times necessary for the file operations are completely determined by the number of disk accesses. (The system is a single user system, so there will be little overlapping of CPU processing and disk accesses.) The number of disk accesses required for operations in a filing system that does not provide recoverability are compared with the number of disk accesses required for the same operations in the implemented recoverable system.

If the operations on files in the recoverable filing system are not done inside a recovery block then their implementations will be identical to those of the operations in the unrecoverable filing system. This makes the evaluation of the overhead easy to analyse, both formally and empirically.

If read access to a file is done inside a recovery block then this may be cheaper in the recoverable system than a similar read access in the unrecoverable system. This is because access paths have been changed to include the caches (see previous chapter), which are kept in core. Consequently it may happen that a header does not have to be read from disk, which may save several disk accesses.

If a header is updated then the number of disk accesses

necessary for this update are the same whether the update is done inside or outside a recovery block. If this update is done inside a recovery block then the update is postponed until the (outermost) recovery block is left. When the outermost recovery block is left the header is updated in the way in which updates are done outside recovery blocks. However, if a header is updated several times within one recovery block then this will lead to only one update of the header on disk. So the use of recovery blocks as implemented will save disk accesses in this situation.

MFL entries are only updated when files are created and destroyed, and only read when the disk address of a header is wanted. Again if a file is created and subsequently destroyed inside a recovery block then this will have no effects on the MFL-file. So using recovery blocks will lead to the same number or less disk accesses to the MFL-file.

In order to compare the costs of writing a file inside with not inside a recovery block, the actions performed and disk accesses done for the implementation of the two basic update operations: 'update_a_data_page' and 'add_a_data_page' are considered. (Operation 'delete_a_data_page' is basically similar to 'add_a_data_page'.) Table 5.1 below shows the operations performed on file bodies for the implementation of operations 'update_a_data_page' and 'add_a_data_page'.

operation	I unrecoverable system	II recoverable system
A update_a_data_page	1.1 read data page 1.2 write new value in data page	i) The update action. 2.1 read data page 2.2 write new value in newly allocated page if this page has not been updated inside this recovery block yet, else write to original page and goto (2.7) 2.3 read directory page pointing to original page that has just been updated (replaced) 2.4 write new value of directory page to newly allocated page if this directory page has not been updated inside this recovery

		<p>block yet, else write it to original page.</p> <p>2.5 if the directory page just updated was the top directory page or was overwritten instead of replaced then goto (2.6) else goto (2.3).</p> <p>ii) At the end of the (outermost) recovery block</p> <p>2.6 free the original data and directory pages that have been replaced.</p> <p>2.7 finish.</p>
<p>B add_a_data_page</p>	<p>3.1 write value of new data page to newly allocated page.</p> <p>3.2 read the directory page that must point to the new data page.</p> <p>3.3 write new value to this directory page.</p> <p>Note: the case where a directory page is full and a new one is to be allocated for the new entry, is ignored here.</p>	<p>i) The update action.</p> <p>4.1 write value of new page to newly allocated page.</p> <p>4.2 read directory that must point to the newly allocated page.</p> <p>4.3 like 2.4</p> <p>4.4 if the directory page just updated was the topdirectory page or was overwritten instead of replaced then goto (4.5) else goto (4.2).</p> <p>ii) At the end of the (outermost) recovery block.</p> <p>4.5 free the original directory pages that have been replaced.</p> <p>4.6 finish.</p>

Table 5.1, A comparison of file body operations in the recoverable and unrecoverable system.

Free pages are marked "free" and their address is in the Free Store File. Updating of the Free Store File

involves very few disk accesses and will therefore be ignored.

In both the recoverable and unrecoverable system freeing a page involves an extra disk write. The "free"-mark in a page is redundant, because there is a list of free pages in the Free Store File (see a previous chapter). Dispensing with this redundant "free"-mark in free pages would also dispense with the extra disk writes necessary to free pages. However, it is felt to be desirable to keep this redundant mark in order to have a check in case the critical update (see chapter four), i.e. the updating of the Header file and MFL-file, goes wrong. Another reason for maintaining "free" marks is that updates of the Free Store File are buffered, so after a crash it may not contain all the addresses of free pages, or addresses of pages that have been allocated just before the crash. A free page is only given on request if the address of that page is in the Free Store File and the page is marked "free". Allocating a free page therefore involves a disk read to check the "free" mark. If the critical update is done in an absolute crash resistant way (see previous chapters) and the Free Store File is updated as first action of the critical update (part of the Free Store File is kept in core and must be forced out) then the "free" mark in free pages will not be necessary.

The number of disk accesses necessary for the two operations under the two systems are given in Table 5.2 below.

	I (unrecoverable system)	II (recoverable system)
A update_a_data_page	2	between 2 and $4+4*\text{number-of-directory-levels}$
B add_a_data_page	4	between 4 and $6+4*(\text{number-of-directory-levels} - 1)$

Table 5.2, The number of disk accesses necessary for the two operations shown in Table 5.1, when performed inside and when performed outside a recovery block.

If free pages were not marked "free" then the upperbounds for the two operations A and B under II (i.e.

inside a recovery block) would be:

$(2 + 2*\text{number-of-directory-levels}),$ and

$(3 + 2*(\text{number-of-directory-levels}-1)),$ respectively. The upperbounds given for an operation inside a recovery block represent the costs of the first update of a file body inside a recovery block. Every subsequent update inside the same recovery block will be cheaper. In order to show this an example is given below.

A few concrete examples of operations on a file F are described to show what these figures and formulae given above can mean in practice. Suppose that file F consists of one directory page and 50 data pages (pointed to by this one directory page).

If one disk page of file F is updated inside a recovery block then this will cause: $4+4*1=8$ disk accesses to be done. If no "free" marks were used in free pages then it would cause $2+2*1=4$ disk accesses. (The allocation of two pages costs two reads to check the "free" marks, the freeing of two pages costs two writes. Thus four disk accesses are saved by dispensing with the "free" marks.) If the update is done outside a recovery block then 2 disk accesses are needed. (The directory page is not updated, which saves another two disk accesses.)

If ten disk pages of file F are updated inside a recovery block then this will cause: $4+4*1=8$ disk accesses for the first update and $9*6=54$ disk accesses for the other nine disk page updates. Thus in total $8+54=62$ disk accesses are done. Every subsequent update costs still 6 disk accesses: one to read the data page, one to check the "free" mark in the newly allocated page, one to write its new value to a newly allocated page, one to read the directory page to be updated, one to update that page and finally one to free the replaced data page. If the 10 updates were done immediately after each other and a good buffer management system were used, then the reading of the directory page from disk 10 times could be avoided, because the page would probably still be in a buffer. Also writing of that same directory page 10 times could be avoided. So by using such a buffer management system, the 62 disk accesses could be brought down to just 44 disk accesses. The ACCEPT FS and UNDO FS (see previous chapters) would have to force buffers to disk or "clean" buffers, if necessary, as part of the filing system processing at the end of a recovery block alternative.

If no "free" marks were used in free pages then it would cause $2+2*1$ (for the first page) + $9*4$ (for the remaining nine pages) = 40 disk accesses. A good buffer management system could take this down to just 22 disk

accesses (if the updates are done without access to other files in the mean time, which could cause the loss of the directory page from the buffer.) If, however, the updates are done not inside any recovery block then $10*2=20$ disk accesses are needed.

If the same page is updated (assumed is that it is partially updated, so a read before write is required) 10 times, then this will require $4+4*1=8$ disk accesses for the first update and $2*9=18$ for the other nine updates. This totals 26 disk accesses. If no "free" marks were used then this number would be $2+2*1=4$ (for the first update) + $9*2$ (for the other nine updates) = 22 disk accesses.

Consequently the more often disk pages of one file are updated inside a recovery block, the smaller will be the relative overhead.

Summarizing:

It is impossible to generalise the differences in numbers of disk accesses necessary to do operations on files in the recoverable and the unrecoverable system, for the following reasons:

- * If recovery blocks are used then generally less disk accesses will be necessary for reading and writing of headers and MFL entries. (Clearly the standard system could be changed to incorporate similar buffers.)
- * Operations on file bodies will require more disk accesses if recovery blocks are used; however, it is hard to say how many more on average, because this depends on the series of operations performed inside one recovery block.

In general, however, it seems reasonable to expect that operations inside recovery blocks will cost more disk accesses than operations performed outside recovery blocks, because the number of disk accesses necessary to update headers and MFL entries is in the OS/6 environment in general small compared with the number of disk accesses necessary to update the file bodies.

The overhead in disk accesses in the implemented system is as high as shown in Table 5.2 and the examples, for two reasons:

1. A "free" mark is placed in free pages. Since updating a page the first time inside a recovery block, involves the replacing of that page by another page, a free page is to be allocated (and checked) and the original page is to be freed at the end of the recovery block. The use of these "free" marks could be avoided as described

above, which would save the extra disk accesses.

2. Since a data page is replaced, when updated (for the first time) inside a recovery block, the directory page pointing to it has to be updated as well. However, a good buffer management system could bring down the overhead incurred by having to update a directory page many times if many pages it points to, are updated.

5.4 Program sizes

An unrecoverable filing system with exactly the same structure as the recoverable system, was designed and embedded in OS/6 (to replace the original OS/6 filing system). This filing system occupies 9K bytes. The total size of OS/6, with the filing system, is 23K bytes.

In order to make the filing system recoverable, the cacheing mechanisms had to be incorporated. This meant that many filing system programs had to incorporate operations which make cache-entries, and the cache manipulation programs and ENTER_FS, UNDO_FS and ACCEPT_FS (the enter, undo and accept procedures, see previous chapters) had to be written. The new recoverable filing system now occupies 16.5K bytes. Other programs, providing the new recovery block structure (see a previous chapter) and supporting some other facilities required (such as an allocation mechanism for unrecoverable store, which is used for cache (log) space, see a previous chapter) occupy about 0.5K bytes. Therefore the total OS/6 system with recoverable filing system, occupies 31K bytes, an increase of about 30%.

5.5 Cache space

As shown in chapter three, "the cache" in the filing system consists of:

1. The disk pages that are part of new versions of files, but not of the original versions.
2. The page cache, header cache and MFL cache.

The header cache and MFL cache will generally not be very big, because a user will not update many files within one program. The page cache will generally not be very big either, because a user will normally, in OS/6, not update thousands of disk pages inside one program. Initial experiments performed sofar seem to indicate that the sizes

of the caches (page cache, header cache and MFL cache) will not be significant; most users are not expected to need more than a few hundred words of cache space (1 word = 3 bytes).

The number of extra disk pages needed for keeping several versions of files depends completely on the way in which recovery blocks are used and on the operations performed inside these recovery blocks.

5.6 A comparison with other techniques

The whole filing system has been designed and built in such a way that recovery and crash resistance were provided as features of the total system, rather than having been grafted on. A careful replacement technique was used together with the page cache, header cache and MFL cache, to provide these features. The careful replacement technique used in combination with these three caches, therefore implements a cacheing scheme. The three caches are kept in core and the costs of maintaining them is negligible compared with the costs of the disk accesses required. The costs of the careful replacement technique, as used in our system, is therefore compared with other techniques that can be used to provide recovery as defined in the present thesis and provided in our filing system.

Some of the most obvious and reasonable alternative techniques (see also the survey given in chapter two) that could be used to provide recoverability for files are described in this section and compared with the careful replacement technique used:

* An audit trail based technique.

An audit trail could be regarded and used as a cache. However, if an object was updated more than once within a recovery block then more than one entry would be put in the audit trail, because the audit trail keeps an entry for every operation performed. The audit trail is thus likely to contain many more entries than a standard cache. This would lead to gross inefficiencies in the cache processing. The audit trail scheme does not provide crash resistance either, but the system can always be backed out after a crash if the writing to the audit trail is not buffered.

The audit trail technique used in this fashion could be optimised in the manner of a cacheing scheme so that previous values of objects which are updated within recovery blocks, are not written to the audit trail more than once for each recovery block in which they are

updated. This is nothing more than the normal cacheing scheme (Hor74), (Ran75) for disk pages, in our filing system. Thus recoverability would be provided for disk pages, and recoverability for files would be provided implicitly by mapping files onto recoverable disk pages. If the cache is (partially) kept on disk then the updates on the Free Store File should not be cached. A table, similar to the page cache in the present system, could be used to indicate which pages (on disk or tape) contain previous values of the updated pages. This recovery technique cannot, as the careful replacement scheme could, be used to provide crash resistance, but recovery after a crash is possible using the cache. This recovery is only possible after a class 1 crash (see previous chapter), because if a page containing a previous value is corrupted, no normal recovery as discussed here is possible. This optimised audit trail scheme is much more efficient than the ordinary audit trail scheme. The ordinary audit trail scheme will therefore not be discussed any further.

The optimised audit trail scheme, is in fact similar to the cacheing scheme providing recovery for file words, using method I, as described in chapter four. That scheme was compared with the cacheing scheme used in the implemented system. The scheme in the implemented system has definite advantages with respect to crash resistance and in multi-user environments, because the original version of a file is not altered and still accessible while a user is updating the file (inside a recovery block). The provision of recovery is embedded in the total design of the system. It was argued in earlier chapters of this thesis that recovery may impose special requirements on the system and data structures. Embedding recovery in the system such that it forms an integral part of that system was shown to be almost essential if recovery is to be provided as in our filing system. The data structures in our system remain the same while operations are performed or backing out is done. The audit trail technique, however, is in fact an external tool which is grafted on the system and the complete system, with the recovery techniques, will thus be less structured. So the careful replacement scheme is more structured and should therefore lead to better and more reliable software.

The optimised audit trail scheme uses 3 disk accesses if the cache is on disk, or 2 disk accesses plus a tape write, when a disk page is updated for the first time within a recovery block (assumed is that a read before write is needed). The number of disk accesses per update in the careful replacement scheme used was shown to depend on a number of factors and could on average be anything between just over 2, and 8

(if there is only one directory page in the file body). If a series of pages is updated and no "free" marks are placed in free pages and a good buffer management system is used, then this average is likely to be around 3. So the optimised audit trail scheme may be more efficient, but will not always be more efficient, than the scheme implemented. It entirely depends on the use made of the system.

* A differential file based technique.

A differential file could be regarded and used as a associative cache (Hor74) which is described as method II in the previous chapter. A full description of the differential file technique is given in the survey of techniques in a previous chapter.

Basically the scheme is exactly like the optimised audit trail scheme, but new values, rather than previous values of disk pages are cached. The costs in number of disk accesses required for each first update of a page within a recovery block, are the same as for the optimised audit trail scheme: so there are no obvious general advantages in efficiency compared with the present technique.

The disadvantages of the scheme are:

1. The differential files have to be merged with the main files after a successful acceptance test of the outermost recovery block. This may be expensive, certainly if crash resistance is to be provided during the merging. In the present system it merely involves the freeing of some pages.
2. An access to a page must first search the differential file. This could be solved by keeping a table similar to the one described for the optimised audit trail scheme (this table resembles the page cache in the present scheme).

The two disadvantages of the optimised audit trail scheme (with respect to crash resistance and multi-user environments) compared with the present scheme, can only exist for the differential file technique during the merging of the differential files with the main files. The main disadvantage is that merging has to be done and may be expensive. Another advantage of the careful replacement technique, compared with the differential file technique, is that the scheme is more structured, as described above, so the software should be more reliable (for the same reasons as mentioned above, although the differential file technique is more integrated than an audit trail technique).

- * The only other technique that seems worth considering is the backup version and current version technique.

This technique would involve the creation of a backup copy for every file operated upon inside a recovery block. If recovery blocks are nested then many versions may have to be created. It is obvious that the creation of a complete backup copy of a file before it is updated inside a recovery block will, in general, be too inefficient. The scheme could be optimised, as for example for segments in System R (Lor77), using different versions of page tables pointing to the files data pages, such that the different versions of the files overlap in identical pages. This, however, still implies the dynamic creation of such a page table for each file updated inside a recovery block. If a particular page table fits in one disk page then this scheme is identical to the careful replacement scheme used (although no nesting of recovery blocks (transactions) in System R is possible). If it does not fit in one disk page then the overhead of having to copy the whole page table will incur a bigger overhead than the overhead in the present system. The significance of this extra overhead depends on the number of disk pages occupied by the page table and the average number of updates of data pages inside a recovery block. This page table versions scheme provides the same facilities as the present scheme and has the same advantages and disadvantages, but will generally be less efficient.

5.7 Some experimental results

Two experiments have been performed to examine the overhead in disk accesses incurred by the use of recovery blocks. The first one involved a little test program, which was run without any recovery blocks and with recovery blocks used in different ways. The second experiment involved a real-life utility program, which was run to do certain tasks both with and without using recovery blocks. The experiments were such that the acceptance tests of all of the recovery blocks used were trivial and always successful, since we were only interested in measuring the overheads incurred by the use of recovery blocks.

When considering the figures given in this section for the number of disk accesses for different programs, the following two aspects have to be taken into account:

1. The number of disk accesses required by a program may not always be the same, if that program is run several times. For example, the creation of a header will require more disk accesses in case that header does not fit in the last data page of the Header file, than when it does fit in that last page.
2. The filing system was developed as part of the research done. The resulting prototype is, as most prototypes are, not very efficient, and no I/O buffers are used. However, the experiments are still useful and give, in the opinion of the author, a reasonable impression of the overhead required even for more efficient implementations.

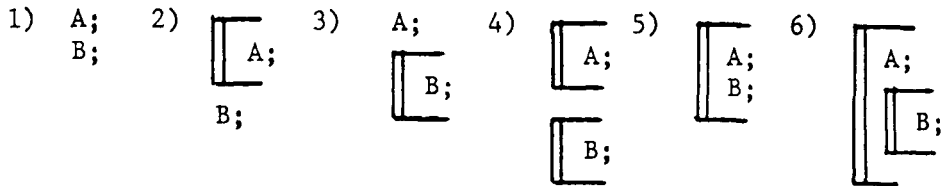
5.7.1 A little test program


A little test program was written to examine the overhead incurred by the use of recovery blocks in a situation where it would be exactly known what was happening. The program consists of two parts:

A: Create a file "TEST" and put a vector of 155 words in it (data pages can contain 55 words).

B: Append another vector of 155 words to file "TEST".

This program was run in the following six structures:



Where:  denotes: inside a recovery block.

The number of disk accesses performed for the running of these programs has been subdivided into reads and writes of free pages, directory pages and data pages. The measured number of disk accesses are given in table 5.3.

disk accesses - program		1	2	3	4	5	6
writes:	free pages	0	0	2	2	0	1
	directory pages	7	8	8	8	7	9
	data pages	14	15	16	13	10	19
	total	21	23	26	23	17	29
reads:	free pages	7	8	8	9	7	9
	directory pages	35	42	23	36	24	42
	data pages	54	63	52	58	37	74
	total	96	113	93	103	68	125
total number of accesses		117	136	119	126	85	154

Table 5.3, The measured number of disk accesses of a test program using recovery blocks in different ways.

The only program which is cheaper than 1) is, not surprisingly, program 5). A lot of disk accesses are saved by not having to read the headers of TEST and the system index (see chapter four) from disk every time: they will be in the cache after the first time they are used, because a field "date-last-accessed" will be updated when a header is read, causing the new header value to be cached.

The overhead is, in general, not very high. Only for program 6 the overhead is just over 30% of the disk accesses required when no recovery blocks are used. For the other programs the overhead is about 15% or less.

5.7.2 A real-life utility program

Similar tests as with the test program in the previous subsection, have been done with a BCPL compiler. There are two good reasons for choosing the BCPL compiler to do these experiments. The first reason is that the compiler is very much a real-life program and, although the exact text may not have been published, it is generally known how the compiler works. The second reason is that the compiler operates on three files called TOKENS, OCODE and TEXT. It empties these files and then uses them to store the tokens in the TOKENS file, the produced OCODE vector in the OCODE file and the final text in the TEXT file. So the compiler uses the filing system fairly substantially. The compiler basically consists of five parts which are invoked subsequently:

LEX: the lexicographical analyser, which reads the source code from cards and writes the tokens to the TOKENS file.

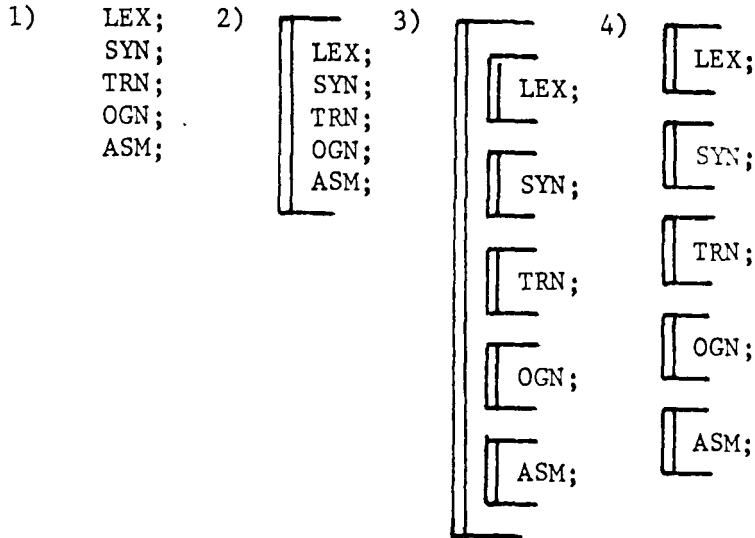
SYN: the syntactical analyser, which reads the tokens from the TOKENS file and produces an in-core tree.

TRN: the transformer, which transforms the identifiers into internal addresses, so it resolves the addressing problems. (This module transforms the in-core tree.)

OGN: the code generator, which takes the in-core tree and builds an OCODE vector which it writes to the OCODE file.

ASM: the assembler, which reads the OCODE from the OCODE file and produces the final object code which it writes to the TEXT file.

The program was run in the following four structures to compile the same BCPL program (of about 30 lines of code) for each case:



Where: [denotes: inside a recovery block.

The number of disk accesses performed by the compiler in these structures has been subdivided again into reads and writes of free pages, directory pages and data pages. The measured number of disk accesses are given in table 5.4.

disk accesses - program		1	2	3	4
writes:	free pages	12	12	14	14
	directory pages	12	12	12	12
	data pages	40	28	46	48
	total	64	52	72	74
reads:	free pages	12	12	14	14
	directory pages	124	82	164	208
	data pages	182	98	336	324
	total	318	193	514	546
total number of accesses		382	244	586	620

Table 5.4, The measured number of disk accesses of a BCPL compiler using recovery blocks in different ways.

The results are very similar to those obtained with the little test program in the previous experiment. The figures can be explained like with the little test program and need no further commenting.

5.8 Conclusions

The recoverable filing system works exactly like the standard filing system if no recovery blocks are used. If recovery blocks are used then an overhead in disk accesses will be incurred. This overhead will be significant if file updates are sparse; if the number of updates of pages of one file inside a recovery block is high, then the overhead will be relatively small for the operations on that file, especially if a good buffer management system is used. The extra space necessary for the caches kept in core is generally not expected to be much. The number of extra disk pages necessary to maintain several versions of files is difficult to estimate. The programs comprising the standard filing system occupy 9K bytes, the programs comprising the recoverable filing system occupy 16.5K bytes.

The major overheads are caused by the use of "free" marks in free pages and the fact that the careful replacement technique also causes directory pages to be replaced whenever a data page is replaced (or added or deleted), if that directory page has not yet been replaced inside the current recovery block. Consequently, the more pages of a file are updated inside a recovery block, the less significant the overhead will be. (Pages are only replaced once inside a recovery block: only one value needs to be cached.)

The careful replacement technique used appeared to compare well with other schemes. It is difficult to make general conclusions about the efficiency of the technique compared with other techniques, because much depends on the use made of the system. However, certainly if a good buffer management technique is incorporated in the system, keeping directory pages as long as possible, and if free pages are not marked "free", then the present technique is not expected to incur a bigger overhead than other techniques. The present technique was shown to have other advantages over techniques that are more efficient under particular circumstances. These advantages are the provision of crash resistance and the structured way in which the provision of recovery is embedded in the system, rather than having been grafted on the system. These factors make the maintenance of consistency in the system much easier.

6.0 DIRECTIONS FOR FUTURE RESEARCH AND CONCLUSIONS

6.1 Directions for future research

The present thesis has tackled the problems of providing recovery in multi-level systems and for complex data, but only in a uni-process environment. One of the important topics of ongoing research is the provision of recovery for parallel, possibly interacting, processes. There are two ways in which this is approached at present (these two approaches are basically also distinguished by Curtice (Cur77)):

1. Prevent the interactions.

Clearly this approach is only feasible when the interactions are not required, so that the results that would be obtained by, for example, executing the processes in sequences are acceptable. This can, for example, be done by using a locking scheme, either explicitly (Gra76) or implicitly (BaS77). The explicit locking schemes are widely used in data base systems. A user, or a program, can, in general, request access to an object, in various modes. Gray, for example, distinguishes six modes, such as exclusive access or shared access. Several access modes are incompatible, which means that if one user has access to an object in a certain mode, then another user is refused access to the same object in an incompatible mode. Thus such locking schemes prevent unwanted interactions, while still allowing shared access to objects in cases where this will not lead to any unwanted effects of one program on another program. Implicit locking by programs is done if, for example, monitors (Hoa74) are used to implement resource allocation algorithms. Programs wishing to acquire and release resources invoke appropriate monitor procedure calls. If these locking schemes are used then basically uni-process recovery techniques can be used, because the schemes ensure that only one process at a time will update an object and commit itself before releasing the object. So other processes are prevented from updating that object as long as the process which is updating that object is in a unit of recovery, which may be, for example, a transaction (Gra76) or a recovery block (BaS77). A similar way in which the restrictions can be enforced is by using a capability architecture (Den76) to implement a high degree of error confinement (Lin76). Compared with a lock, as used in a locking scheme, a capability could be best described as a key;

a capability is a generalized permission to use storage objects and procedures. Capabilities are, therefore, in fact a means of implementing locking. The use of capabilities prevents unwanted interactions like locking schemes, and therefore limits the risk that errors will do much damage before being detected.

2. Synchronize the processes with respect to recovery.

Where interactions are intended and required, a conversation, as described in chapter one, incorporates processes working on one task and prevents other tasks from reading or writing objects updated by that task. If one process involved in a conversation fails, then the effects of the operations performed inside the conversation by that process and all other processes involved in the conversation will be undone. In order to be able to back out all the processes involved so-called recovery lines (RLT77) must exist. A recovery line is a set of consistent recovery points (states that can be reinstated by the recovery mechanism) for the process which is in error, and for all other processes affected.

These approaches are designed to overcome one of the major recovery problems of parallelism, namely the so-called domino-effect (Ran75). This effect occurs if no recovery lines exist for the interacting processes. This means that all the processes that have interacted with a failing process, and all the processes that interacted with those processes, and so on, will have to be backed out to their "begin"-states. The two approaches above force the system to progress such that recovery points always form recovery lines, and so can be used if a failure occurs.

The present thesis has discussed recovery by state restoration. Whenever a failure occurs, a state which is hoped to be error-free is restored before attempting to continue further operation. Other recovery techniques deserve further investigation, for example:

* Error diagnosis and repair.

Instead of restoring a state when an error is detected, an attempt could be made to identify the fault that caused the error (RLT77), and repair the error(s) it has caused. This may be very difficult, because different errors may be caused by one fault and different faults may cause the same error.

* Compensation.

Rather than undoing operations by state restoration, it could be attempted to nullify the impacts of these operations by compensating their effects. This can be done by providing supplementary corrective information (Dav72), (RLT77). For example, if a data base had been updated to indicate that an employee has been given a £1000 wage increase, instead of an intended £100 increase, then a wage decrease of £900 could be given as compensation, rather than invoke some general form of backing up of the data base.

Recovery techniques have been discussed and a survey of existing techniques has been given. This thesis has not concentrated on the costs of recoverability very much. Different kinds of recovery can be used to provide recovery for different kinds of failures or to provide different degrees of recovery. For example, recovery may just restore the data structures, such that they are in consistent or valid states again, or it may restore the data to previously existing valid states. Hardly any work has been done to examine which failures recovery should cope with and what degrees of recovery are required in different environments, and what the costs are of providing them. The survey given provides some general guide lines, rather than detailed analyses of some degrees of recovery.

Apart from the costs of the recovery mechanisms used, the costs of the error detection scheme used also have to be taken into account. The error detection scheme used, and its cost, will depend on the failures the recovery mechanism has to cope with. Hardly any work, so far, has been done on systematic approaches to error detection. Error detection is absolutely essential to make recovery useful as a mechanism for providing fault tolerance. Error detection, using tests, could be done using the following two approaches:

1. Test if algorithms perform completely according to their specifications.
2. Distinguish certain types of faults and errors, and test for their presence (or absence).

Testing the validity of all of the input data and parameters of procedures is an example of a systematic error detection scheme based on the second approach. However, few systematic ways in which tests can be constructed, using either approach, are reported in literature (RLT77) and little is known about the costs of error detection schemes. Error detection in system software is generally done in an ad hoc fashion using the second approach. Some initial work on the construction of (run time) tests, using the first approach, has been done and the results of this work and the problems encountered are the subject of the rest of this

section.

6.1.1 The construction of complete acceptance tests

To date, hardly any work has been done on the construction of run time tests for testing whether or not the results of the programs comply with the specifications of the programs. The notion of acceptance test, as used in recovery blocks, will be used to denote such tests for the purpose of this section. An acceptance test is just a special (syntactic) form of error detection and can easily be generalized to any other form of error detection. A complete acceptance test is a test which tests whether all the required effects of a computation have been achieved.

It is probably contrary to the spirit of acceptance tests to expect or require them to be as complete as specifications of the programs they test, because they are just a special form of error detection. Acceptance tests could, for example, be used to check only redundant data or check if the data are consistent (rather than whether they have the correct value). However, the designer of the acceptance test may want to know what would constitute a complete acceptance test, so that he can decide what to put in the (run time) test and what not. If a test is constructed by trying to think of a few important effects of the program to be checked then the designer of the test will not have a very good idea of the degree of completeness of the test; he doesn't know what the loopholes are, nor how big they are.

A program designer may design his programs such that redundant data, such as sumchecks, and tests on its correctness are incorporated in programs and data. This may increase the reliability of the programs enormously. However, this approach will not be considered here. This section will concentrate on acceptance tests that try to test whether the required effects of programs were achieved or not. Methods to find complete acceptance tests (which may be cut down for efficiency reasons, in a particular implementation) will be investigated.

To illustrate how difficult it is to construct complete acceptance tests I will show a simple inconsistency in tests used in a very simple example given in a report describing the verification and abstraction in ALPHARD (WLS76).

ALPHARD provides the programmer a very nice framework within which pre- and post-conditions for operations on objects of the types being specified can be included. On page 15 of this report the specification of type "stack" is

given to illustrate the concepts. The post-condition for the operation "pop" on a stack is:

$(s.sp=s.sp'-1)$

where s = the stack

$s.sp$ = the stack pointer

An apostrophy is used to indicate the value before the operation was performed.

So after a "pop" only the effects on the stack pointer are checked. This is of course not a complete test since also the effects on the stack should be checked. In contrast the post-condition given for the operation "push" is complete, namely:

$(s.sp=s.sp'+1) \ \& \ (s.v=a(s.v',s.sp',x))$

where $s.v$ = a vector used to represent the stack

$a(v,n,x)$ = a vector identical to v except that $v(n)=x$.

These kinds of errors in the construction of tests (or conditions in ALPHARD) are very easy to make (even for very simple types, as has been shown), unless these tests are constructed such that their completeness and sufficientness can be proven easily. This is still a big problem. Another big problem with the use of tests in multi-level systems is an efficiency problem. It is shown that a system may spend more time doing tests than actual processing, if complete tests are used in multi-level systems.

6.1.2 A single computation

An abstract specification of a program specifies the meaning and function of a program and the effects visible to the user of the program. It does not specify actions to be taken or operations to be performed for achieving these effects and for implementing the program's functions. A concrete specification of a program, however, does specify these actions or operations.

If a program is a computation for which no abstract specification exists, for example a numerical analysis program built from a set of formulae (a very concrete specification), then in general an acceptance test can only be done by recomputing the result. The programmer may be lucky in that that result (i.e. the effects of the program which are visible to the user) can be checked easily. It may also be possible that certain properties that must hold for the result can be used to construct a number of tests which may give a very good (possibly incomplete) acceptance test. A test which simply does the same computation, is in fact an application of majority voting rather than a complete checking of the results.

6.1.3 The construction of acceptance tests from abstract specifications

Programs in software systems considered in this thesis, are programs that can be regarded as implementations of operations on abstract types. There are several formal methods in which data types and operations on them can be defined. A very good survey of existing techniques is given by Liskov and Zilles (LiZ75). Formal specifications could also be used to design and implement the acceptance tests for the operations.

One of the most promising specification methods (for building acceptance tests using specifications) is Guttag's specification method for abstract data types (Gut75). This is the only known specification method which disentangles the abstract meaning of a data type from a particular representation of it. The specification method consists of two parts:

1. A syntactic specification
2. A set of relations (axioms).

As an example the specification of type "stack" might be as follows:

1) Syntactic specification:

NEWSTACK :	--> stack
PUSH : stack*integer	--> stack
POP : stack	--> stack
TOP : stack	--> integer

2) Axioms:

TOP(NEWSTACK)	= error
TOP(PUSH(stack1, integer1))	= integer1
POP(NEWSTACK)	= error
POP(PUSH(stack1, integer1))	= stack1

Suppose that the specifications are given, then the question is whether acceptance tests, for the operations defined, can be constructed using the axioms. What is to be checked by the acceptance test of an operation is whether the operation performed did not contradict any of the axioms. If it can be shown that the operation did not contradict any of the axioms then it follows automatically that the operation was performed according to the specifications. If the acceptance test tests all of the axioms then the correctness of the acceptance test is implicit and does not have to be proven explicitly.

In order to "construct" an acceptance test for an operation on a type, using the axioms, the following is

done:

- * Take all the axioms in which the operation occurs.
- * Use these axioms to formulate an acceptance test for that operation.
- * Show that none of these axioms can ever be in contradiction with the acceptance test.

To show how this could be done the acceptance test for PUSH(S,x) (see specification above) is constructed as follows:

- * There are two axioms in which PUSH occurs:
TOP(PUSH(S,x))=x
POP(PUSH(S,x))=S
- * The acceptance test AT(PUSH(S,x)) is:
TOP(S)≠x & POP(S)≠ prior S.
(Note: S stands for the value of the stack after the execution of PUSH(S,x), prior S denotes the value of S before the execution of PUSH(S,x).)
The operation ≠ is a boolean operation which returns "true" if both operands have equal value and "false" otherwise.
- * This acceptance test merely checks whether or not the operation PUSH complies with its definition since it just validates the results of the operation against the axioms. It is obvious that the axioms are not contradicted when the acceptance test is true, and if the operation PUSH has been performed correctly then the acceptance test must be true.

In general the axioms cannot be used so easily to construct the parts of the acceptance test. To show this we examine the construction of the acceptance test for TOP(S) (x is the result of operation TOP(S)):

- * There are two axioms in which TOP occurs:
TOP(NEWSTACK)=error
TOP(PUSH(S,x))=x
- * The acceptance test for TOP(S) consists of two parts:
 - The first part is derived from the first axiom:
if result is error then prior S ≠ NEWSTACK

- The second part is derived from the second axiom:
PUSH(? ,x) \neq prior S.
The ? in the expression is to be solved. This is simply a matter of solving a number of equations: the axioms and the equation above. Axiom POP(PUSH(S,x))=S is used to solve "?":
POP(PUSH(? ,x))=POP(prior S)=?.
So the second part of the acceptance test is:
PUSH(POP(prior S),x) \neq prior S.
The complete acceptance test is:
if result is error then S \neq NEWSTACK
else PUSH(POP(prior S,x) \neq prior S.

* The proof of correctness of the acceptance test is very trivial. The axioms involved can easily be used to show that if the acceptance test is true then the axioms are obeyed, and if the operation was performed correctly (the axioms were not contradicted) then the acceptance test will give the result "true". This proof will not be shown here.

The acceptance tests obtained in the way shown above are exactly the post-conditions in the specifications in ALPHARD. These tests can also be "translated" into so-called output assertions (WLS76), which are the post-conditions in terms of the representation used for the type. The acceptance test of an operation could be specified either in abstract terms (the post-condition) or concrete terms (the output assertion) and for the sake of efficiency could be made incomplete. However, the complete test will be known and the programmer will know the risks and loopholes if he decides to use an incomplete test.

An acceptance test (for a certain operation on a type) makes use of other operations on the type, but acceptance tests for those operations are not incorporated in those operations. Obviously this is not necessary, because if, for example, always TOP(S)=x & POP(S)= prior S (operations TOP and POP without acceptance test) after PUSH(S,x) then the program tested provides a correct implementation of PUSH(S,x).

At first sight the method used seems a good one. Unfortunately, however, this method suffers from two big problems if complete acceptance tests are required. The first problem is called the "computability" problem, the second one is called the "constructability" problem.

The first problem (the computability problem) is that predicates on instances of the specified types are not always computable given the operations on the abstract type. For example the operation " \neq " for type stack is not given in the specification of stack shown above. However, this operation is needed in the acceptance tests of operations on

- * Axioms.
 1. ?EMPTY?(EMPTY) = true
 2. ?EMPTY?(ADD(Q,i)) = false
 3. FRONT(EMPTY) = error
 4. FRONT(ADD(Q,i)) = if ?EMPTY?(Q) then i else FRONT(Q)
 5. REMOVE(EMPTY) = error
 6. REMOVE(ADD(Q,i)) = if ?EMPTY?(Q) then EMPTY else ADD(REMOVE(Q),i)

The acceptance test for $x=FRONT(Q)$ is not so easy to construct. The construction of it is tried below.

- * Applying the third axiom gives: if result is (error) then prior Q = EMPTY ... (1).

- * Applying the fourth axiom gives: prior Q = ADD(?1,?2).
 ?1 and ?2 have to be found. Applying FRONT gives:
 FRONT(prior Q) = FRONT(ADD(?1,?2)) = (ax.4)
if ?EMPTY?(?1) then ?2 else FRONT(?1) ... (*)
 Axiom 6 gives: Q=ADD(EMPTY,x) ==> REMOVE(Q)=EMPTY.
 Using (*) this gives:
if ?EMPTY?(REMOVE(prior Q)) then prior Q=ADD(EMPTY,x).
 ?1 is still to be resolved in:
if \neg ?EMPTY?(?1) then FRONT(?1) = x.
 The term \neg ?EMPTY?(?1) can be replaced by:
 \neg ?EMPTY?(REMOVE(prior Q)).

The biggest problem is to find a useful equivalent for $x=FRONT(?1)$. From the axioms it is obvious that ?1 is equal to prior Q without the element last put in. To find ?1 a "piece of program" which constructs ?1 is to be written, like given below:

```

q1 = EMPTY
begin
  y = FRONT(q)
  q = REMOVE(q)
  q1 = ADD(q1,y)
end repeat until ?EMPTY?(REMOVE(q))
After this program ?1 = q1.

```

The correctness of this program is far from obvious. Of course an operation to form ?1 could be defined, but that would be a rather strange operation for this type queue, and its specification might be as difficult as proving the correctness of the piece of program shown above.

Obviously this solution is not satisfactory at all. The problem is that elements are put in the queue at the "bottom" and taken away from the "top". The only way in which the bottom element can be seen is by taking away all elements from the "top" of the queue.

Summarizing:

- * If the specification method used exhibits a representational bias then the acceptance test programmer will probably have to know certain implementation details. This makes the proving of the sufficiently completeness of both the specifications and the acceptance tests very difficult. Therefore abstract specifications seem most useful for the construction of complete acceptance tests.
- * Abstract specifications merely specify a type (i.e. its semantics and the operations on it). Therefore they seem unsuitable for practical use for the construction of for example acceptance tests.

6.1.4 An acceptance test problem in multi-level systems

If complete acceptance tests are used in programs in a multi-level system, in the way suggested in the previous subsection, then these tests may cause a pyramid effect in tests being invoked.

An example to illustrate this effect is given below: Suppose that in a multi-level system as described by Madnick and Alsop (MaA69) the following routines exist in different levels:

- 1) write_in_file
- 2) write_in_volume
- 3) write_in_page
- 4) write_in_disk_block

Operation 1) invokes 2) which in turn invokes 3) which subsequently invokes 4).

Suppose that every routine has a complete acceptance test, then the acceptance test of 1) will be: read_file(...) plus a test on the data read. This operation read_file invokes operation read_volume. Operation read_volume invokes read_page which invokes read_disk_block. Operation write_in_file invokes operation write_in_volume. The acceptance test for write_in_volume will be: read_volume(...) plus a test on the data read. This operation read_volume invokes read_page which invokes read_disk_block. Similarly write_in_volume invokes write_in_page, which has a complete acceptance test. And so on.

So a pyramid effect is created. This effect can easily cause the system to slow down to such an extent that it spends more time performing acceptance tests than the actual operations. For example, if write_in_file, in the example

above, causes one disk write then another four disk reads will be done in the acceptance tests of the four operations (provided that complete acceptance tests are used in the four operations). So the whole operation will cost five disk accesses in stead of one.

6.2 Conclusions

The first part of this thesis has investigated the Completely Recoverable Interface (CRI) scheme and a Partially Recoverable Interface (PRI) scheme. It was shown that in certain cases the PRI scheme is less extravagant in space and time than the CRI scheme. The CRI scheme also appears to be possible only for multi-level systems consisting of interpreters. The PRI scheme has been shown to be a good alternative.

The consequences of having recoverable and unrecoverable types in a single interface and of mapping new types onto unrecoverable types have been shown. The main consequences and conclusions are:

1. A special mechanism is needed to allow a PRI level to use unrecoverable types inside recovery blocks. This mechanism is called the logging mechanism. Basically this mechanism provides, semi-automatically, sufficient recoverability for those types. The recoverability is not recoverability as defined in the beginning of this report, because the state of the machine as seen by this PRI level may not be restored exactly to its original state (for example as in the buffer management example described previously). User programs can use the new recovery block structure, provided by the logging mechanism, and are not aware of the ways in which recovery is provided for files and variables in the system. A logging mechanism has been implemented in a two-level prototype system and, as expected, did not lead to a great overhead in the programming effort. A cost analysis of this prototype is described in chapter four.
2. An unrecoverable data structure called a log is used as part of that mechanism (for the implementation described, see also Fig.3.5). The fact that this log has to consist of unrecoverable data structures has two main consequences:
 - a. Extra protection is needed for the log, since recovery from corruption is not possible.
 - b. It seems preferable also to provide unrecoverable objects of the same types as the recoverable types that are provided. The reasons for this is that a level does not know what types the next level up will wish to use for the implementation of a log. Another reason for this is, however, that higher levels may want unrecoverable objects of the newly provided types for the representation of more abstract recoverable types (for which these higher

levels wish to provide recovery explicitly).

As seen previously, it is also useful to have unrecoverable objects for debugging purposes. So even if a log does not have to be unrecoverable, as was shown to be possible in other implementations, it would still be preferable to provide unrecoverable objects of the same types as the recoverable objects provided.

In particular cases, where a level provides some new recoverable types which are mapped onto unrecoverable types of the underlying machine and does not change the interface any further, the logging mechanism appears to be very suitable for implementing what have been called recoverable type managers. Levels in, for example, operating systems in many cases can be regarded as extending an existing interface by providing some new types. Recoverable type managers appear to be very suitable for the construction of such levels (in recoverable multi-level systems). The flexible use of recoverable type managers has been demonstrated in examples given and in the implemented two-level prototype system.

If a procedural level provides new recoverable types by performing the caching and providing a new recovery block structure, then it would map these new types onto unrecoverable types (it only makes sense if they are mapped onto unrecoverable types). It was shown that this does not have to be a restriction, because levels can be made to provide recoverable and unrecoverable objects of a type in a reliable way. Higher levels can then use unrecoverable objects of that type to provide a new (more abstract) recoverable type.

The recovery-for-types approach appears to be a very good way to build up reliable multi-level systems. This approach was compared with an approach whereby recovery is provided for operations.

The major conclusion of the first part of the thesis is that partially recoverable interfaces form a useful, and for efficiency reasons probably necessary, concept for the building of recoverable multi-level systems. It has been shown that the concept is not only useful but also implementable and flexible.

The problems of providing recoverability for complex data structures have been discussed in the second part of this thesis. It appears that using a recovery cache to cache previous values of all objects is not a sufficient mechanism as such. The kind of information cached and the way in which updates are made appear to be of extreme importance if recoverability is to be provided in a

"reasonably efficient" way, and to guarantee consistency of the data structures after a crash (although no measure of efficiency has been defined). Also the way in which complex data structures are formed has been shown to be important if recovery is to be provided for those structures.

A main conclusion is that data providing an abstraction ("mapping tables") and information carrying data have to be separated both conceptually and physically whenever possible. They have to be treated differently when recoverability is to be provided for them.

Different strategies and mechanisms that guarantee that the data structures will have the states they were in before entering the (outermost) recovery block in case a crash occurs have been described and compared. These mechanisms are said to provide crash resistance. It appeared that crash resistance can always be provided, but possibly at very high costs. A prototype recoverable filing system incorporating many of the mechanisms discussed has been implemented successfully and has been described for illustration purposes.

Finally a cost analysis of the implemented recoverable two-level system (including the recoverable filing system) has been given. It is difficult to make a cost benefits comparison since benefits could not really be measured. However, the costs of the recovery mechanisms used in the filing system seemed justified, partly because the programmer may decide not to use the recovery provided, in which case there is virtually no overhead.

REFERENCES

And75

Anderson, T.,
"Provably safe programs".
Technical Report 70, Computing Laboratory,
University of Newcastle upon Tyne, England,
February 1975.

AnK76

Anderson, T., 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.

Ast75

Astrahan, M.M., Boyce, R.F., Chamberlin, D.D.,
Eswaren, K.P., Fehder, P.L., Mehl, J.W.,
"SEQUEL, Release 2".
IBM Research Laboratory, San Jose, California,
February 1975.

Ast76

Astrahan, M.M., et al,
"System R : Relational approach to data base management".
ACM Transactions on data base systems,
Vol.1,2, June 1976, pp.97-137.

Avi71

Avizienis, A., Gilley, G.C., Mathur, F.P.,
Rennels, D.A., Rohr, J.S., Rubin, D.K.,
"The STAR (Self-Testing-and-Repairing) Computer:
an investigation of the theory and practice of
fault-tolerant design".
IEEE Transactions on Computers, Vol.c20,11,
November 1971, pp.1312-1321.

Avi75

Avizienis, A.,
"Fault-tolerance and fault-intolerance:
Complementary approaches to reliable computing".
Proc. Int. Conf. on reliable software.
Los Angeles, April 1975, pp.458-464.
(SIGPLAN Notices 10,6).

BaS77

Banatre, J-P., Shrivastava, S.K.,
"Reliable resource allocation between unreliable
processes".
Technical Report 99, Computing Laboratory,
University of Newcastle upon Tyne, England,
April 1977.

BaW76

Bartussek, W., Wurges, H.,
"Proving that an implementation meets its abstract
verification".
Research group on operating systems.
T.H. Darmstadt, Germany,
Forschungsbericht BSI 76/2, Darmstadt, May 1976.

Bir73

Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., Nygaard, K.,
"SIMULA BEGIN".
Auerbach Publishers Inc., Philadelphia, Pa., 1973.

BjD72

Bjork, L.A., Davies, C.T.,
"The semantics of the presentation and recovery of
integrity in a data base system".
IBM - Technical Report, TR 02.540, December 1972.

Bjo74

Bjork, L.A.,
"Generalised audit trail (Ledger) concepts for data
base applications".
IBM - Technical Report, TR 02.641, September 1974.

Bjo75

Bjork, L.A.,
"Generalised audit trail requirements and concepts for data
base applications".
IBM Systems Journal, Vol.14,3, 1975, pp.229-245.

Bor72

Borgerson, B.R.,
"Dynamic confirmation of system integrity".
AFIPS Conf. Proc. Vol.41, part I,
AFIPS Press, Montvale, New Jersey,
1972, pp.89-96.

Boy75

Boyer, R.S., Elspas, B., Levitt, K.N.,
"SELECT, A formal system for testing and debugging programs
by symbolic execution".
Proc. Int. Conf. on reliable software.
Los Angeles, April 1975, pp.234-245.
(SIGPLAN Notices 10,6).

Cur77

Curtice, R.M.,
"integrity in data base systems".
Datamation, Vol.23,5, May 1977, pp.64-68.

DaN65

Daley, D.C., Neumann, P.G.,
"A general purpose file system for secondary storage".
FJCC 1965, pp.213-229.

Dav72

Davies, C.T.,
"A recovery/integrity architecture for a data system".
IBM, Technical Report TR 02.528,
May 1972.

DDH72

Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.,
"Structured programming".
A.P.I.C. Studies in Data Processing, No 8, 1972
(Academic Press London and New York).

Den76

Denning, P.J.,
"Fault-tolerant operating systems".
ACM Computing Surveys, Vol.8,4, December 1976, pp.359-390.

DeW73

De Witt, D.J., Schlansker, M.S., Atkins, D.E.,
"A micro programming language for the B-1726".
Preprints of the sixth annual workshop on
microprogramming.
ACM SIG MICRO, University of Maryland 1973, pp.21-29.

Dij68

Dijkstra, E.W.,
"The structure of the THE multiprogramming system".
Comm. ACM, Vol.11,5, 1968, pp.341-346.

Els72

Elsas, B., Levitt, K.N., Waldinger, R.J.,
Waksman, A.,
"An assesment of techniques for proving program correctness".
ACM Computing Surveys, Vol.4,2, June 1972, pp.97-147.

EMA74

EMAS-Report 2 : "The EMAS directory" (by Rees, D.J.).
EMAS-Report 3 : "The standard EMAS subsystem" (by Millard, G.E.,
Rees, D.J., Whitfield, H.).
EMAS-Report 4 : "The Edinburgh Multi-Access System scheduling
and allocation procedures in the resident
supervisor"
(by Shelness, N.A., Stephens, P.D., Whitfield, H.).
Department of Computer Science of Edinburgh, April 1974.

EsC75

Eswaran, K.P., Chamberlin, D.D.,
"Functional specifications of a subsystem for data base
integrity".
Proc. Int. Conf. on Very Large Data Bases,
Framingham, MA, September 1975, pp.48-68.

Fra69

Fraser, A.G.,
"Integrity of a mass storage filing system".
The Computer Journal, Vol.12,1, February 1969.

Gam73

Gamble, J.N.,
"A file storage system for a multi-machine environment".
Ph.D. thesis, Victoria University, Manchester, England,
October 1973.

Gil74

Gilb, T.,
"Parallel programming".
Datamation, Vol.20,10, October 1974, pp.160-161.

GiS

Giordano, N.J., Schwartz, M.S.,
"Data Base Recovery at CMIC".
Proc. 1976 SIGMOD Int. Conf. on Management of Data,
Washington D.C., June 2-4, 1976, pp.33-42.

GoG75

Goodenough, J.B., Gerhart, S.L.,
"Toward a theory of test data selection".
Proc. Int. Conf. on Reliable Software.
Los Angeles, April 1975, pp.493-510.
(SIGPLAN Notices 10,6).

Gra76

Gray, J.N., Lorie, R.A., Putzolu, G.R.,
Traiger, J.L.,
"Granularity of locks and degrees of consistency in
a shared data base".
Modelling in data base management systems,
(Nijssen, G.M., ed.),
North-Holland Publishing Company 1976, pp.365-394.

Gut75

Gutttag, J.V.,
"The specification and application to programming of
abstract data types".
Ph.D. thesis,
Dept. of Comp. Sc., University of Toronto, Canada,
1975.

Gut76

Gutttag, J.V., Horowitz, E., Musser, D.R.,
"Abstract data types and software validation".
Information Science Institute, California.
ISI/RR-76-48, August 1976.

Hoa74

Hoare, C.A.R.,
"Monitors: an operating system structuring concept".
Comm. ACM Vol.17,10, October 1974, pp.549-557.

Hor74

Horning, J.J., Lauer, H.C., Melliar-Smith, P.M.,
Randell, B.,
"A program structure for error detection and recovery".
Proc. Conf. on Operating Systems, IRIA,
1974, pp.177-193.

IBMa

IBM, System/360 & System/370
"Fortran IV Language".
GC28-6515-10, File no S360/370-25.

IBMb

IBM, System/360 Operating system
"PL/I (F) Language Reference Manual".
GC28-8201-4, File no S360-29.

IBMc

IBM, Information Management System.
IMS/VS, "Utilities reference manual", SH20-9029
IMS/VS, "Operators reference manual", SH20-9028
IMS/VS, "System programmer reference manual", SH20-9027

Inf75

"Infotech state of the art report,
data base systems".
Infotech Information Limited, Nicolson House,
Maidenhead, Berkshire, England,
1975.

Jac75

Jackson, M.A.,
"Principles of program design".
A.P.I.C. Studies in Data Processing, No 12, 1975
(Academic Press London and New York).

Ker74

Kerr, R.,
"An alternative implementation of the recursive cache".
Internal memo SRM/79, Computing Laboratory,
University of Newcastle upon Tyne, England,
March 1974.

Kin76

King, J.C.,
"Symbolic execution and program testing".
Comm. ACM, Vol.19,7, July 1976, pp.385-394.

Knu73

Knuth, D.E.,
"The art of computer programming, Vol.III : sorting and
searching".
Addison-Wesley, Reading, Mass., 1973.

Kop74

Kopetz, H.,
"Software redundancy in real time systems".
Proc. I.F.I.P., Stockholm, Sweden, 1974, pp.182-186.

Lam75

Lampson, B.W.,
"An open operating system for a single-user machine".
Xerox Palo Alto Research Center,
Palo Alto, USA, January 1975.

LaS76

Lampson, B., Sturgis, H.,
"Crash recovery in a distributed data storage system".
Computer Science Laboratory,
Xerox Palo Alto Research Center, Palo Alto,
California, 1976.

Lin76

Linden, T.A.,
"Operating system structure to support security and
reliable software".
ACM Computing Surveys, Vol.8,4, December 1976,
pp.409-445.

LiZ75

Liskov, B., Zilles, S.,
"Specification techniques for data abstraction".
Proc. Int. Conf. on reliable software
Los Angeles, April 1975, pp.67-72.
(SIGPLAN Notices 10,6)

LoK68

Lockemann, P.C., Knutsen, W.D.,
"Recovery of disk contents after system failure".
Comm. ACM, Vol.11,8, 1968, pp.542.

Lor77

Lorie, R.A.,
"Physical integrity in a large segmented database".
ACM Transactions on Database Systems, Vol.2,1,
March 1977, pp.91-104.

MaA69

Madnick, S.E., Alsop, J.W.,
"A modular approach to file system design".
AFIPS Conference Proc., Vol.34, 1969, pp.1-13.

Mar75

Martin, J.,
"Computer data-base organisation".
Prentice-Hall Inc., 1975.

Mar76

Martin, J.,
"Principles of data-base management".
Prentice-Hall Inc., 1976.

Mas71

Mascall, A.J.,
"Studies of the reliability and performance of computing
systems at Barclays Bank".
Internal memo SRM/10, Computing laboratory,
University of Newcastle upon Tyne, 1971.

Mas73

Mascall, A.J.,
"Checkpoint, backup and restart in a "reliable" system".
Internal memo SRM/37, Computing Laboratory,
University of Newcastle upon Tyne, April 1973.

MeS77

Melliard-Smith, P.M.,
"On reliability in data base systems".
(unpublished report, Computing
Laboratory, University of Newcastle upon Tyne, England).
1977.

MeR77

Melliard-Smith, P.M., Randell, B.,
"Software reliability: the role of programmed
exception handling".
ACM Conf. on language design for reliable software.
North Carolina, USA, March 1977, pp.95-100.
(SIGPLAN Notices 12,3)

MiM75

Miller, E.F., Melton, R.A.,
"Automated generation of testcase datasets".
Proc. Int. Conf. on Reliable Software.
Los Angeles, April 1975, pp.51-58.
(SIGPLAN Notices 10,6).

New72

Newell, G.B.,
"Security and resilience in large scale operating
systems".
1900 Series Operating Systems Division.
International Computers Limited, London, S.W. 15,
England, 1972.

Par72a

Parnas, D.L.,
"A technique for software module specification with examples".
Comm. ACM, May 1972, pp.330-336.

Par72b

Parnas, D.L.,
"On the criteria to be used in decomposing systems
into modules".
Comm. ACM, December 1972, pp.1053-1058.

Ran70

Randell, B.,
"Visit to BOAC".
Internal memo SRM/5, Computing Laboratory,
University of Newcastle upon Tyne, 1970.

Ran75

Randell, B.,
"System structure for software fault tolerance".
IEEE Trans. on Software Engineering, SE-1,2
June 1975, pp.220-232.
(also published as Technical Report no 75, Computing
Laboratory, University of Newcastle upon Tyne, England).

Rap75

Rappaport, R.L.,
"File structure design to facilitate on-line
instantaneous updating".
ACM SIGMOD Conf. May 1975, pp.1-14.

Ric69

Richards, M.,
"BCPL: a tool for compiler writing and system
programming".
Spring Joint Computer Conference 1969, pp.557-566.

Ric71

Richards, M.,
"The portability of the BCPL compiler".
Software-Practice and Experience, Vol.1,
April-June 1971, pp.135-146.

Ric73

Richards, M.,
"The BCPL Programming manual".
The Computing Laboratory, University of Cambridge,
England, 1973.

RLT77

Randell, B., Lee, P.A., Treleaven, P.C.,
"Reliable computing systems".
Technical Report 102, Computing Laboratory,
University of Newcastle upon Tyne, England,
June 1977.

Rob75

Robinson, L., Levitt, K.N., Neumann, P.G.,
Saxena, A.R.,
"On attaining reliable software for a secure operating
system".
Proc. Int. Conf. on reliable software.
Los Angeles, April 1975, pp.267-284.
(SIGPLAN Notices 10,6).

Sat72

Satterthwaite, E.,
"Debugging tools for high level languages".
Software-Practice and Experience.
Vol.2,3, 1972, pp.197-219.

Sch73

Schwartz, M.S.,
"A storage hierarchical addressing space for a computer
file system".
Ph.D. thesis, Case Western University, U.S.A., 1973.
(also available as Jennings report 1144.)

SeL76

Severance, D.G., Lohman, G.M.,
"Differential files: their application to the maintenance
of large databases".
ACM Transactions on Database Systems, Vol.1,3,
September 1976, pp.256-267.

Sk176

Sklaroff, J.R.,
"Redundancy management technique for space
shuttle computers".
IBM Journal of Research and Development,
Vol.20,1, January 1976, pp.20-28.

Sno76

Snow, C.R.,
"An exercise in the transportation of an operating system".
Technical Report no 94, Computing Laboratory,
University of Newcastle upon Tyne, England, December 1976.

SmH72

Smith, J.L., Holden, T.S.,
"Restart of an operating system having a permanent
file structure".
The Computer Journal, Vol.15,1, 1972, pp.25-32.

StS72a

Stoy, J.E., Strachey, C.,
"OS6 - An experimental operating system for a small
computer.
Part 1: general principles and structure".
The Computer Journal, Vol.15, 1972, pp.117-124.

StS72b

Stoy, J.E., Strachey, C.,
"OS6 - An experimental operating system for a small
computer.
Part 2: input/output and filing system".
The Computer Journal, Vol.15, 1972, pp.195-201.

StS72c

Stoy, J.E., Strachey, C.,
"The text of OSPub".
Oxford University Computing Laboratory.
Programming Research Group, July 1972.
Technical Monograph PRG - g t .

StS72d

Stoy, J.E., Strachey, C.,
"The text of OSPub".
Oxford University Computing Laboratory.
Programming Research Group, July 1972.
Technical Monograph PRG - g c .

Tay76

Taylor, J.M.,
"Redundancy and recovery in the HIVE virtual machine".
Royal signal and radar establishment, Christchurch,
England, Report no. 76010, May 1976.

Tit74

Titman, P.J.,
"An experimental data base system using binary relations".
Data base management (Klimbie, J.W. and Koffeman, K.L. eds.)
North-Holland Publishing Company, 1974, pp.351-360.

Ton75

Tonik, A.B.,
"Checkpoint, restart and recovery:
selected annotated bibliography".
FDT, Bulletin of ACM - SIGMOD, Vol.7, 3-4, 1975.
pp.72-76.

Ver76

Verhofstad, J.,
"A recoverable OCODE machine, a technical note".
Internal memo SRM/158, Computing Laboratory,
University of Newcastle upon Tyne,
England, November 1976.

Ver77

Verhofstad, J.,
"Recovery and crash resistance in a filing system".
ACM SIGMOD Conf., Toronto, Canada, August 1977.

Ver77a

Verhofstad, J.,
"The costs of recoverability provided by a fault tolerant
OCODE interpreter".
Internal memo,
Computing Laboratory, University of
Newcastle upon Tyne, 1977.

Von76

Vong, Y.S.,
"A recovery cache mechanism using a highspeed buffer".
M.Sc. Thesis,
Computing Laboratory, University of
Newcastle upon Tyne, 1976.

Wen72

Wensley, J.H.,
"SIFT - software implementation fault tolerance".
AFIPS Conf. Proc. Vol.41, part I,
AFIPS Press, Montvale, New Jersey,
1972, pp243-255.

Wim71

Wimbrow, J.H.,
"A large-scale interactive administrative system".
IBM Systems Journal, Vol.10,4, 1971, pp.260-282.

WLS76

Wulf, W.A., London, R.L., Shaw, M.,
"Verification and abstraction in ALPHARD".
Carnegie-Mellon University, January 1976.

Wye73

Wyeth, D.,
"Estimates for the size of the recursive cache".
Internal memo SRM/71, Computing Laboratory,
University of Newcastle upon Tyne, England,
December 1973.