Exploiting type inheritance facilities to implement recoverability in object based systems

G.N. Dixon
S.K. Shrivastava

Abstract

One of the key concepts available in many object based programming languages is that of type inheritance, permitting new object types to be refined out of existing object types. This paper discusses how this concept can be exploited for introducing recoverability into a system. A multilevel object based recovery model is employed, permitting recoverable objects to be constructed out of recoverable and unrecoverable objects. Simple examples are used to illustrate the ideas and to demonstrate the suitability of the proposed approach.

Series Editor: M.J. Elphick

© 1986 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.
Bibliographical details

DIXON, Graeme Neville


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1986.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 223.)

Added entries

SHRVASTAVA, Santosh Kumar.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

One of the key concepts available in many object based programming languages is that of type inheritance, permitting new object types to be refined out of existing object types. This paper discusses how this concept can be exploited for introducing recoverability into a system. A multilevel object based recovery model is employed, permitting recoverable objects to be constructed out of recoverable and unrecoverable objects. Simple examples are used to illustrate the ideas and to demonstrate the suitability of the proposed approach.

About the author

Mr. G.N. Dixon is a Research Associate in the Computing Laboratory.

Professor S.K. Shrivastava joined the Computing Laboratory in August, 1975 where he is now a professor.

Suggested keywords

BACKWARD ERROR RECOVERY
FAULT TOLERANCE
OBJECT BASED PROGRAMMING
TYPE INHERITANCE

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 001.6424
U.D.C. 681.322.06
1. Introduction

Object based programming languages usually support the facility of data abstraction, enabling a programmer to associate a set of operations with data structures. The term abstract objects (or objects for short) is used to refer to instances of data abstractions. Abstract objects are structured entities that can only be manipulated by invoking the operations associated with them. The term object manager will be used to refer to the provider of the corresponding abstract object. Programs constructed using abstract objects usually have a hierarchical structure in the sense that higher level objects are constructed using lower level objects. Making such a program fault tolerant essentially involves making objects fault tolerant. In other words, it will be the task of object managers to employ whatever fault tolerance techniques are necessary to provide (or maintain) reliable objects. A widely employed control abstraction in distributed systems is that of an atomic action (atomic transaction) with the failure atomicity property: a computation structured as an atomic action can be aborted without producing any effects. To achieve this property, we require that fault tolerant objects be made recoverable, such that whenever an action is to be aborted, the relevant object managers can recover the states of the objects to those at the start of the action.

The development of distributed systems supporting atomic actions and recoverable objects is an active area of research. Various language features have been proposed (e.g. [1]) to express the recoverability properties of objects. Rather surprisingly, one of the key concepts of many object based languages - that of type inheritance - has been ignored by researchers investigating ways of incorporating recoverability in objects. Simply stated, a type inheritance facility - pioneered in the language SIMULA [2] - enables a programmer to define a new object by refining an existing object definition such that an instance of the new object inherits some or all of the properties of the existing base object. Suppose that we have constructed a base object definition that provides rudimentary recovery facilities. Higher level objects may now inherit these recovery facilities to provide recoverability. This paper explores this very simple idea and demonstrates that the type inheritance concept does indeed provide a powerful basis for introducing recoverability into object based systems. Given that objects can be constructed out of other objects, it is natural to assume that one should be able to construct recoverable objects out of
existing recoverable and unrecoverable objects. We show how this *multilevel recovery* can be implemented.

The rest of the paper is structured as follows. In the next section we review the basic ideas of multilevel object based recovery; the following section (section 3) presents the type inheritance facilities of a widely used object based language C++. Section 4 contains a number of simple examples to illustrate how these facilities can be exploited to construct recoverable objects. The last section contains the conclusions from our work.

The ideas we wish to put forward can be explained quite effectively by considering sequential programs, although extensions using concurrency control techniques applicable to atomic actions are clearly possible.

2. **Principles of Multilevel Recovery**

We will assume that a process can *establish a recovery point* (operation *erp*), thus indicating the start of a *recovery region* (see Figure 1). This implies that, if necessary, the states of any

![Diagram of Recovery Regions](image)

*Figure 1: Recovery Regions*

*recoverable objects* modified in that region can be restored automatically to those at the start of the region. A process can establish multiple extant recovery points, giving rise to *nested* recovery
regions. A process can also discard a previously established recovery point (operation drp) to indicate the end of a recovery region. To invoke backward recovery a process is given a restore primitive.

By definition, a recoverable object has the property that if its operations are invoked from within a recovery region, then the invocation of restore by the calling process will have the object restored to the state that prevailed at the start of the recovery region. In particular, we will assume that even if several operations on a recoverable object have been invoked from within a recovery region, a single restore will have the desired effect - it is the responsibility of the object manager to take whatever steps are necessary to perform state restoration. Note that if operations on recoverable objects are invoked from outside a recovery region, then no automatic recovery will be available.

![Diagram](image)

Figure 2: Managing Recovery Data in the P_list for nested regions

A simple way of mechanising recovery is as follows: every recoverable object provides an operation named restore, whose function is to perform state restoration for that object. With a process we associate a special data structure (called the P_list) that records the names of the recoverable objects updated within each recovery region. The P_list object provides a restore
operation which may be invoked by its process to in turn invoke the restore operations for all the objects whose names are recorded in the relevant recovery region of the P_list. Consider the situation depicted in Figure 2(a) (where we have numbered erps and drps to indicate recovery regions). Assume that the flow of control has reached the point shown by the first arrow; then the P_list of the process will contain the names of objects invoked from within the two recovery regions as depicted in Figure 2(b). Once drp2 is executed, region 2 has to be discarded. The names of objects recorded in the region to be discarded are merged with the enclosing region as depicted in Figure 2(c), which shows the state of the list when the flow of control is at the point indicated by the second arrow. This merging will ensure that if recovery is invoked at the point indicated by the second arrow, the states of all three objects will be restored.

Consider the situation where a process has invoked an operation OP1 on a recoverable object A (Figure 3) and the object A is constructed out of two recoverable objects (x, y) and an unrecoverable object (z). The body of the operation OP1 is also depicted in the figure. We next ask the question: when the body of the operation is executed, should that execution be regarded as taking place from within the recovery region of the calling process? If the answer to the question is yes then the P_list of the process will be shown as in the figure (under the label inclusive

```
Process
    └── erp1
        └── A.OP1(...)

operation OP1 of object A
OP1(...)  
    /* operation body */
    ....
        operations on x, y and z
    ....

Figure 3: Illustrating Inclusive and Disjoint Recovery
```
recovery). Since objects x and y are recoverable, their names, in addition to that of A will be recorded in the list. So, when recovery is invoked, objects x and y will be restored and the restore operation of A need only be concerned with explicitly restoring the state of object z. Following Anderson et al, we will say that this way of managing the P_list has given us inclusive recovery [4]. On the other hand, if the answer to our question is no, we would obtain what is known as disjoint recovery, in which case the P_list would only contain the name of object A (see the figure). Thus, the restore function of A may have to explicitly restore objects x, y and z. What if the body of OP1 contains a recovery region? For the inclusive recovery scheme, this region will be regarded as nested within that of the calling process (and the P_list will be managed as discussed before for nested regions). For the disjoint recovery scheme, the recovery region of OP1 will be totally separate (disjoint) from the recovery region of the calling process (in particular, no merging will take place when the recovery region of OP1 is discarded).

The concept of inclusive and disjoint recovery is crucial when considering the construction of recoverable objects out of existing - recoverable and unrecoverable - objects. Inclusive recovery, as the previous example demonstrated, can provide a simpler means of constructing recoverable objects. In particular, if object z is recoverable as well, then there is no need for object A to provide a restore operation. On the other hand, the use of disjoint recovery means that the recoverability of objects such as x and y cannot be passed on to higher levels. There are cases where the use of disjoint recovery is desirable: for instance, if it is desired to provide a new form of recovery for A. We will show a concrete example in a subsequent section to illustrate this point further, but will assume the provision of inclusive recovery in the rest of the paper unless otherwise stated. However, we will also provide a pair of primitives - new_region and end_region - to explicitly indicate the start and end of a disjoint region with the property that any recovery region created within this new region is not nested within the recovery region (if any) of the calling process. This provides a simple means of introducing disjoint recovery in a system that supports inclusive recovery. Thus, if the body of OP1 (Figure 3) begins with new_region and ends with end_region, then we will get disjoint recovery, the P_list will then only contain the name of object A.

The details presented for the management of the P_list for nested recovery regions are derived from the elegant recovery cache algorithm of recovery blocks [3]. The discussion of
inclusive and disjoint recovery is from the paper [4] where more details of the two recovery schemes can be found. The recovery model presented here can be extended to distributed systems supporting crash recovery and atomic actions as discussed in [5,6].

3. Type Inheritance in the language C++

The language we have chosen to illustrate and implement our ideas is C++ [7], largely because of its ease of availability and its incorporation of most of the features we require for implementing multilevel recovery. C++ is a superset of the language C [8], adding facilities for data abstraction, type inheritance and other features such as operator overloading. The data abstraction and type inheritance facilities were inspired by the SIMULA programming language [2] and are based on the class concept. Instances of classes are objects, with specific operations provided for their manipulation. The type inheritance mechanism of C++ works as follows: given a base class C1, another class C2 - a derived class of C1 - can be defined so that it inherits some or all of the operations of C1.

Classes are defined in the manner shown in Figure 4(a) which is a skeleton declaration of a

class base
{
    ....
    int val1;
    int val2;
    op1();

    public:
    base();
    "base();
    ....
    op2();
    op3();
};

(a)

class derived : base
{
    ....
    int val3;

    public:
    derived();
    "derived();
    ....
    dop4();
    dop5();
};

(b)

Figure 4: C++ syntax

class called base. The variables and functions before the public label are private members of the class; the only operations which may access private variables or invoke private operations are the member operations of the class itself (base, "base, op1, op2 and op3). The variables and operations following the public label constitute the interface to objects of the class (op2 and op3 in Figure 4(a); operations base and "base are special operations, see below). An example of a class derived from the base class is shown in Figure 4(b). This new class, called derived, inherits the operations
op2 and op3. The inherited operations may only be invoked by the operations defined within the derived class.

Each class may have a constructor which is a public operation with the same name as the class (base () and derived ()), which will be invoked each time an instance of the class is created. There is also a complementary operation (~base () and ~derived ()), called a destructor, which is invoked automatically when the object is deleted. The constructor allows an object to perform class specific initialisation, such as creating objects contained within the object itself, and the destructor enables an object to tidy up before it is deleted, for example by closing any files. Both operations are special in that they will be automatically invoked when objects are created or deleted and yet as a part of the public interface to the object, they cannot be directly invoked by a user of the object.

4. Examples of Multilevel Recovery

In this section we will show how type inheritance facilities can be exploited for constructing recoverable objects. As stated earlier, we will assume that the default recovery mode is that of inclusive recovery, with the primitives new_region and end_region used to explicitly delineate disjoint regions. We will start with a base class stable which provides operations save and restore for, respectively, saving object images on a data structure called log and restoring previously saved images from the log. We will next construct recoverable integers by defining a derived class that provides recoverable integers by inheriting the recovery facilities of the base class stable. Finally we will discuss how more complex recoverable objects - in particular recoverable arrays - can be constructed.

Despite the fact that our examples are very simple, they do illustrate that the object based recovery technique which we are proposing provides a very flexible approach for constructing a variety of recovery strategies for objects. To stress this point further, we will provide one more example of a simple recoverable file system constructed using the base class stable. What follows is a simplified description of the various C++ classes we have implemented and tested.
4.1 Class stable

This is the base class which provides operations for saving and restoring object images to and from an object called log (a file). The operations save and restore must be capable ofperforming the corresponding actions on any object derived from stable. This functionality is achieved as follows. For each C++ object, the C++ compiler allocates a structure to hold the object's variables and a pointer to this structure called this. This pointer may be used by operations in both the base class and any derived class. The size of the structure can be found from this pointer by using sizeof(*this). The constructor of the derived class can use the sizeof operation in this way to pass the size of the object to the constructor of the base class, enabling the save operation to allocate appropriate storage space on the log to copy the object's structure (see also the example in section 4.3). The restore operation performs the complementary action of copying back the image from the log into the structure pointed to by this. The log employs a number of techniques for quickly locating object images which, for the sake of brevity, are not discussed here.

4.2 Class P_list

A P_list object is associated with a process for recording recovery data as indicated previously. If a process executes the operation new_region, then a new instance of P_list is created for the process (thereby creating a disjoint region). The P_list provides the primitives for backward recovery:

```cpp
class P_list
{
    /* private class variables */

    public:
    check (...);
    erp ();
    drp ();
    restore 0;
    new_region 0;
    end_region 0;
};
```
The operation *erp* inserts a region separator in the P_list to indicate the start of a new recovery region, while the operation *check* searches the current recovery region of the P_list to look for an entry for the object whose address is passed as a parameter. If no entry is found, a new entry is made in the P_list which, in addition to the object's address, contains the addresses of the object's *save* and *restore* operations, which are also passed as parameters to *check*. After making an entry in the P_list, *check* also invokes the object's *save* operation to save its image. Note that if *check* is called when the process is not in a recovery region then nothing will happen. The *restore* operation of the P_list sequentially invokes the *restore* operations for the objects whose addresses are recorded in the current recovery region. Assuming that objects inherit *save* and *restore* operations from their base class *stable*, object images will be copied on to the log and copied back from the log when the *check* and *restore* operations of the P_list are invoked respectively. The operation *drp* performs the discard and merge operations stated earlier while the operation *end_region* deletes the P_list created by the corresponding *new_region* operation.

### 4.3 Class StableInteger

This class defines recoverable integers, whose operations are implemented using the

```cpp
class StableInteger : stable
{
  int value;
  public:
    StableInteger();  // constructor
    // integer operations such as *
    int operator* (StableInteger &);
    int operator+ (int);
    // and -, +, / */
};
```

operator overloading facility of C++. The only public operation which updates the value of the recoverable integer is the assignment operation (implemented as *operator =* in C++ classes), which must first call *check* to ensure that there is an entry for the object in the P_list before changing the value. (Note that in C++, as in C, '=' stands for the assignment operation).

Here are some C++ related implementation details of operations StableInteger() and *operator =* (...):
StableInteger :: StableInteger : (sizeof (*this))
{
  /* StableInteger constructor.
     Empty function, used for passing the size of the object to
     the constructor of the base class, the parameter list following
     the "::*" is the specific mechanism for this purpose.
     The notation classname :: functionname is used to name specific
     functions of a class. */
}

int StableInteger :: operator = (StableInteger& newvalue)
{
  /* Stableinteger assignment operation */
  check (...);  /* check if the object is on the P_list */
  return (value = newvalue.value);
}

4.4 Class StableArray

This section describes three different ways in which a recoverable array, called
StableArray, can be constructed using the classes described above. It is our aim to illustrate that
our approach provides a convenient framework for a programmer to implement a variety of
recovery strategies. The three approaches used here are described below:

(i) construct a recoverable array out of recoverable integers;
(ii) construct a recoverable array out of ordinary (unrecoverable)
     integers, but inherit recovery from the base class stable;
(iii) construct a recoverable array out of recoverable integers and
     also inherit recovery from the base class stable.

4.4.1 Array of recoverable integers

A recoverable array can be constructed quite simply out of recoverable integers. Since this

class StableArray
{
  StableInteger .... [ ]  /* array of StableIntegers */
public:
  ....operator [] (int);  /* array index operation */
};

array is constructed out of recoverable objects, and we are employing the inclusive recovery
strategy, no special provision for recovery of the array is required. Note that it is necessary to
provide an array index operation for accessing a StableArray element.
4.4.2 Array of integers inheriting recoverability

Let us consider an alternative strategy: rather than performing recovery for an array on an integer by integer basis (previous case), let us adopt a complete checkpoint strategy. That is, a complete image of the array is saved, so recovery simply consists of restoring this image. We can do this by constructing the array out of ordinary (unrecoverable) integers and inheriting properties of stable for performing recovery.

```cpp
class StableArray : stable
{
    int .... [ ] /* array of integers */

    public:
        ...
        ...operator [] (int); /* array index operation */
);
```

As before, we need to provide an array index operation. However, since the array is constructed out of unrecoverable integers, explicit recovery measures are necessary. These are implemented by calling check from within the algorithm for the index operation to make sure that the P_list has an entry for the array. The advantage of this approach is that if a process performs a number of array assignments from within a recovery region, saving and restoring complete array images will generally provide faster recovery. A disadvantage is that the array image is saved even if only read operations are performed on it from within a recovery region (since the array image is stored as soon as an index operation is invoked from within a recovery region).

4.4.3 Array of recoverable integers inheriting recoverability

We can also adopt a mixed recovery strategy on the following lines: perform recovery of individual array element updates (of the form a[i] = b[j]) on an integer by integer basis (section 4.4.1) and recovery of complete array copy (of the form a = b) by using a complete checkpoint strategy (section 4.4.2):

```cpp
class StableArray : stable
{
    StableInteger .... [ ] /* array of StableIntegers */

    public:
        ...
        ...operator = (StableArray&); /* array copy operation */
        ...
        ...operator [] (int); /* array index operation */
    };
```
By constructing the array out of recoverable integers, we get recoverability for operations of the form \( a[i] = b[j] \). The array copy operation can be coded as follows:

```cpp
void StableArray::operator=(StableArray& newarray)
{
    check(...);  // save array image */
    new_region;  // start of a disjoint region */
    // for each element in the source array copy
    // the value to the destination array element */
    ...
    end_region;  // end of the disjoint region */
}
```

Note that we have turned off recovery for recoverable integers by establishing a disjoint region - since there is no recovery region (no erp has been performed in this region), assignments to recoverable integers will not be recoverable - this is just the effect we want (as an image of the array has been saved anyway).

These examples illustrate how our approach permits a variety of recovery techniques to be implemented very conveniently. There is no reason why we can not apply our approach to other types of objects. The next section shows how a very simple recoverable file can be constructed.

### 4.5 A recoverable file

The design of a recoverable file may be considered in the same manner as that of a recoverable array. In particular, a file can be made recoverable using the three techniques discussed in the array example. Let us consider the first case, a recoverable file constructed out of recoverable pages. A recoverable page (StablePage), whose structure is shown below, employs

```cpp
class StablePage : stable
{
    char .... [ ]  // page buffer */
    ...

public:
    ...
    read_page();
    write_page();
    flush_page();
};  // ...
```

techniques similar to those used for a recoverable integer. Since file pages reside on the disk, the first time a page is accessed through the \texttt{read\_page} or \texttt{write\_page} operation, the contents of the file's disk page are copied into the page buffer so that subsequent read or write operations on the page may be performed on this buffer. The \texttt{write\_page} operation saves an image of the buffer on
the log, using the same technique as was used for the assignment operation of recoverable integers, before the contents of the buffer are altered. A _flush_page_ operation is also needed to copy the buffer back onto the disk; this will ensure that page changes are reflected on the original file. If a recoverable file is constructed as an array of recoverable pages, file _read_ and _write_

```java
class StableFile {
    StablePage ... [ ] /* array of Stable Pages capable of holding the entire file */

    public:
    ...open (...);
    ...close();
    ...read (...);
    ...write (...);
};
```

operations will make use of _read_page_ and _write_page_ operations for accessing the opened file. When the file _close_ operation is invoked, a call is made to the _flush_page_ operation for each page which has been altered to ensure that the file is kept consistent. Clearly some optimisations are possible to the simple scheme suggested here. The StableFile can be designed to maintain a pool of a few StablePages in place of the array capable of holding the entire file. The same _flush_page_ operation can be used for selectively swapping file pages.

A recoverable file can also be constructed out of unrecoverable pages (similar to the example in section 4.4.2). In this case, the file _open_ operation can make use of the inherited _save_ operation for saving the entire file on the log. The mixed recovery strategy employed for arrays (section 4.4.3) can also be implemented for recoverable files.

5. Conclusions

The type inheritance facility of object based languages provides a very powerful means of constructing objects that inherit properties of lower level objects. We have shown how this facility can be exploited for constructing recoverable objects. We have done this in two steps: first we described the concepts of inclusive and disjoint multilevel recovery, and then we applied these concepts to the design of recoverable objects by making use of type inheritance facilities of C++. Perhaps the two most striking advantages of our approach are: (i) no new language features have been found necessary for incorporating recoverability; and (ii) the technique is flexible enough to
allow a wide variety of recovery strategies for objects to be considered. The ideas discussed here are currently being employed in the development of a prototype object based distributed system supporting recoverability and crash resistance.
Acknowledgements

Conversations with Ron Kerr clarified our understanding of object based programming. It was Robert Stroud who first drew our attention to the possibility of exploiting type inheritance facilities for implementing recoverability. The paper has benefited from critical comments by Pete Lee, Graham Parrington and Robert Stroud. This work was supported partly by an SERC/Alvey grant in Software Engineering.

References


