Object-Oriented Approach to State Restoration by Reversion in Fault Tolerant Systems

N. V. Tikhomirova and I. V. Shturtz **A. B. Romanovsky**
Applied Mathematics Department, Department of Computing Science,
Technical University of St. Petersburg , Russia University of Newcastle, UK

Abstract

This paper describes an approach to providing object state restoration in fault tolerant (FT) object-oriented (OO) computing systems by means of a reversion strategy. The unit of reversion is an object method. Two primitives are introduced that are used for creating a reverse-recoverable (RR) object: state saving primitive *save* and state restoring primitive *restore*. Reverse operations are processed in the order specified by a third primitive, *undo*. It is demonstrated how the approach suggested can be generalised to build a hierarchy of RR objects in the case of inheritance. The implementation of the approach is described for both the entire system and a separate object. The requirements for reverse operations are analysed. Finally, preferable areas of reversion applicability are discussed.

1 Introduction

Over the last years, OO programming languages have been widely used for creating complex software systems. Although OO programming provides a particularly appropriate system framework based on the abstract data type theory for enforcing reliability and controlling complexity, it is widely recognized that a complex system 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.

An important element in any measures for FT is an error recovery feature which transforms a system from an erroneous to an error-free state from which the system can continue to provide its specified service. There are two ways to reach the error-free state: backward error recovery (BER) and forward error recovery (FER) [1]. BER techniques restore a prior state of a system without regard to the current state, in the hope that the earlier state is error free. In contrast, FER techniques manipulate some portion of the current state to produce a new state, again in the hope that the new state will be error free. The difference between these two approaches is quite relative since BER may be considered as a specific case of FER.

BER mechanisms can be specified in terms of the strategy they adopt for preserving data needed for state restoration, i.e., recovery data. Three basic strategies are well known [1]: checkpointing, audit trail and recovery cache. The first one consists in saving a complete copy of the system state at certain moments during the system execution (recovery points). Rather than taking a copy of the entire system state when establishing the recovery point, the audit trail approach involves preserving the record of the subsequent system activity to enable all changes in the system state to be reversed, should the restoration of a prior state be required. The third strategy, the recovery cache, is an intermediate one: it stores the copy of that part of the state which is about to be changed first after the last recovery point.

In this paper we introduce a new strategy for state restoration, to be referred to as *reversion*. Similarly to the three strategies mentioned above, reversion transforms the system state into a prior state, but unlike them it can manipulate a current system state. The reversion strategy stores, as recovery data, the actions which caused changes in the system state. The restoration is achieved by executing actions *reverse* to the stored ones, in *reverse* order.

In accordance with [1], we will regard a system as consisting of a set of components structured according to a design; each component can be in turn regarded as a system. Components receive requests for services and produce responses when a service has been completed. As Lee and Anderson [1] observe, there is a strong correspondence between the component-based model of systems and the OO model of software, and it is convenient to think of objects as software components.

Objects (like components) have a well-defined external interface that provides operations for manipulating an encapsulated internal state. The term *recoverability* is taken to mean the ability to recover an earlier state of the object, thereby undoing the effects of operations that were performed on the object. Then a *recoverable object* is an object for which measures to support recoverability are provided.

The notion of the recoverable object is not new in literature concerning FT. An experimental implementation of recoverable objects in Distributed Path Pascal language was proposed in 1983 [2]. The state restoration is based on the recovery cache strategy there. Prior to the execution of a group of recoverable object routines, a recovery cache is established. The initial values of all variables changed during the execution of routines for the first time are stored in the cache. After all the routines of the group have completed their execution, the correctness of their results is checked. If no errors are detected, then the recovery cache is discarded and the routines return their results. Otherwise, the cached values of the internal variables are restored and alternate routines for all executed routines are invoked.

Object state restoration is considered in [3] within the frame for the implementation of recovery blocks in C++ language. The restoration strategy applied is the recovery cache, which can be implemented both by hardware and software. The latter is based on a unique property of C++ that permits the object copy semantics to be controlled by the programmer.

The computational model adopted for the distributed program system Arjuna [4] uses atomic transactions (AT) to control operations on persistent objects. Each persistent class is a descendant of class *StateManager* that contains two virtual operations: *save_state* and *restore_state*. The designer of a persistent class has to provide a suitable implementation for these operations. When a persistent object enters an AT, its state is copied from the stable storage into the volatile one by operation *restore_state* provided by this object. In the case of a successful completion of the AT, the object state is copied as a whole from the volatile storage into the stable one by operation *save_state* provided it was changed (which must be explicitly indicated). If the AT is aborted, the object state in the volatile storage is discarded. Thus, the entry of the object into the AT amounts to establishing a recovery point for this object. The fact that the object state is preserved as a whole, but only if the state was really changed, makes the recovery mechanism suggested in Arjuna similar to the software recovery cache based on the unique property of C++ language mentioned above.

To sum up, in the papers mentioned state restoration for recoverable objects is based on either checkpointing or recovery cache strategies. As distinct from this, the state restoration of reverse-recoverable objects that we propose is based on the reversion strategy.

The structure of the rest of this paper is as follows. In section 2 we try to substantiate why a new strategy for state restoration is needed. We also discuss there the relation between reversion and the other state restoration strategies. Section 3 describes the implementation of reversion itself. In particular, it introduces the notion of reverse-recoverable object and describes a scheme for programming such objects. Section 4 discusses what requirements object methods must satisfy for the reversion mechanism to be able to transfer the object from an erroneous state to a prior state. In section 5 we demonstrate how the nature of object state damage influences the applicability of the reversion strategy and conclude that it is worth using rather as an auxiliary tool for FT than as a basic one. Section 6 outlines areas where the reversion strategy could be successfully applied. Finally, section 7 contains concluding remarks.

2 Recovery by reversion

2.1 Why reversion is necessary

Let us define a set of objects as being at level (of abstraction) 0. Recursively, the level of an object using only objects of level k ($k = 0,1,...$) is defined to be $k+1$. If an error is detected during the execution of an object operation, the state restoration of the object itself must be provided as well as the state restoration of all objects it invoked during the operation. Some approaches to state restoration in multilevel systems have been proposed. The most widespread among them are those based on the recovery cache and checkpoint strategies. However, a certain disadvantage of these approaches is that they implicitly assume that the states of all objects are stored in the main storage and operations generate their results exclusively by assignment to storage locations. Then recovery is done simply by reversing those assignments. But, as observed in [5-8], there are objects for which such assumption does not hold. These are objects of non-program nature [8] (e.g. peripheral devices such as a printer, physical disk; robots in computer control systems, etc.). The only possible way to restore the state of such objects is executing reverse operations. For example, let *Robot_Arm* be an object implementing the arm of a robot. If the last operation executed was *Robot_Arm.Move_Left*, then the physical state of the arm can be restored by executing the reverse operation *Robot_Arm.Move_Right.* If the non-program object is not reversible (e.g. a printer), then the aim of reversion operation is to undo the changes produced by direct operation. For example, operation *Printer.Write(no_of_line)* can be associated with reverse operation *Printer.Cancel(no_of_line)* which will print the message (*"please ignore the line", no_of_line*) [6].

A mixed recovery strategy supporting the state restoration both for assignment and other operations was first introduced in the paper [5]. The assignment operation is handled with the help of the normal recovery cache strategy while it is proposed that the operation requiring special recovery action be structured as a recoverable procedure that provides three entry points: save, reverse and normal. The save entry point preserves recovery data. The reverse entry point is invoked in the case of an error to provide an opportunity to restore the recoverable procedure in accordance with the recovery data. The authors demonstrate how the normal recovery cache strategy can be extended to serve recoverable procedures as well.

The paper [7] describes how the mixed state restoration strategy based on extending the normal recovery cache mechanism can be applied to multilevel modular software systems. The peculiarity of implementing the approach proposed is the following: if the language designer wants to add a recovery mechanism to an existing language, he will have to change both the run-time interpreter and the compiler. This would be a labour-consuming task and so undesirable. Moreover, the following restriction is needed to allow the automatic invocation of reverse operations by level 0 interpreter: if $v{:}T$ is the result value returned by direct operation, then the only parameter of reverse operation is $v{:}T$.

A recovery mechanism implemented on level 0 supports the recoverability of upper level objects. It means that should the error be detected during the execution of the operation of the level k object, the automatic state restoration for all objects being invoked during this operation will be executed by the interpreter. But, as was mentioned above, there are some difficulties in supplying the system with such an interpreter. Consequently, the problem of creating recoverable objects when the lower level objects are not recoverable seems to be important. In this case, a recoverable object is responsible not only for its own restoration but for the state restoration of all objects it has used. The distributed file system described in the paper [9], where the unrecoverable features of the lower level interface of the standard Unix file system are concealed in the upper level interface to provide recoverable abstract files, can be considered as an example of a system where this problem had to be solved. The recoverability of the upper interface is achieved by logging the operations invoked on this interface. Codes of operations with appropriate parameters are stored in the stack. When the user invokes recovery, operations reverse to those saved in the cache are executed (reverse operations are put in accordance with direct ones with the help of operator *case*). For example, an operation of the upper interface, *lopen*, consists of two primitive operations which are locking the file and calling the standard Unix file system open operation. Hence, the codes (*lock, open*) with appropriate parameters are saved in the stack. If recovery is invoked, the file is closed and unlocked. The problem under discussion cannot evidently be solved by the recovery cache or checkpoint strategies.

The discussion above proves the necessity of a recovery strategy based on reverse operations which would allow restoring the state of non-program objects as well as creating recoverable objects if the lower level objects are not recoverable. The main principles of such a strategy are observed in the papers [5] and [7]. We propose introducing a distinct strategy on the basis of these principles and calling it *reversion*. In this paper we demonstrate how reversion is applied in OO software systems and describe an implementation of reversion based on such features of OO languages as inheritance and method overloading, polymorphism and dynamic binding, without changing the run-time interpreter or the compiler. The approach suggested is quite general and applicable for any OO language. In this paper we use the notation of the widely used OO language C++.

2.2 Audit trail and reversion

In contrast to the checkpointing and recovery cache strategies, reversion as well as the audit trail strategy are based on preserving a history of the system activity. So, the audit trail mechanism preserves the values of that part of the system state that is about to be changed. To restore a recovery point, the audit trail mechanism processes the recovery data stored in the trail in reverse order. Each entry specifies a modified system state part and the values to which it must be restored. The main application domain of the audit trail strategy is data base systems (see, for example, [10]). Before executing any action which would change the data base, a copy of the data to be modified is preserved in a log so that the recovery by audit trail can be executed.

Unlike audit trail, the reversion strategy stores the operation that causes the system to be changed. The restoration of a recovery point using reversion is currently achieved by executing the operations that are reverse to those saved in the trail in reverse order. Each entry specifies a modified system state part, a reference to the operation executed and some parameters needed for recovery. Let us consider an example. We shall assume that integer N has value 15 and the operation consists in adding integer M with value 3 to integer N. The recovery data for the audit trail strategy is N value equal to 15. The recovery data for reversion is a reference to the add operation and M value equal to 3. Should the restoration of a prior value of integer N be required, the reversion mechanism will execute the subtract operation with respect to the current value of integer N. In the case of audit trail, the value preserved will be assigned to integer N.

There are some situations where reversion saves memory and time overhead as compared to the strategies that preserve a system state (or part of it) before its modification. One of those is when the total volume of data that represent a system state is rather big and, in addition, the chunk of data modified during the operation is big, too. For example, the reversion strategy for the operation of adding integer *N* to a set of variables in a data base will require only saving parameter *N* and executing the reverse operation *Subtract(N)* to restore the prior state of the data base. The recovery cache strategy, however, will require saving and then restoring the old values of all data modified.

If in the reversion strategy the assign operation is taken as the action reverse to any action changing the system state, then reversion degenerates into audit trail. So, in a sense, reversion can be seen as a generalisation of audit trail.

As to operations that change the system state in ways other than by assigning values to variables, the audit trail strategy is not able to eliminate the effect of such operations because, being a BER strategy, audit trail is not allowed to use the knowledge of the current system state whereas reverse operations, in general, are executed just over the current state. Reversion, unlike audit trail, can use the current state to execute the state restoration. This suggests we should classify reversion as a FER technique rather than a BER technique. This is no contradiction, since we regard the reversion strategy as a generalisation of the audit trail strategy and, as was mentioned above, the BER approach can be seen as a particular case of the FER approach.

3 Implementing reversion in OO systems

3.1 Reverse-recoverable object

Let an *operation* be defined as the execution of any of the object methods resulting in the changes in the object state or invoking some operation external to the object. The state of the object is *stable* while none of its methods is being executed. Then a *reverse-recoverable* (RR) object is an object each of whose stable states can be restored by executing the reverse actions in the proper order. Thus, for a RR object, recovery points are those and only those moments at which it is in a stable state.

3.2 OO framework of reverse-recoverable objects

One of the principles we relied on in working out our restoration scheme is using the existing OO languages without modifying them. Therefore, it is necessary, first of all, to specify how the initial object is to be modified by means of an OO language to become a RR object.

Though it is not any *object method* that is an *operation* in the strong sense defined above, we will not distinguish between these two terms, for simplicity sake. So the unit of reversion is an object method. Two methods implementing restoration primitives, *save-method* and *restore-method*, are defined for each object method. Let method *A* transfers an object from state *Q1* into state *Q2* (Figure 1). Suppose that for any recoverable object there exists sequence *~A* of reverse operations, whose execution returns the object from state *Q2* back to

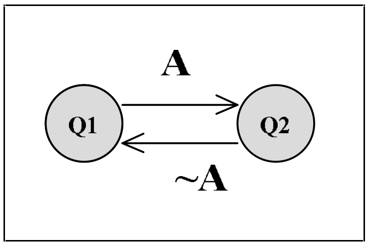

Figure 1. State transition diagram

state *Q1*. That sequence often consists in a single operation. Examples are easy to give: to add an integer - to subtract the integer, to allocate memory - to free the memory, to draw a figure - to erase the figure, to take a tool - to put down the tool, etc. To execute sequence *~A*, it is sufficient to know the parameter values of the methods forming this sequence. The save-method contains saving and preserving these values. The restore-method contains sequence *~A* itself.

Let us consider the scheme of designing a RR object. We propose the following class inheritance hierarchy (Figure 2), where *C* is the initial class that does not have a reverse recovering feature, *SAVE_RESTORE_C* is the one containing the description of *save*- and *restore-methods* corresponding to the methods of class *C*, and, at last, class *RECOVERABLE_C* obtained by multiple inheritance of the two previous classes is the required RR class, i.e. the one whose instances are RR objects.

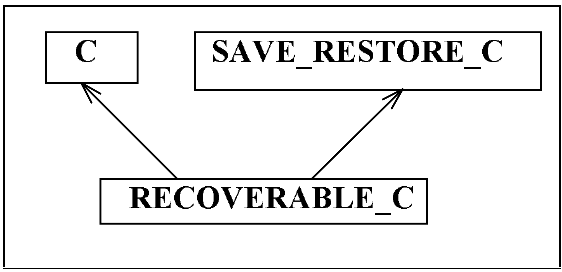

Figure 2. The scheme of designing a RR object

In developing our scheme four requirements were formulated:
(1) that the primary description of the object be explicitly separated from the part of the description concerning reversion;
(2) that the reversion mechanism be transparent for the users of RR objects;
(3) that the scheme be flexible in case the initial classes are linked by inheritance relation;
(4) that the object interface be maintained after the initial object has been converted into a RR one.

It is obvious that the scheme shown in Figure 2 meets the first two requirements; the third will be discussed later. To meet the fourth requirement, each method of class *C* is to be transformed in the following way:

C::A() =>
RECOVERABLE_C::A() { SAVE_A; C::A(); }

This means that when executing method *A* of class *RECOVERABLE_C*, not only method *A* of class *C* is executed but before that all data necessary for restoration are preserved. Figure 3 demonstrates the implementation of this transformation scheme in the C++ language. All methods of class *C* are defined in class *SAVE_RESTORE_C* as virtual ones.

If the user of the object wants now to make it RR, he or she has to define it as an instance of class *RECOVERABLE_C* instead of class *C*.

Apart from the one described, some different schemes of constructing a RR object can be considered (Figure 4).

However, each of those has important disadvantages as compared with the proposed one. In particular, with scheme (a), none of the requirements listed above are met. Scheme (b) differs from scheme (a) in that it introduces class *RECOVERABLE_C* which enables the reversion mechanism to be made transparent for the RR object user. So this one makes it possible to meet requirement (4), but at the cost of a great mishmash in redefining the appropriate methods. Scheme (c) does not meet requirement (2). Class *RECOVERABLE_C* in scheme (d) is empty since all *save-* and *restore-methods* and new methods (those which are formed from the initial method and *save-method*) are defined in class *SAVE_RESTORE_C*. Therefore, scheme (d) is identical to scheme (c).

```
class C {                          class SAVE_RESTORE C {
      void A(int);                   virtual void A(int) = 0;
}                                    void  SAVE_A(void);
                                     void A(int x) {  SAVE_A(); A(x); }
                                     void  RESTORE_A(void);
                                    }

           class RECOVERABLE_C: public C, SAVE_RESTORE_C {
              void A(int x) { C::A(X); }
            public:
              void A(int x) { SAVE_RETORE_C::A(X);  }
            }
```

Figure 3. Transformation of original class method into method of RR class

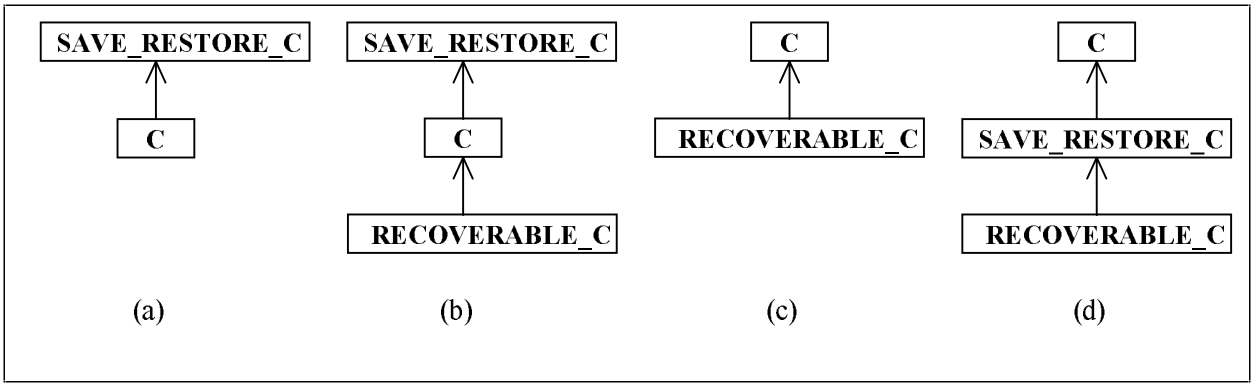

Figure 4. Possible schemes of constructing RR object

3.3 Reverse-recoverable object hierarchy

Inheritance is one of main principles of OO programming, so we have to generalise our scheme of constructing RR object for the case when the initial classes are in the inheritance relation.

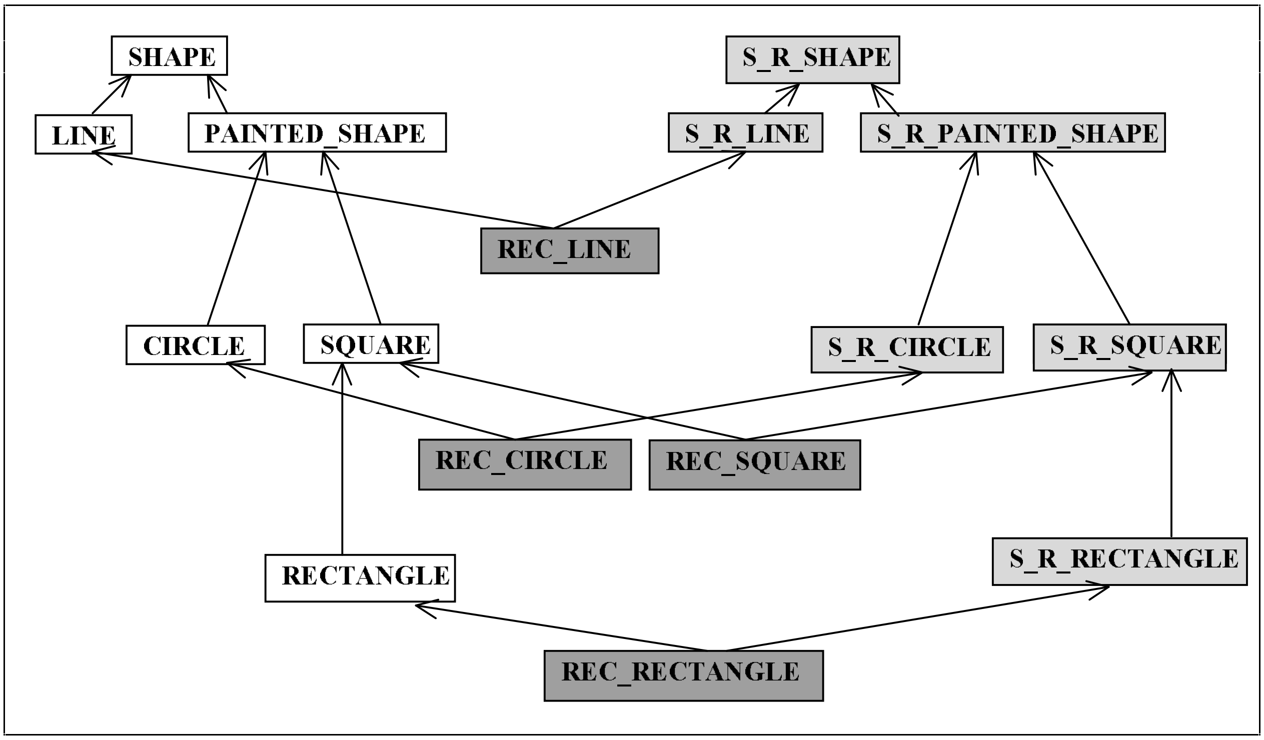

Figure 5. Class inheritance hierarchy

Assertion: The inheritance hierarchy of *SAVE_RESTORE-classes* coincides with that of the initial classes (this assertion is illustrated by Figure 5)

Proof: A method of class *C* can be obtained in three ways:

1) by inheriting a method of the ancestor class without redefining it in class *C*;

2) by inheriting a method of the ancestor class and redefining it in class *C*;

3) by defining a new method in class *C*.

If a method is inherited without redefining, then the same is true about the reverse method. Therefore, it seems reasonable that the appropriate *save*- and *restore-methods* should also be obtained by inheritance without redefining. The redefining of a method can lead to that of the reverse method, which, in turn, can entail the redefining of *save*- and *restore-methods*. So in this case *save*- and *restore-methods* will be either inherited without redefining or redefined. In the case of defining a new method in class *C*, new *save*- and *restore-methods* are defined in class *SAVE_RESTORE_C* accordingly. Consequently, in all of the three cases the hierarchy of *SAVE_RESTORE-classes* coincides with the initial class hierarchy.

If class *SAVE_RESTORE_C* (obtained by inheritance) should turn out empty (that can happen when the redefining of methods of class *C* does not lead to that of *save*- and *restore-methods* of class *SAVE_RESTORE_C*), the appropriate class *RECOVERABLE_C* can be obtained by inheritance of the ancestor of class *SAVE_RESTORE_C*.

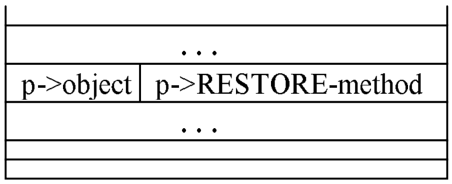

Figure 6(a). Global stack

Figure 6(b). Object's stack

3.4 Implementation reverse mechanism for entire system and individual object

The reverse mechanism can be adapted both for the entire system and for an individual object. In the first case, it is necessary to use a global stack where the order of the operations being executed in the system is stored. In the second case, each object has its own stack containing the order of the operations being executed over this object. Pictures of a global stack and an object stack are shown in Figure 6, where *p->object* is a pointer to the object over which the operation has been executed and *p->restore-method* is a pointer to the method reverse to executed one. (To make the implementation simpler, we keep reverse operations in the stack instead of initial ones.)

Along with these stacks, we also need those for preserving the parameter values of reverse methods. In the case of the entire system reversion, these stacks are created for each method (whose reverse method has parameters) of each class. In the case of an object reversion it is done for each method of each object. When executing *save-method*, a new element is saved in the appropriate stacks. Figure 7 gives the interface of template class *STACK* in C++ notation.

```
template<class T>
  class STACK {
    T* s;
     int stack_top;
     int max_size;
     int current_size;
   public:
     STACK(int);
     ~STACK(void);
     void Push(T);
      T* Pop(void);
  }
```

Figure 7. Interface of template class STACK

The type of a stack element depends on the data preserved. An element of the global stack consists of a pointer to the object and a pointer to the method. How should this element be described for it to be in accordance both with a pointer to an object of any class and a pointer to a method belonging to any

```
struct STACKELEM {
    void (SAVE_RESTORE::*pf)(void);
    SAVE_RESTORE *p;
}
```

Figure 8. Description of global stack element

class? Evidently, a base class, an ancestor for all RR objects, is needed. In OO language EIFFEL we could use class ANY for this purpose, which is the base class for all created objects. There is not a similar class in C++ language, but it can be easily created. Given class *SAVE_RESTORE* is such a base abstract class, we can describe the global stack element as shown in Figure 8.

The entire-system reversion mechanism itself is extremely simple and consists in invoking *restore-method*, the pointer to which is on top of the global stack:

```
STACK<STACKELEM> St;

void Undo(void)
{
    STACKELEM *elem = St.Pop();
    ((elem->p)-> *(elem->pf))();
}
```

In the case of reversion used for an individual object, each RR object has its own method *undo*. However, the body of this method is the same for all objects and can be inherited from the base class *SAVE_RESTORE:*

```
STACK<void(SAVE_RESTORE::*)(void)> St;

  void Undo(void)
  {
    void(SAVE_RESTORE::**elem)(void);
    elem=St.Pop();
    (this->p)->**elem)();
  }
```

3.5 Creation and destruction of objects

An object can neither create not destroy itself. It has to be either created or destroyed by some other object that provides methods which allow the operations reverse to the creation and destruction of that object to be executed. While in the case of entire system reversion the saving of the object state copy is sufficient to execute the operation reverse to the destruction operation, for object reversion it is also necessary to preserve all recovery stacks used by the object being destroyed.

4 Requirements for reverse methods

As was discussed above, reversion as an instance of the FER approach can need the current state of the object to restore its prior state. To execute state restoration in the case when the current state of the object is erroneous, reverse methods have to meet some requirements. Let *~A* be the method reverse to method *A*. Provided methods *A* and *~A* are error-free, the state-transition diagram for them is as shown in Figure 1.

But if in executing method *A* the object passes from error-free state *Q1* to erroneous state *Q2*, then method *~A*, provided it uses the "really erroneous" part of state *Q2*, will transfer the object to state *Q3*, which in most cases will differ from state *Q1,* as shown in Figure 9 (we will use the term an *inconsistent* part to denote this part of the state below).

In order to avoid the situation shown in Figure 9, it is necessary that method *~A* be interdicted to use the inconsistent part of the state Q2. However, the inconsistent part varies depending on the fault,

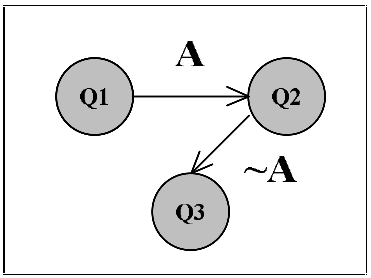

Figure 9. State transition diagram

which involves inconsistency, while our approach assumes that only one reverse method corresponds to the direct method. This suggests that the only way to write the reverse method for restoring the erroneous state of the object is to interdict the use of the entire current state by reverse method that leads to degrading of the reversion strategy.

5 The types of object state damage

In order to better understand the preferable domains of using reversion for state restoration, let us define the notions of object state *incorrectness* and *inconsistency*.

If in executing method *A* the object transfers from an error-free state in an erroneous state provided method *A* receives error-free input data, then this erroneous state will be defined as an *inconsistent* state. The transition of the object to an inconsistent state means that a software or hardware fault took place precisely while executing method *A*.

Normally, fault detection takes some time called the latent period of fault revealing. Assume that in just such period method *A* (whose object may have been in an erroneous state) invokes method *B* of a different object, called, for example, *B_OBJ*, with erroneous values of parameters. If the values received do not evoke any suspicion, method *B* will be completed, having transferred the object in an incorrect state. The incorrectness of *B_OBJ* state can also be caused by the invocation of *B* before the occurrence of a fault in *A*. Though the incorrect state is consistent because it was reached without faults in *B* or methods it has used, the correct state should be restored because otherwise further valid calls to *B_OBJ* may cause new faults.

Programming method *~A* requires that the restrictions in the previous section should be observed. The difficulties associated with this process suggest that for the recovery of an object whose state is erroneous it is preferable to use a recovery strategy other than reversion. In contrast to method *~A*, it is unnecessary to impose any restrictions on method *~B*, and the execution of method *~B* will transfer the object from an incorrect state to the error-free state that the object had had before invoking method *B*.

To sum up, reversion is worth using for object state restoration only when the object is in a consistent state.

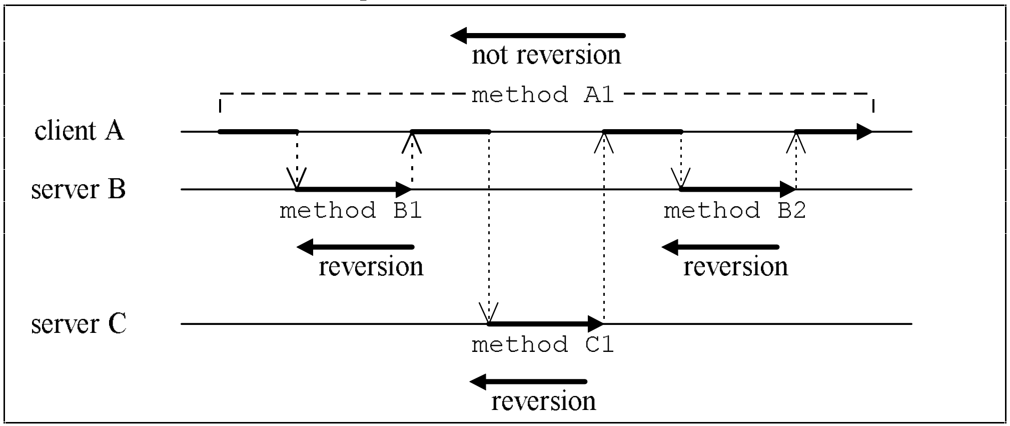

Figure 10. Recovery scheme for the system "client - server"

6 Application domains of reversion strategy

6.1 Reversion as auxiliary means for recovery

As follows from the preceding section, an object is worth converting into a RR one only when it is guaranteed that, being in consistent state, the object will remain in a consistent state after executing any of its methods. This conclusion suggests that reversion is rather an auxiliary recovery means than a basic one.

Due to a great overhead in providing FT, in particular, to software faults, FT techniques are often used only for the most critical objects in order to make them able to resist failures. The reliability of non-critical objects can be enhanced by the use of fault prevention techniques and such objects are subsequently assumed to be error free.

To provide the service required, a critical object may need to address another one. We will call such an object a *client* and those which it addresses *servers*. If the BER approach is used for client error recovery, obviously, apart from the rollback of the client, the rollback of all of its servers will be required. Otherwise, the system as a whole will not return to a prior state. Following [11], we shall maintain that every server has to provide its own state restoration. It is precisely for the restoration of a server that is not a critical object that it is convenient to use the reversion strategy. The proposed recovery scheme for the system "client - servers" is shown in Figure 10.

Let us consider a simple example of how reversion can be used as an auxiliary means for state restoration in the recovery block (RB) scheme. Following [12], we shall assume that algorithmic diversity is implemented on the level of objects. Then the RB scheme would presumably take some form like the following:

```
enum STATUS{NORMAL,EXCEPTION,ERROR,...}
STATUS recoveryBlock
 (ACCEPTANCETEST *pa,  // pointer to class containing
                       // an    acceptance test algorithm
  VARIANT *pv[],       //pointers to variants
  int n,...)           // the number of variants
 {
   for (int i=o; i<n; i++)
     { pv[i]->variantDefinition();
       if (pa->acceptanceTest() ==1)
          return 'NORMAL';
     }
   return 'ERROR';
 }
```

Let every object involved as an alternate in the RB scheme (the pointers to these objects are in massive *pv*) during the execution of method *VariantDefinition*() invokes the method of its server. Such a server becomes an implicit participant of the RB (unlike the client, which is an explicit participant of RB). If the acceptance test is not satisfied after method *VariantDefinition*() is completed, then the states of all implicit RB participants have to be restored to the moment when the alternate started (we do not consider the state restoration of the explicit participant because it will have to be provided by strategies other than reversion).

A simple recovery algorithm can be implemented if implicit RB participants are RR objects and their names are known while executing routine *recoveryBlock*. If a RR object is to be able to become an implicit RB participant, a status variable *participantRB* indicating whether the object is a participant has to be introduced and the following methods have to be implemented for the object:

enterRB - assigns the value of 1 to *participantRB* and creates stacks needed for saving recovery data;

rollBack - invokes methods *undo* for all elements of the object stack (see section 3.4);

exitRB - assigns the value of 0 to *participahtRB* and discards recovery stacks.

```
STATUS recoveryBlock
              (ACCEPTENCETEST *pa,
               VARIANT *pv[],
               int n,...)
{
 LIST serverList;
 for (int i=0; i<n; i++)
  {
   pv[i]->variantDefinition
                 (...,&serverList);
   if (pa->acceptanceTest() == 1)
     {
       // for all elements of serverList list
       for(...)
         // p - pointer on the next in turn element
        { p->exitRB(); }
       return 'NORMAL';
     }
    // for all elements of serverList list
   for(...)
     { p->rollBack(); p->exitRB(); }
 }
  return 'ERROR';
}
```

Figure 11. Recovery block algorithm

The variable and the methods can be inherited from class *SAVE_RESTORE*.

Taking into account what we said above, the recovery block algorithm may presumably be modified as shown in Figure 11.

List *serverList* can be built while involving implicit participants in the alternate. For this purpose, any invocation of the server method is replaced by one of routines *call* to check whether this server has been included in list *serverList*. If it has, then the server method is invoked; otherwise before its invocation the pointer to the server is inserted in *serverList*, and method *enterRB* of this server is executed.

We have considered the example of using reversion for server state restoration for the case when, upon implementing a client object for tolerating design faults, software diversity is applied. Reversion can also be used for rollback of servers in the case when a client object is implemented so as to be able to resist hardware faults. The only condition for reversion applicability in this case is that the fault should not damage memory where the server itself and its recovery stacks are kept.

6.2 Other reversion application domains

The reversion mechanism is applicable not only for providing FT. The use of reversion may prove to be convenient in various interactive systems, for example, in text and graphic editors, in software shells. If the result of an action executed does not suit the user, there is a chance to come one or more steps back. When the reversion strategy is used for providing such supplementary services, it is not very difficult to write the reverse method. In most cases, the method needed is already present among object methods. Consider an example of a text editor. If there is method "*delete char*", then there is method "*insert char*"; method "*remove mark*" is the counterpart of method "*mark*", method "*replace marked text piece*" is its own reverse and so on. In text editors, the service described is known as the UNDO facility.

As was mentioned above, the reversion strategy seems to be useful and often irreplaceable in recovering objects of non-program nature, when the only possible way of restoring the object state is executing reverse actions.

The reversion mechanism can also be successfully used as a means of resolving deadlocks when the rollback of some objects is required to break the deadlock.

7 Conclusions

This paper introduces a state restoration strategy called reversion and proposes a way of implementing this strategy in OO software. It discusses cases where just such a strategy is necessary and where the use of reversion is preferable to that of other strategies. As it follows from the discussion, in designing FT OO software, reversion can only be used together with other state restoration strategies. The best combination seems to be that of reversion and the recovery cache. Thus, a promising direction of future research is elaborating the OO framework for FT software that will provide a convenient use of reversion jointly with the recovery cache strategy.

Acknowledgments

This research has been supported by the Royal Society (Joint Project 638072.P793) and by Basic ESPRIT Action 'Design for Validation' (DeVa).

References

[1] Anderson T., Lee P., *"Fault Tolerance: Principles and Practice"*, 2nd Edition, Springer-Verlag, 1990

[2] Campbell R.H., Anderson T., Randell B., "Practical Fault Tolerance Software for Asynchronous Systems", In *IPAC Safecomp'83*, Cambridge, UK

[3] Rubira-Calsavara C.M.F., Stroud R.J., "Forward Error Recovery and Backward Error Recovery in C++", In *PDCS 1st Year Report*, vol. 1, Toulouse, 1993

[4] Shrivastava S.K., Dixon G.D., Parrington G.D., "An Overview of the Arjune Distributed Programming System", *IEEE Software*, vol. 8, no. 1, pp. 66-73, 1991

[5] Horning J.J., Lauer H.C., Melliar-Smith P.M. and Randell B., "A Program Structure for Error Detection and Recovery", In *Lecture Notes in Computer Science,* vol. 16, pp. 177-193, Springer-Vervag, 1974

[6] Shrivastava S.K., Banatre J.P., "Reliable Resourse Allocation Between Unreliable Processes", *IEEE Trans. on Soft. Eng.* SE-4, no. 3, 1978

[7] Cristian F., "A Recovery Mechanism for Modular Software", In *4th Int. Conf. on Software Engineering,* Munich, 1979

[8] Romanovsky A.B., Shturtz I.V., "Unplanned recovery for non-programm objects", *Computer systems science & engineering,* vol. 8, no. 2, pp. 72-79, 1993

[9] Jegado M., "Recoverability Aspects of a Distributed File System", *Software: Practice and Experience,* vol. 13, pp. 33-44, 1983

[10] Gray J.N., "Notes on Data Base Operating Systems", In *Lecture Notes in Computer Science* 60, ed.R.Bayer et al, Springer-Varlag, Berlin, pp. 393-481, 1978

[11] Campbell R.H, Randell B., "Error Recovery in Asynchronous Systems", IEEE *Trans. on Soft. Eng*, vol. SE-12, no. 8, pp. 811-826, 1986

[12] Xu J., Randell B., "Object-Oriented Construction of Fault-Tolerant Software", University of Newcastle upon Tyne, Tech. report no. 444, July, 1993