Abstract:

Backward error recovery (that is, resetting an erroneous state of a system to a previous error-free state) is an important general technique for recovery from faults in a system, especially those faults which were not foreseen. However, the provision of backward error recovery can be complex, particularly if the implementation of the system is "multi-level" and recovery is to be provided at a number of these levels. This paper discusses two distinct categories of multi-level system, and then examines in detail the issues involved in providing backward error recovery in both types of system.

A Model of Recoverability in Multi-level Systems

By

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

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 115
November, 1977

© 1977 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
ANDERSON, Thomas,
A model of recoverability in multi-level systems
Newcastle upon Tyne: University of Newcastle upon Tyne,
Computing Laboratory, 1977.
(University of Newcastle upon Tyne, Computing Laboratory,

12"
Introduction

The demand for ever more powerful, flexible and convenient interfaces to computational systems has led to the construction of increasingly complex hardware and software intended to support such interfaces. Unfortunately, the complexity inherent in the design and construction of these systems has been, and is, a major source of unreliability in their operation. Thus, attempts have been made to limit and master complexity by means of various design and implementation methodologies. In particular, approaches involving structuring a system into a hierarchy of interfaces (or levels of abstraction) are often advocated, and have been adopted in the construction of some systems, for example, the THE multiprogramming system [4] and the VENUS operating system [6]. While such "multi-level" approaches are certainly laudable, it is widely recognised that complex systems will in general always contain residual design faults; for some systems, specifically those with a high reliability requirement, there is a need for tolerance of such faults. Residual design faults are by their nature unanticipated and unanticipatable, and in consequence are particularly difficult to deal with. However, backward error recovery is an important general technique for recovery from the errors caused by such faults, and involves resetting the state of the system to a previous (and hopefully error free) state. Given this recovery capability, it may be possible to enhance the reliability of the system by invoking recovery when erroneous situations are detected.

This paper is concerned with the issues involved in the provision of backward error recovery to independent processes in multi-level systems. After identifying two distinct categories of interface support for multi-level systems, a model of the implementation of backward error recovery for each category is developed and examined in depth. Techniques for error detection and fault treatment are not covered; Randell, Lee and Treleaven [9] discuss these topics and both backward and forward error recovery in some detail.
Multi-level Systems

Many systems can be described as multi-level in that a hierarchy of abstract interfaces (or levels) can be discerned in their implementation. An abstract interface may be conveniently thought of as being characterised by a language providing objects and operations to manipulate those objects. A useful notion is that of the state of an interface, which may be regarded as the set of the current states of the objects provided by that interface.

Computational systems are usually sufficiently complex that numerous interfaces could be delineated within the implementation of such a system. However, examination of the implementation will usually enable certain significant interfaces to be identified. The most significant interfaces arising in the implementation of a system are those interfaces supported by interpretation, as described below.

\[ \begin{array}{c}
\text{program} \\
\hline
I_3 \\
\hline
\text{data} \\
\hline
L_3 \\
\hline
\text{program} \\
\hline
I_2 \\
\hline
\text{data} \\
\hline
L_2 \\
\hline
\text{program} \\
\hline
I_1 \\
\hline
\text{data} \\
\hline
L_1 \\
\hline
\text{program} \\
\hline
I_0 \\
\hline
\text{data} \\
\hline
L_0 \\
\end{array} \]

Figure 1. Interpretive Multi-level System

In figure 1, each interface $L_i$ is implemented by means of a program $I_i$ which is executed on the interface $L_{i-1}$. Every interaction with the interface $L_i$ (corresponding to the execution of an operation in the program $I_{i+1}$) is in fact directly supported by means of the program $I_i$. Any "abstract" object available in $L_i$ has a "concrete" representation as a set of objects in $L_{i-1}$ which are managed by $I_i$.
and held in a data area maintained for this purpose by Ii. The program Ii is referred to as an interpreter for Li.

Example:

An APL interpreter is a program which, after loading, is executed on an interface characterised by a machine language and supports an interface characterised by (the internal representation of) APL source code.

A further category of interfaces is important in the implementation of multi-level systems and usually occurs in the following circumstances. If the implementer of a system is presented with an interface Li-1 and wishes to provide an interface Li such that Li and Li-1 have many behavioural properties in common, then it may not be necessary to support Li by means of another level of interpretation. If Li-1 makes available sufficiently powerful extension facilities it will be possible to provide Li by generating an extension of Li-1. Run time subroutine or procedure mechanisms can be regarded as a commonly available, though limited, interpreter extension facility, whereby new operators can be built up as programmed sequences of those originally provided. More powerful mechanisms could allow the addition of new types and even notations, as well as permitting the removal of features of the original interface. Ideally, the extension facilities would still be available in Li so that further extensions could be made if required.

Note that the above discussion refers to extending an interface at run time; indeed, all of the interfaces considered in this paper should be regarded as existing at run time, and not merely present in a source program before perhaps being thrown away during compilation.

<table>
<thead>
<tr>
<th>program</th>
<th>program</th>
<th>program</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>E2</td>
<td>E3</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
<td>data</td>
</tr>
<tr>
<td>L1</td>
<td>L2</td>
<td>L3</td>
</tr>
<tr>
<td>I10</td>
<td>I10</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Extended Interpreter Multi-level System
In figure 2 the interface $I_\emptyset$ is supported by an interpreter $I_\emptyset$ which provides extension facilities. Each interface $L_i$ ($i = 1, 2, 3$) is constructed as an extension of the interface $L_{i-1}$, the extension being implemented by means of a program $E_i$ which is executed on the interface $L_{i-1}$. The program $E_i$ is referred to here as an interpreter extension. Every interaction with the interface $L_i$ is first examined by the underlying interpreter $I_\emptyset$, which determines whether that interaction is directly supported by $I_\emptyset$ itself or, if not, which of the available interpreter extensions ($E_j$, $j \leq i$) does support that interaction. Thus, the interactions of a program may be supported by any lower extensions or by the underlying interpreter. Any "abstract" object available in $L_i$, other than those directly supported by $I_\emptyset$, has a "concrete" representation as a set of objects in one of the interfaces $L_j$ ($j < i$). This set of objects is managed by $E_{j+1}$ in a data area maintained for this purpose by $E_{j+1}$.

The extended interpreter multi-level system of figure 2 is portrayed differently to the interpretive system of figure 1 with the intention of

(i) indicating diagramatically the anticipated overlap of behavioural properties between an interface $L_i$ and those beneath it ($I_\emptyset$, ... , $L_{i-1}$) in an extended interpreter system, and

(ii) emphasising the differing mechanisms used to implement the interfaces of the two systems; in figure 2 the interpreter $I_\emptyset$ has responsibility for programs executed on all interfaces $L_i$ (with assistance from the extensions as required), whereas in figure 1 each interpreter only has responsibility for the interface directly above.

Example:

The nucleus of most operating systems can be regarded as an interpreter extension which provides a user interface from the underlying hardware interface by removing certain privileged instructions and adding a set of "operating system" call instructions. A collection of system procedures often provides a further extension.
Specifically, the programming language Concurrent Pascal [3] provides a facility whereby an operating system (written in Concurrent Pascal) can make available procedures that can be invoked by user programs (written in Sequential Pascal). The Concurrent and Sequential Pascal programs are both executed by the same underlying interpreter, and the procedures which are made available to the user programs can be regarded as interpreter extensions.

The potential advantage to be gained from using an interpreter extension to implement a new interface (when this is possible) in preference to a further level of interpretation is in avoiding the overhead that the latter entails. Whenever an interaction on the new interface could be directly supported by the underlying interpreter, an interpreter extension implementation will ensure that this direct support is available. Both techniques can, of course, be used in the same multi-level system; for example, in figure 1 an interpreter II could be implemented by first extending Li-1 to a more convenient interface before constructing an interpreter for Li. However, it should be noted that to a program being executed on an interface it is completely immaterial whether that interface is implemented by an interpreter extension or not - this can only be determined from an examination of the details of the implementation of an interface, and not merely from the properties of the interface itself.

Recovery on an Interface

Before the problems of recovery in multi-level systems can be considered, it is necessary to introduce terminology and discuss the recovery of a program at one level in a system.

In this paper, recoverability is taken to mean the ability to recover an earlier state of an interface, thereby undoing the effects of operations that were performed on the interface. This ability is referred to as backward error recovery; backward error recovery necessitates the recording of recovery data, which can be used to restore an earlier state of the interface.
To be more specific, this paper considers recovery mechanisms which make the following features available to programs executed on an interface:-

(i) the interface supports both recoverable and unrecoverable objects

(ii) recovery points can be established, which ensure that the current state of the recoverable objects of the interface is (at least conceptually) recorded so that it can be restored if necessary

(iii) recovery points can be discarded with the effect that information maintained for recovery to those recovery points is discarded.

Clearly, a recoverable object is one for which recovery is provided. In contrast, unrecoverable objects are objects for which state restoration is not available or appropriate for recovery. As such, unrecoverable objects can also be used to model the effects of the external environment, for example, process intercommunication. Verhofstad [11] has discussed the provision and uses of both recoverable and unrecoverable objects.

A recovery point is said to be active from when it is established until it is discarded. The term recovery region will be used to refer to the period for which a recovery point is active (this will usually correspond to a section of the text of a program). When an active recovery point is discarded, commitment occurs in that the associated recovery data is also discarded and in consequence recovery to that recovery point is no longer available.

A further useful notion is that of the recovery environment associated with a recovery point. The recovery environment is that set of recoverable objects which are available on the interface at the time a recovery point is established. In a program the recovery environment would consist of the set of recoverable variables that were in existence at the time the recovery point was established.
It is assumed that in a general recoverable system a program can have more than one active recovery point. Thus the recovery regions of a program may partially or totally overlap. Figure 3(a) illustrates a program with partially overlapping recovery regions; figure 3(b) illustrates a program with totally overlapped regions.

![Diagram showing multiple recovery points](image)

(a) recovery point 1
(b) recovery point 1
recovery region 1
recovery region 2
recovery point 2
discard recovery point 1
discard recovery point 2

time

Figure 3. Multiple Recovery Points

Although the use of partially overlapping recovery regions is not precluded in this paper, it may well be considered preferable in a system only to permit totally overlapping recovery regions, since the arguments for a "good" recovery structure may be similar to those advocated elsewhere for the control structures in programs.

Example:

A programming language construct called the recovery block has been introduced by Horning et al [5], which enables a program to establish recovery points, while constraining recovery regions to be properly nested.
It is possible that although two recovery points are simultaneously active, their respective recovery environments may have no objects in common with each other. Such recovery environments are said to be disjoint. The notion of disjoint recovery environments has little significance when recovery of a program on a single interface is considered but will be returned to when recovery in multi-level systems is described.

The above discussion has been concerned with the recoverable objects of a program. The question remains as to what happens to any unrecoverable objects used by a program within a recovery region? Clearly, the answer is that no recovery data will be generated for such objects; consequently, if recovery is invoked, the current state of the unrecoverable objects will prevail. If in these circumstances some form of restoration of the unrecoverable objects is required then the program would have to perform this restoration itself. It would, therefore, have to record some information specifically for this purpose. The terminology introduced by Verhofstad [11] refers to the data structure used to hold this information as the log. Since the log contains information to be used by the program for recovery, this information must not be lost when recovery is invoked.

Having discussed recovery on a single interface, and distinguished between interpreters and interpreter extensions as techniques for supporting the interfaces of a multi-level system, it is now possible to consider the problems of providing recoverability to such interfaces.

**Recovery in Interpretive Multi-level Systems**

The first problem to be considered is that of providing recoverability to an interface completely supported by an interpreter. If the recovery features identified above are to be supported, then the interpreter must include programs and their data structures such that all information necessary for the recoverability of that interface is maintained. (Clearly this is true for any of the features supported by the interpreter.) Those programs which are concerned
with providing recoverability will be termed recovery programs.

There will, in general, be three distinct recovery program parts, concerned with the following actions:

(i) recording recovery data
(ii) performing recovery
(iii) performing commitment.

There are several strategies that could be adopted for the recording of recovery data. The simplest method is that of recording a complete checkpoint, in which the state of all of the objects within the current recovery environment is recorded when a recovery point is established. Although this is the simplest approach, it has the disadvantage that the state of all of the objects in the current recovery environment has to be recorded, despite the possibility that many of those objects might not be changed. This disadvantage is often mitigated in practical systems by only recording recovery data for those objects that are updated; for example, the COPRA system [7] attempts to assess in advance which objects fall into this category. Alternatively, recovery data can be recorded dynamically (that is, the state of an object is saved just before that object is updated) in what may be termed an incremental checkpoint.

Example:

A highly optimised technique for recording recovery data (in conjunction with recovery blocks) was proposed by Horning et al [5]. This technique, called the recursive/recovery cache, consisted of recording incremental checkpoints in such a way that a minimum of recovery data was maintained.

Audit trails [2] are a further strategy for providing recoverability, where the recovery data essentially provides a record of the operations that were performed on the objects.
Whatever technique is employed for recording recovery data it must only ensure that recovery and commitment can be performed as required. The concepts discussed in this paper are independent of the strategy adopted for recording and committing recovery data.

In figure 4, L1 is a recoverable interface supported by an interpreter I1 which contains recovery programs and their data structures. The mapping of the data space of I2 to its concrete representation in that of I1 is also indicated.

![Diagram of recovery structures in an interpreter](attachment:image)

Figure 4. Recovery Structures in an Interpreter

Changes to the objects in the data space of I2 are implemented (by I1) as changes to their concrete representations in the data space of I1. It is the responsibility of the recovery program of I1 to ensure that appropriate information is recorded as recovery data. Should it be necessary to restore the objects of I2, the recovery program of I1 uses this recovery data to update the concrete representations such that the objects of I2 have the appearance of having been restored. It may also be observed that backward error recovery is provided to I2 by means of normal (forward) computation of the recovery program part of I1, as noted by Randell [8].
It is important to observe that any recovery environment on L1 is disjoint from those on L0. Thus, an active recovery point at one level is completely independent of those in other levels. (This point will be returned to subsequently when recovery in extended interpreter systems is discussed.) It should be clear that the provision of recoverability in L1 does not imply that higher interfaces are also recoverable. For L2 to be recoverable, I2 would have to include recovery program and data to support that recoverability, just as I1 does to support the recoverability of L1. Use that I2 makes of the recovery features of L1 in order to provide a more reliable mode of support to L2 is completely independent of whether or not I2 itself provides recovery features in L2. (It may be possible for an interpreter to use the recovery facilities provided to it to achieve a more direct implementation of recoverability for the interface it supports. However, the implications and implementation of such a scheme are not at all straightforward.)

For I1 to provide unrecoverable objects in L1 is very simple. For these objects I1 records no recovery data. The only impact that the presence of unrecoverable objects in L1 has on I2 is that if such objects are used within recovery regions by I2 then obviously their prior values could not be restored in the event of recovery actually being invoked. Consequently, I2 will have to log its own data for recovery, as discussed in the previous section.

Example:

The EML system described by Anderson and Kerr [1] has exactly the structure shown in figure 4. In this system, the underlying interface L0 was that provided by (PDP-11) hardware, while the interface L1 was supported by an interpreter which emulated a high-level abstract machine architecture. The interpreter implemented a recovery cache mechanism, enabling programs executing on L1 to make use of recovery blocks.

In summary, the recoverability of a hierarchy of interfaces in an interpretive system is straightforward in concept.
Recovery in Extended Interpreter Multi-level Systems

This section considers the problems of recovery in a multi-level system implemented by a sequence of interpreter extensions. It is assumed that the underlying interface presented to the interpreter extensions (for example, L∅ in figure 5) supports both recoverable and unrecoverable objects, and permits recovery points to be established and discarded.

The principal objective of an interpreter extension is, as its name suggests, to extend an interface with new kinds of abstract object. Clearly, it is desirable for an extension to also provide recovery features for these abstract objects; it should extend the recovery features of the existing interface to include these objects. The issues concerning the provision of recoverability for these new abstract objects form the subject of this section.

![Diagram](attachment:figure_5.png)

**Figure 5. Invocation of an Interpreter Extension**

Consider the situation depicted in figure 5, which will be used as an example in the following discussions. This system has an interpreter extension E1 which is (indirectly) invoked by a program E2. Both of these programs are interpreted by L∅, the underlying interpreter. The calling program has established a recovery point, and subsequently invoked an operation supported by E1. To E2, this operation appears to be indivisible, as do any of the operations interpreted directly by L∅. (Indeed, E2 should not be aware of the distinction between an operation supported by L∅ and one supported by E1.)
In this example it is assumed that all objects in the abstract data space of E2 are recoverable, and that some of these are maintained by the extension E1 while the rest are maintained by the interpreter IΩ. This is indicated in figure 6 by dividing A2, the abstract data space of E2, into two parts; these two parts have concrete representations A1 (in E1) and C2 (in IΩ) respectively. In fact, A1 is in the abstract data space of E1 and has its own separate concrete representation C1, which in this example is maintained solely by IΩ.

Figure 6. Abstract Data Space Mappings

At some time following the return from the extension program E1, an error has been detected in E2 (figure 5); consequently E2 has to be backed up to the recovery point it had previously established. In order to achieve this backup, the recoverable objects used by E2 must be restored to their previous (abstract) state.

Certainly, IΩ can restore those objects it is directly maintaining on behalf of E2 (i.e. those whose concrete representation is C2 in figure 6). The question arises as to how the abstraction of recoverability is provided for the objects used by E2 which are supported by the extension E1?

Consider first the situation in which this support is achieved through the use of unrecoverable objects (i.e. A1 in figure 6 contains unrecoverable objects only). In this situation it will be necessary
for the extension to provide both recovery programs and data (as shown in figure 7) so that it can restore the abstract state of the relevant E2 objects. These recovery programs will need to be automatically invoked (by Iφ) as required. For example, in a complete checkpointing scheme, when the calling program establishes a recovery point, Iφ would record a checkpoint and then invoke the recovery program of the extension so that it could also record a checkpoint of those objects it was maintaining for the program.

```
program       recovery program
E1            data           E2
              recovery data
```

Figure 7. Recovery Structures in an Interpreter Extension

In a more general situation, an extension may well make use of both unrecoverable and recoverable objects for its implementation of new recoverable objects. A second question therefore follows, namely, should the recoverable objects used by the extension be regarded as being within a recovery environment of the calling program, and in consequence should they be included in the recovery data recorded by the underlying interpreter for the calling program? By analogy with the interpretive multi-level system discussed in the previous section the answer would be no - in an interpretive system the recovery environment of one level is completely independent of that of the level above. If this view is adopted for extended interpreter systems, the recoverable objects used by an extension should not be regarded as being within a recovery environment of the calling program. An extension would therefore be completely responsible for all of the recovery of the objects it was maintaining. Indeed, this would be the behaviour
expected if the underlying interpreter itself provided the features of the extension. To obtain this behaviour in an extended interpreter system it is only necessary to stipulate that the recovery environments of a program be disjoint from those of any supporting extensions. The scheme of recovery which exhibits this property in an extended interpreter system is referred to as the disjoint recovery scheme. When a program establishes a recovery point, the disjoint recovery scheme must ensure that the recovery environment of that recovery point only encompasses the abstract objects available on the interface, and not any of the objects used to implement those abstract objects. Thus, for example, referring to figures 5 and 6, when E2 establishes a recovery point, IQ will only record recovery data for those recoverable objects represented in C2, and not for any of those represented in C1. (E1 will of course be invoked to record recovery data for objects it is maintaining for E2.)

However, the disjoint recovery scheme does not prevent an extension from establishing its own recovery points. In this situation, any recoverable objects used by the extension within a recovery region would behave normally. When the local recovery point was discarded, all of the recovery data being maintained for that recovery point would be discarded.

Generalising, the recovery in a multi-level system implemented by interpreter extensions with the disjoint recovery scheme would be as follows. Following the detection of an error in program Ei, the underlying interpreter would restore all of the recoverable objects that it was directly maintaining for Ei. The interpreter would then signal all of the extensions that could be called directly by Ei (in the set Ei, ..., Ei-1) so that they could perform recovery for any objects they were directly maintaining for Ei. Following the completion of these actions the program Ei will have been recovered and can be restarted as necessary. Conceptually, the interpreter also has to signal all of the directly accessible extensions in Ei, ..., Ei-1 whenever program Ei creates or discards a recovery point, so that the
extensions can, in a manner similar to that of the underlying interpreter, record or commit the necessary recovery data for the program $E_i$. Optimisations of this conceptual organisation are clearly possible.

A significant characteristic of the disjoint recovery scheme is that the behaviour of an extension with respect to both recoverable and unrecoverable objects is uniform, in so far as the state restoration of the new abstract objects is concerned. The scheme also models the recovery behaviour of the well understood multi-level interpreter system.

However, the scheme does have a disadvantage which becomes apparent if the recoverable objects used in an extension are reconsidered. As discussed previously, one of the main aims of an interpreter extension is to extend a given interface without incurring the inefficiency of re-implementing all of the features it did not wish to change (as would happen in an interpreter system). However, as far as the recoverable objects used in the disjoint scheme are concerned, the extension will have to re-implement some form of recovery for these recoverable objects, even though that supplied by the underlying interface may have been exactly what was required. For example, if the interface $IØ$ in figure 6 provided recoverable objects only, the extension $E1$ would still have to re-implement recoverability for $E2$ even though restoration of $A1$ by $IØ$ is equivalent to the restoration of the corresponding objects in $A2$.

It is this apparent inefficiency which leads to an alternative answer to the question concerning the behaviour of the recoverable objects used in an extension, namely, that the recoverable objects used in an extension are regarded as being within the recovery environment of the calling program, and are therefore automatically restored when the calling program is backed up. This recovery scheme is called the inclusive recovery scheme.
An example of the behaviour of the inclusive recovery scheme can be obtained by reconsidering the system depicted in figures 5 and 6. The recoverable objects in A1 (figure 5) are then regarded as being within the recovery region of the calling program E2. Consequently, when E2 establishes a recovery point, any recovery data generated for the recoverable A1 objects will be maintained with the recovery data associated with E2. (Of course, the extension still has local recovery features available to it.) When the calling program E2 is backed up, the extension E1 can assume, when it is invoked by IØ, that any recoverable objects used by it will have been automatically restored; to maintain its abstractions, E1 therefore needs only to change, as necessary, the unrecoverable objects used in the concrete state. As in the disjoint case, the underlying interpreter IØ will have to signal the extension to obtain this recovery.

If the recoverable objects used by an extension were all objects supported directly by the underlying interpreter (as was the case in figure 6), then only this signalling would be required. However, in general some of the recoverable objects used by an extension may themselves be implemented by lower extensions (i.e. to the "left") as indicated in figure 8.

---

**Figure 8. Multi-level Abstract Data Space Mappings**
If E3 were backed up, then E2 would be signalled to perform any recovery actions on the uncoverable objects it used on behalf of E3. However, E2 would expect all of its recoverable objects to be recovered automatically, although some of them had been implemented by E1.

Clearly, therefore, when a program E1 is backed up, all of the lower extensions E1, ..., Ei-1 will be required to return the abstract objects they are maintaining to their prior states, and the underlying interpreter IØ must ensure that this occurs, signalling all of the relevant extensions until all of these restorations have been completed. At this point, execution of the program Ei can be resumed. In figure 8, recovery of E3 would result in the recovery program of both E2 and E1 being invoked by IØ - in the disjoint recovery scheme discussed above, E3 being recovered would not result in the recovery program of E1 being invoked by IØ (unless of course E3 had directly invoked E1). As before, the underlying interpreter will (conceptually) have to signal the extensions E1, ..., Ei-1 whenever a program Ei establishes or discards a recovery point.

The signalling of extensions in a "left" to "right" order (i.e. least to most abstract) seems to be the most natural order. Certainly, the interpreter IØ will be aware of this order, for each extension would have to be identified to the interpreter when it was created, so that it could be subsequently invoked. In fact, the ordering of the signalling is only significant in two somewhat improbable cases: firstly, if the unrestored values of recoverable objects were of interest to the recovery implementation of an extension; and secondly, if the extension wished to update a recovered object. In the disjoint recovery scheme discussed previously, the signalling order would have no effect as each extension is responsible for all of its recovery, and does not depend on any lower extensions to achieve this automatically.
The main advantage of the inclusive recovery scheme is that mentioned above, namely that an extension can rely on the automatic recovery of any recoverable objects that it uses - an extension which only used recoverable objects would not therefore have to provide any programs or data for recovery purposes. Note that, unlike the disjoint scheme, the behaviour of the extension with respect to its recoverable and unrecoverable objects is not uniform.

An important disadvantage of the inclusive recovery scheme stems from the fact that it is difficult (although not impossible) for this scheme to model the behaviour of the disjoint scheme; one specific situation in which it is incorrect for the inclusive recovery rules to apply is when the extension is executing the recovery program parts discussed previously. Although these program parts are executed by an extension on behalf of a higher level program, their recovery environments must be regarded as being local to the extension (that is, disjoint from recovery environments of the higher level program) in contrast to the normal situation for the inclusive scheme. The need for this behaviour can be illustrated by considering that part of the recovery program which records recovery data. If the data structures used to hold this recovery data were taken to be within the recovery environment of the calling program then any backing up of that program would result in those data structures being automatically recovered, thus erasing the recovery data stored by the extension.

The desired behaviour could be achieved in two ways: firstly, by the extension (or the underlying interpreter) ensuring that the recovery program parts of an extension did not make use of any recoverable objects that were within the recovery environment of the calling program (for example, by constructing the recovery data structures from unrecoverable objects); and secondly, and more generally, by the underlying interpreter being able to distinguish between the programs and recovery programs of an extension, and thus determining when to apply the disjoint recovery rules. Note that these problems do not arise in the disjoint recovery scheme since all programs (including recovery programs) exhibit the required behaviour.
It would appear, therefore, that an implementation of the inclusive recovery scheme would also be required to provide the features of the disjoint scheme, at least for use as described above. Given this requirement, the availability of these features of the disjoint scheme could be extended to allow them to be used, as required, by any extension, for there may be other situations in which the inclusive recovery rules are not appropriate or convenient.

Conclusions

This paper has investigated the issues involved in the provision of recoverability in multi-level systems implemented both by interpreters and by interpreter extensions. In particular, recoverability in extended interpreter systems has been examined in detail, and two recovery schemes (disjoint and inclusive) described. The disjoint recovery scheme models the recovery behaviour which is obtained in a multi-level interpretive system. As such it shares advantages and disadvantages of interpretive systems. These are, respectively, conceptual simplicity, and the inability to automatically inherit and make use of the recoverability of lower level objects. To avoid this disadvantage, the inclusive recovery scheme was suggested. However, it was shown that this scheme needs the features of the disjoint scheme for use in the recovery program parts of extensions.

It seems appropriate, therefore, that an underlying interpreter should be able to support both of these recovery schemes for the efficient implementation of recovery in multi-level systems.

It is of interest to relate the recovery schemes discussed in this paper to two experimental multi-level recoverable systems using interpreter extensions that have been (or are being) implemented at the University of Newcastle upon Tyne, since these motivated our investigations. In one experimental system [11] a first extension implements a recoverable single user filing system; subsequent extensions can be built on top of this filing system. This experimental system implements the inclusive recovery scheme, and relies
on programmer discipline to ensure that the recovery program parts of
the extensions do not generate any recovery information. Disjoint
recovery is not available to the implementers of extensions.

The other recoverable system, currently being implemented, uses
the disjoint recovery scheme to provide for recoverable resource
allocation between many competing processes [10]. This system
provides a facility whereby a program can create recoverable resource
objects (called ports); creation of such an object is equivalent to
establishing a new extension. The scope rules of ports ensure that
the recovery environment of a port is disjoint from that of the
creating program. Inclusive recovery is not available to the imple-
menters of ports.

Experimentation with the above systems should shed further light
on the adequacy of the two recovery schemes for implementing recover-
able multi-level systems by means of interpreter extensions. Work is
in progress to examine appropriate architectures for implementing
recoverable multi-level systems, and to consider the issues raised by
the introduction of shared recoverable objects.

Acknowledgement

We would like to express our gratitude to our fellow members of
the Reliability Project, sponsored by the Science Research Council of
Great Britain, at the University of Newcastle upon Tyne. In particular,
we wish to acknowledge the many discussions that took place between the
authors and Joost Verhofstad in the initial stages of the preparation
of this paper.
References


