Shared Recoverable Objects

Jörg Kienzle, Alfred Strohmeier

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

email: {Joerg.Kienzle, Alfred.Strohmeier}@epfl.ch
phone: +41 (21) 693.42.37 – fax: +41 (21) 693.50.79

Abstract. This document describes an implementation of recoverable objects that can be accessed concurrently. After a brief description of the possible uses of recoverable objects and after reviewing some of the new features of Ada 95 used in the implementation, the design issues are discussed and the interface of the recoverable object class is presented. An example application using multitasking demonstrates its applicability.

Keywords. Recoverable Objects, Ada 95, Concurrency, Object-Oriented Programming, Controlled Types, Abort Deferred Regions, Transactions, Software Fault Tolerance.

1 Introduction

It is inconceivable that the role of fault tolerance in computing systems will diminish, its importance can only increase. There are clear indications of a widespread and growing demand for increased reliability in many areas. Fault tolerance techniques for coping with the occurrence and effects of hardware failures are now well established and form a vital part of any reliable computing system. Hence, *software fault tolerance* — the ability of software to cope with faults, especially design faults — is getting more and more important [1].

Various approaches (N-version programming, recovery blocks) have been proposed to address such unanticipated software faults. These mechanisms have been modified or extended to address concurrency (transactions, conversations, atomic actions). In most of these schemes, especially those using backward error recovery, it is often necessary to be able to undo the effects of a faulty computation, that is restoring the state of the application to a point where it was known to be correct. Objects offering such restoration capabilities in a concurrent environment are called *shared recoverable objects*.

This paper describes an implementation of shared recoverable objects for Ada 95 [2, 3]. The outline of the paper is as follows: section 2 describes recoverable objects in more detail and situates them with respect to various approaches for providing software fault tolerance; section 3 gives an overview of the important features of Ada 95 for the implementation of shared recoverable objects; section 4 presents the interface of the shared recoverable objects package and describes the offered functions; section 5 shows an example of how shared recoverable objects can be of help in an application using multitasking; section 6 exposes the internals of the implementation and the last section draws some conclusions and mentions future work.

2 Recoverable Objects

2.1 Terminology

A *recoverable object* is an object whose state can be saved and later on restored, if needed. Saving is sometimes also referred to as *taking/establishing a checkpoint/snapshot*; it is usually performed when the system, including the object to be saved, is in a consistent state. Once saved, the state of the object can be restored at any time. The reasons for restoration depend on the context in which the recoverable object is used.

It is possible to take several snapshots of an object, which means that it is possible to have multiple *versions* of the state of an object to which one can revert to. Branches are not allowed: each version has at a given time at most one parent and one child. The most recent version is called the *current* version.

Shared recoverable objects can be used by multiple tasks concurrently. Such tasks will be called *participant tasks*. Note that a shared recoverable object does neither provide nor impose any concurrency control between tasks accessing it. Consistent access to the data encapsulated in the object must be ensured by the application programmer.

2.2 Usage

Recoverable objects can be of use in a variety of different contexts, i.e. in applications that want to provide "undo" functionality (see "Example" in section 5). Before letting the user modify an object, a snapshot of its state is taken. If later on the user of the application is not satisfied with the results of his changes, the old state of the object can be restored. Multiple snapshots allow multiple levels of "undo".

More and more applications make use of multitasking to improve performance, or to provide better feedback and interaction to the user. Recoverable objects should therefore be designed with concurrency in mind.

Software fault tolerance techniques, especially those using backward error recovery such as recovery blocks and transaction-like systems can also make use of recoverable objects.

Recovery blocks [4] have been introduced as a software structuring mechanism providing software fault tolerance based on the concept of design diversity and backward error recovery. A recovery block consists of one or more alternatives implementing functionality, coupled with an acceptance test that determines whether an alternative has functioned correctly or not. Under normal conditions, only the first alternative is executed. Only if an error is detected by the acceptance test, other alternatives are tried successively. The typical syntax for expressing recovery blocks is as follows:

```
ensure <acceptance test> by <primary alternate>
else by <second alternate>
...
else by <n'th alternate>
else error
```

It is important to note that prior to execution of the primary alternate, a snapshot of the current state of the component is taken. When an alternative fails to pass the acceptance test, this state is restored, and the next alternative is tried. This is repeated until either an alternative succeeds, or there are no more alternatives available. In that case, the execution of the component is considered a failure and an error is propagated.

Recoverable objects can be very useful to a programmer of a recovery block. He can encapsulate the state of the component in one or several recoverable objects and take a snapshot before executing the primary alternative. If it fails the acceptance test, the state can be easily restored.

Transactions [5] are a well-known mechanism for structuring an application to provide data consistency in a concurrent environment and in the presence of failures. A transaction is a sequence of statements, whose execution is characterized by the famous ACID properties:

- *Atomicity*: A transaction gives an "all or nothing" guarantee: either all the state changes of the transaction are performed, or none are.
- *Consistency*: The execution of a transaction starting in a consistent state will produce another consistent state.
- *Isolation*: Transactions executing concurrently (*competitive* concurrency) are isolated from each other; the result must be the same as if they had been executed in some serial order.
- *Durability*: Once a transaction has terminated successfully, its effects are permanent, even when a failure occurs.

As with recovery blocks, recoverable objects can help providing atomicity of transactions. If the transaction aborts, the changes to the state of the application can be undone, provided that it is encapsulated in one or several recoverable objects and that a snapshot has been taken before entering the transaction. Nested transactions [6] can make use of nested snapshots.

Conversations [4], atomic actions [1] and finally coordinated atomic actions [7] can be seen as extensions of the basic transaction concept to cope with *cooperative* concurrency. In all three schemes, processes or tasks cooperate and potentially access shared data, that must be restored if an error occurs and if backward error recovery is used. Here again, recoverable objects are only useful if they can "live" in a concurrent environment.

3 Features of Ada 95

This section will describe two features of Ada 95 that are used extensively inside the implementation of shared recoverable objects, namely controlled types and task identification.

3.1 Controlled Types

Controlled types [2, 7.6] have been introduced in Ada 95 to facilitate resource management within abstract data types while preserving the abstraction. The language-defined package `Ada.Finalization` declares two abstract tagged types, `Controlled` and `Limited_Controlled`. A user-defined type derived from one of these two types inherits the operations `Initialize`, `Finalize` and (in case of the type `Controlled`) `Adjust`. These operations are somehow special, since they are automatically invoked in the following circumstances:

- `Initialize` — is invoked whenever an object of the controlled type is created and there is no explicit initialization.
- `Finalize` — is called just before a controlled object is destroyed, i.e. goes out of scope, is deallocated using an instantiation of `Ada.Unchecked_Deallocation`, or is overwritten during an assignment.
- `Adjust` — is invoked during an assignment just after the target object has been overwritten.

The default implementations of these three primitive operations do nothing at all. By overriding the inherited versions in the derived type, the application developer can precisely control object creation, destruction, and assignment. Especially the ability to associate automatic finalization actions with a data type is extremely important for Ada, given the orientation toward information hiding, coupled with the many ways that a scope may be exited in Ada (exceptions, exit, return, abort, asynchronous transfer of control).

3.2 Task Identification and Task Attributes

In order to identify instances of a task type, the language-defined package `Ada.Task_Identification` defines a `Task_ID` type. The operations provided in this package allow an application programmer to get the `Task_ID` of the current task, to abort tasks identified by a `Task_ID`, and to query information about the status of a task.

It is also possible to associate user-defined information with a task using the generic package `Ada.Task_Attributes`. The application programmer can attach a data type to each task in the system by instantiating the package:

```
package My_Task_Info is
   new Ada.Task_Attributes    (Attribute => My_Data_Type,
                               Initial_Value => My_Initial_Value);
```

The package provides functions to set and retrieve the value of the attribute associated with each task. Task attributes are of special interest when using controlled types: through their `Finalize` primitive operation application-defined actions can be performed on task abortion or termination.

4 Design Issues

4.1 Update Strategies

When implementing recoverable objects, one has to choose between two different strategies of updating the state of the object: *in-place* update and *deferred* update.

In-place Update

When using in-place update, only the current version of the recoverable object is visible and gets modified. The recoverable object can be represented as a list of versions of the state of the object (see figure 1), where all except the head of the list is hidden from the user. The different versions of the object are typically identified using a version number *n*. When a snapshot is taken, a copy of the current version is made and added to the list of hidden states; subsequent modifications are still performed on the current version. When a certain version must be restored, the corresponding state is copied from the list back to the current version.

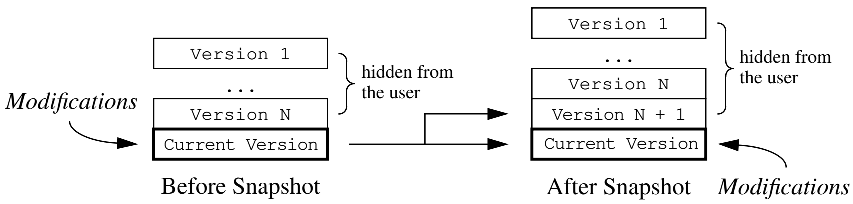

Figure 1: In-place Update

Typical operations that must be provided by an implementation using in-place update are:

- *Establish Checkpoint*: take a snapshot of the current state of the object (duplicate the current version of the state)
- *Discard Checkpoint (n)*: discard the version *n* (take the corresponding version out of the list)
- *Restore Checkpoint (n)*: restore the state of the object saved in version *n* by replacing the current state with the one stored in the version

Deferred Update

When using deferred update, a new version is created when a snapshot is taken and handed back to the user. Subsequent modifications are applied to this new version (see figure 2). Once the user is satisfied with his changes to the object, he can commit the modifications; this results in copying the new state back to the previous version.

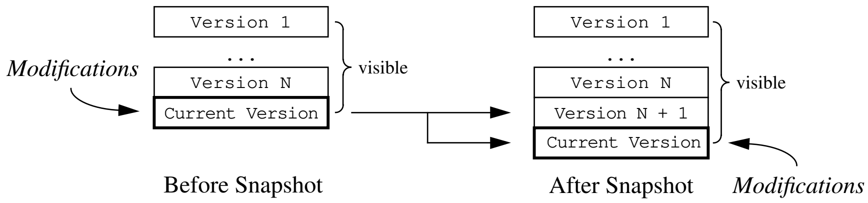

Figure 2: Deferred Update

In a deferred update scheme, the typical operations provided to the user are slightly different:

- *New State (S)*: duplicate the version *S* of the object and return the new version to the user
- *Discard State (S)*: discard version *S* (take the corresponding version out of the list)
- *Commit State (S)*: copy the contents of the version *S* of the object back over the contents of the parent version of *S* and then discard version *S* (take the corresponding version out of the list)

Update Strategies and Concurrency

It is highly advantageous to use a deferred update scheme when sharing recoverable objects between multiple tasks. Imagine a scenario where multiple tasks work on the same version of an object. Suddenly one of the tasks decides that it wants to revert to a previously saved state.

When using the in-place update scheme, calling *Restore Checkpoint* can lead to inconsistencies, because the other tasks don't know about the abort and will continue to work on the object as if it were still in the current state. To prevent such problems, the task must first communicate its decision to all other tasks. This synchronization of tasks is an additional burden for the application programmer.

With the deferred update scheme, the task that wants to revert to a previous version *S* can safely call *Discard State* on the current version, and then start working with version *S*, since all the versions of the object are visible. It can even create a new subversion of *S* if it wants to. In the mean time, the other tasks can continue with their work, and will be informed of the abort as soon as they try to commit their changes to the object.

In order to handle recovery correctly in the presence of concurrency, a recoverable object must know the number and identity of all the tasks that have manipulated it. It is the only way to know for instance when a version of the object can be discarded and the memory associated with it released. Hence, tasks must somehow register as users of a version of the recoverable object.

Rules for Recoverable Objects

Given these considerations, the following rules apply to shared recoverable objects:

- Several tasks can work with a version of the recoverable object concurrently. All but the task that created the version must explicitly register with the version as participants.
- Once a participant task has called *Discard State* on a version of the recoverable object, the version will be discarded regardless of the votes of the other participants.
- A version is considered committed and its state copied back to its parent only if *all* participant tasks call *Commit State*.
- A version can only have one non-discarded subversion at a given time. This means that it is possible to create a second subversion of a version only after at least one of the participants of the first version has called *Discard State*.
- Participants that disappear without calling *Commit State* or *Discard State* (deserters) are ignored. If the only participant of a version disappears, the version is discarded.

4.2 Interface

Following the previous arguments, this subsection describes the interface of the implementation of shared recoverable objects. The recoverable object class described below allows an application to transform arbitrary objects into recoverable objects. The package `Recoverable` declares an abstract tagged type `Recoverable_Object` with all the necessary functions for recovery:

```
package Recoverable is

   type Recoverable_Object is abstract tagged private;
   --  Although this type is not limited, assignment of versions
   --  is not allowed (Program_Error will be raised at run time)

   type Recoverable_Object_Ref is access all Recoverable_Object'Class;

   procedure New_State     (Object       : in Recoverable_Object_Ref;
                            New_Object   : out Recoverable_Object_Ref);

   procedure Join_State    (Object : in Recoverable_Object_Ref);

   procedure Commit_State  (Object : in out Recoverable_Object_Ref;
                            Sync   : in Boolean := False);

   procedure Discard_State (Object : in out Recoverable_Object_Ref);

   type Object_Status_Type is
      (Undecided, Discarded, Committing);

   function Current_Status ( Object : in Recoverable_Object_Ref)
      return Object_Status_Type;

   Object_Discarded,
   Already_Created_Subversion,
   Task_Already_Joined,
   Task_Never_Joined,
```

```
      Cannot_Join_Original,
      Null_Object : exception;
private
   ...
end Recoverable;
```

Application programmers must write their own recoverable objects containing application dependent data using type extension:

```
type Recoverable_Data is new Recoverable.Recoverable_Object with
   record
      Data : Application_Data;
      ...
   end record;
```

The abstract tagged type `Recoverable_Object` is non-limited, and therefore all derived types must also be non-limited. Hence, recoverable objects can not contain any limited components. This drawback can be overcome by making the abstract `Recoverable_Object` type limited, and requiring the user to provide a *copy* procedure for the extended data type, or, following the approach described in [8], having the user write *flattening* and *reconstitute* operations that allow the user data to be copied using streams.

In order to share a recoverable object between multiple tasks, it must obviously be accessed by reference. This is why the procedures that provide recoverability all use references.

In the following subsections, the operations that allow the manipulation of recoverable objects will be presented. Note that all of the procedures and functions will raise the exception `Null_Object` if the user passes a non-valid pointer as a parameter.

4.2.1 New_State

```
procedure New_State     (Object       : in Recoverable_Object_Ref;
                         New_Object   : out Recoverable_Object_Ref);
```

The `New_State` procedure must be used to establish a new version for the recoverable object `Object`. The object is duplicated, and a pointer to the new version is returned in `New_Object`. The task that calls this procedure is automatically added as a participant task for the new version of the object.

This procedure may raise the following exception:

- `Already_Created_Subversion` is raised if the version pointed to by `Object` has already a subversion associated with it.

4.2.2 Join_State

```
procedure Join_State    (Object : in Recoverable_Object_Ref);
```

If a task wants to participate in an already existing version, it must call `Join_State`, passing a reference to the version of the object. This will add the task to the list of participants.

The procedure may raise two different exceptions:

- `Task_Already_Joined` is raised if the calling task is already registered as a participant of the version pointed to by `Object`.

4.2.3 Commit_State

```
procedure Commit_State  (Object : in out Recoverable_Object_Ref;
                         Sync   : in Boolean := False);
```

By calling `Commit_State` a task states that it has successfully completed its modifications to the version of the object pointed to by `Object` and desires to commit these changes to the parent version. The parent version will be updated once *all* participating tasks have committed their changes. If the parameter `Sync` is set to true, the procedure will block the calling task until all other participant tasks have given their vote.

This procedure may raise the following exceptions:

- `Task_Never_Joined` is raised if the calling task is not a participant of the version pointed to by `Object`.
- `Object_Discarded` is raised if the parameter `Sync` is set to true and any of the other participating tasks calls `Discard_State` for the version pointed to by `Object`.

4.2.4 Discard_State

```
procedure Discard_State ( Object : in out Recoverable_Object_Ref);
```

With `Discard_State`, a task can cancel all its changes to the version of the object pointed to by `Object`. Once a participating task has called `Discard_State`, the state of the version will be discarded, regardless of the decisions of the other participating tasks.

The procedure may raise the following exception:

- `Task_Never_Joined` is raised if the calling task is not a participant of the version pointed to by `Object`.

4.2.5 Current_Status

```
function Current_Status ( Object : in Recoverable_Object_Ref)
   return Object_Status_Type;
```

This procedure allows a participant to get information about the current status of the version of the object pointed to by `Object`. Possible results are:

- `Discarded` — meaning that a participant has already called `Discard_State`,
- `Committing` — meaning that all other participants have called `Commit_State`,
- `Undecided` — otherwise.

5 Example

5.1 Introduction

The following section will present a small example application that manipulates digital images by applying so-called *filters*. Current digital image enhancement applications such as Adobe Photoshop® offer a huge variety of filters, including image sharpening or softening, stylizing, distortion, removal of dust and scratches, lightning, color adjustments, adding noise, applying textures, etc...

A filter can be seen as a function which is applied to each pixel of an image. The resulting pixel depends on the original pixel, the surrounding pixels, other global properties of the image and on parameters of the filter (i.e. for the *Gaussian Blur* filter, the radius of the pixel area is a parameter).

Some filters use very simple functions (i.e. inverting the picture), others can get very complicated. Image restoration filters such as edge detection or noise reduction filters use discrete Fourier transforms to decompose an image into periodic structures — moving the picture from the spacial to the frequency domain. This makes it easy to remove certain frequencies from the image, noise being typically in the high frequency range. Finally the spacial picture can be reconstructed using the inverse Fourier transform.

It is also much easier to calculate a convolution[1] in the frequency domain than in the spacial domain, since it comes down to multiplying the Fourier transform of the original image with the Fourier transform of the convolution matrix.

5.2 Performance Considerations

The general formula calculating the two-dimensional Fourier transform of a $M \times N$ matrix with complex components is the following [9]:

$$F(u, v) = \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} e^{-\frac{2\pi jmu}{M}} e^{-\frac{2\pi jnv}{N}} f(m, n)$$

Applied directly, this formula is prohibitively expensive. If $M = N$, each pixel of the image requires N^2 complex multiplications and $N^2 - 1$ complex additions, which gives us an algorithmic complexity of N^4 for the entire picture. Using mathematical transformations and precalculation of constants, the complexity can be reduced to $N^2 \log N$, which still represents a considerable calculation effort.

With the emergence of multiprocessor systems in the personal computer area, current image processing applications start to make use of the additional processing power to boost the performance of filters. The following programming example will illustrate how recoverable objects can be used in a multitasking environment to implement software fault tolerance and “undo” functionality.

5.3 Example Code

The first step is to define a type that can hold the image data. As shown in section 4.2, type extension must be used to obtain a recoverable object containing application data. The following package defines a new type `Recoverable_Picture` derived from the type `Recoverable_Object`, adding a matrix of pixels to hold the color information of the picture. The bounds of the picture can be specified at the moment of instantiation through discriminants.

```
with Recoverable; use Recoverable;

package Recoverable_Picture is

   subtype Color_Span is Natural range 0 .. 255;
   type Pixel is record
      Red   : Color_Span := 0;
      Green : Color_Span := 0;
      Blue  : Color_Span := 0;
      Alpha : Color_Span := 0;
   end record;

   type Picture is array (Integer range <>, Integer range <>) of Pixel;

   type Recoverable_Picture ( Top    : Integer;
                              Left   : Integer;
```

1. A convolution is basically a function where the output value of the pixel depends on its original value and the values of the surrounding pixels. Smoothing filters are typical convolution examples.

```
                                 Bottom : Integer;
                                 Right  : Integer)
         is new Recoverable_Object with record
            Image : Picture (Left .. Right, Top .. Bottom);
         end record;

      type Recoverable_Picture_Ref is access all Recoverable_Picture'Class;

   end Recoverable_Picture;
```

The main procedure that implements the "undo" functionality is presented below:

```
   with Recoverable;          use Recoverable;
   with Recoverable.Picture;  use Recoverable.Picture;
   with Ada.Text_IO;          use Ada.Text_IO;

   procedure Photoshop is

      Original_Picture : Recoverable_Picture_Ref;
      Filtered_Picture : Recoverable_Picture_Ref;

      type Square is record
         Top    : Integer := 0;
         Left   : Integer := 0;
         Bottom : Integer := 0;
         Right  : Integer := 0;
      end record;

      Number_Of_Processors    : constant Positive := 5;
      Image_Square            : constant Square := (-4, -3, 4, 3);

      task type Filter_Task;

      task body Filter_Task is separate;

   begin
```

Now, the image must be allocated.

```
      Original_Picture := new Recoverable_Picture
                            (Image_Square.Top, Image_Square.Left,
                             Image_Square.Bottom, Image_Square.Right);
```

The user of the application should now draw the contents of the image, or load an existing image from disk. Once he chooses to apply a filter to the image, a new version must be created.

```
      New_State   (Recoverable_Object_Ref (Original_Picture),
                   Recoverable_Object_Ref (Filtered_Picture));
```

The following block will create the tasks that actually perform the filtering. Note that the block is only exited when all of the `Filter_Tasks` have terminated.

```
      declare
         Tasks : array (1 .. Number_Of_Processors) of Filter_Task;
      begin
         null;
      end;
```

Now, we must check if the tasks have completed their work successfully. If not, the (probably incompletely) filtered picture must be discarded and an alert message should be displayed to the user.

```
      if Current_Status (Recoverable_Object_Ref (Filtered_Picture))
         /= Committing then

         Discard_State (Recoverable_Object_Ref (Filtered_Picture));
         Alert_User;

      else
```

Show the resulting picture to the user and ask him if he wants to keep the filtered picture, or undo the changes, then call `Commit_State` or `Discard_State` accordingly.

```
         if User_Accepts_Result then

            --  Make the changes definitive
            Commit_State (Recoverable_Object_Ref (Filtered_Picture));
```

```
         else
            --  Revert to the previous version
            Discard_State (Recoverable_Object_Ref (Filtered_Picture));
         end if;
      end if;
   end Photoshop;
```

The actual filtering is done in the body of the `Filter_Tasks`. Lets assume there is a function that implements the desired filtering for one pixel using the interface:

```
   function Filter_Pixel ( Source : Picture;
                           X, Y   : Integer)
      return Pixel;
```

A task that applies this filter on a square subregion of the image could be written the following way:

```
   task body Filter_Task is
```

First we must initialize the local variables. This will be done using calls to a protected object[1].

```
      Source_Picture   : Recoverable_Picture_Ref  := Get_Source_Picture;
      Dest_Picture     : Recoverable_Picture_Ref  := Get_Dest_Picture;
      Subregion        : Square                   := Get_Subregion;
   begin
```

Now we must tell the version of the recoverable object that we want to work with it.

```
      Join_State (Recoverable_Object_Ref (Dest_Picture));

      for Y in Subregion.Top .. Subregion.Bottom loop
         for X in Subregion.Left .. Subregion.Right loop
            Dest_Picture.Image (X, Y) :=
               Filter_Pixel (Source_Picture.Image, X, Y);
         end loop;
      end loop;
```

The filtering has been successful and we can commit the changes.

```
      Commit_State (Recoverable_Object_Ref (Dest_Picture));
   exception
      when others =>
         Discard_State (Recoverable_Object_Ref (Dest_Picture));
         raise;
   end;
```

Note that if the `Filter_Pixel` function raises an exception in one task, the current version of the picture is discarded, regardless of the decisions of the other participants. This avoids leaving the picture in an inconsistent state.

6 Implementation

This section will describe the basic structure of the implementation, and point out some of its subtleties.

1. The program code of the protected object is not shown here, but its implementation is straightforward.

From an implementation point of view, a recoverable object is essentially a doubly linked list of versions of the object. Each version contains a pointer to its parent and child version (if it exists), a field representing the current state of the version, and a list of the task IDs of all participant tasks (see fig. 3).

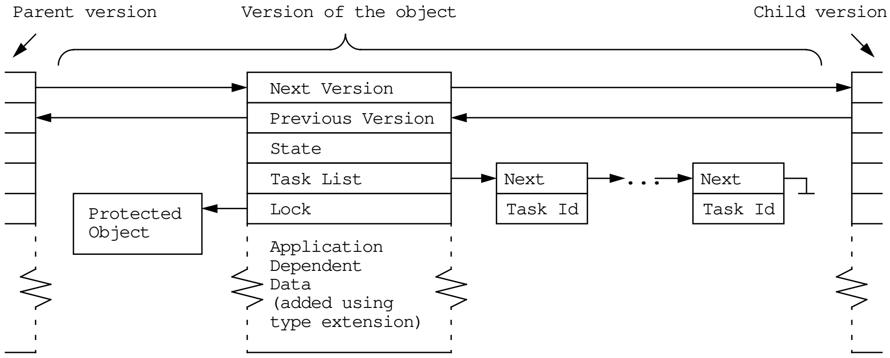

Figure 3: Recoverable Object Data Structures

Each version has a protected object providing simple `Lock` and `Unlock` operations associated with it, in order to protect it from concurrent accesses. It also defines an entry `Wait_For_Participants` that is used to block the participant tasks when they call `Commit_State` with synchronization.

```
protected type Recoverable_Object_Lock is

   entry Lock       ( L : in out Lock_Handler;
                      O : in Recoverable_Object_Ref);

   procedure Unlock ( L : in out Lock_Handler;
                      O : in Recoverable_Object_Ref);

   entry Wait_For_Participants;

end Recoverable_Object_Lock;
```

By using a separate lock for each version of the object, concurrency is increased. Attention must be paid in the implementation of the `Discard` and `Commit` procedures, for they have to remove a version from the linked list. In order to prevent deadlocks, the locks of the parent, the version to be removed and the child version must be acquired in top-down order. That way, the bottom-most request will always complete.

The protected object cannot be directly included in the tagged type (see fig. 3), for this would make the type limited. It is important that assignment of versions be possible, since it is the only way to make copies of the application data added to the type using type extension. Copying is needed in the `New_State` procedure, when the new version of the object must be initialized with the contents of the old version, and in `Commit_State`, where the contents of the old version are replaced with the contents of the new one.

A user of the package on the other hand should not be allowed to make copies of the entire version of an object, for this could corrupt the recoverable object. Simply copying a version does not correctly allocate a new protected object, and it corrupts the doubly linked list of versions.

These problems can be overcome by making the `Recoverable_Object` type a controlled type. The `Initialize` and `Finalize` procedures take care of allocating and deallocating the protected object. The `Adjust` procedure is used to disallow copying of the entire version object outside of the recoverable objects package. To achieve this, the recoverable object type contains a private boolean value, which must be set to true in order to allow an assignment to complete. The `Adjust` procedure raises an exception if the flag is not set properly.

```
procedure Adjust (Object : in out Recoverable_Object) is
begin
   if Object.Copy_Allowed then
      Object.Copy_Allowed := False;
   else
      raise Program_Error;
   end if;
end Adjust;
```

Note that according to the Ada Reference Manual [2, 7.6.1(14)] it is a bounded error to raise an exception inside `Adjust`, but [2, 7.6.1(16)] states that when `Adjust` is invoked as part of an assignment operation, any other adjustments due to be performed are performed, and then `Program_Error` is raised, which is exactly what we want.

The recoverable objects presented here are designed to handle concurrent accesses, and should be useful in various forms of concurrency control mechanisms like transactions or conversations. Tasks working on behalf of an atomic action for instance can be aborted when an exception is raised in an other participant of the atomic action. We must therefore also consider ATC (Asynchronous Transfer of Control) when implementing the operations of recoverable objects to prohibit corruption of the internal data structures.

First of all we must ensure that all locks held by a task are always released when an ATC occurs. This is done by saving the fact that an object is locked in a controlled object `Lock_Handler`. Before acquiring the lock, an instance of the `Lock_Handler` type must be declared. Then, a reference to it is passed to the protected operation `Lock`. This guarantees atomicity of actual locking *and* saving of the fact that the version has been locked, since protected operations are abort deferred regions. The `Finalize` procedure of the `Lock_Handler` then takes care of releasing the locks as soon as the `Lock_Handler` instance ceases to exist, even in the presence of raised exceptions or ATCs.

The second problem with ATC is that aborting a task that is manipulating the recoverable object may leave the internal data structures in an inconsistent state. To avoid this, the modifications of the internal data structures must be performed in abort deferred regions. Unfortunately, it is not possible to use a protected object for that purpose, since potentially blocking calls such as acquiring a lock are not allowed from within a protected operation. The only portable way to create abort deferred regions is again using controlled types: the `Initialize`, `Adjust` and `Finalize` procedures of controlled types are abort deferred regions.

The procedure for transforming an operation having in and out parameters into an abort deferred operation is the following:

Controlled types have to be declared at the library level. Hence it is not possible to access the in and out parameters of the operation directly. They must be passed inside the controlled type itself, and therefore the controlled type must be declared as follows:

```
type Operation_Executor_Type is new Controlled with
   record
      In_Parameter_1 : In_Parameter_1_Type;
      --  ...
      In_Paramter_N : In_Parameter_N_Type;
```

```
      Out_Parameter_1 : Out_Parameter_1_Type;
      --  ...
      Out_Parameter_N : Out_Parameter_N_Type;
   end record;
```

The `Initialize` procedure cannot be used to execute the actual statements of the operation, since it is not possible to affect the in parameters to the type before `Initialize` is called. The `Finalize` procedure is not suitable either, since it is not possible to retrieve the out parameters before the instance of the type is deallocated. Hence the `Adjust` procedure must be used:

```
procedure Adjust (O : in out Operation_Executor_Type) is
begin
   --  The actual statements of the operation go here
   --  All occurences of In_Parameter_X must be replaced by
   --  O.In_Parameter_X, Out_Parameter_X by O.Out_Parameter_X
end Adjust;
```

The actual operation must be changed to:

```
procedure Operation ( In_Parameter_1 : in In_Parameter_1_Type;
                      Out_Parameter_1 : out Out_Parameter_1_Type) is
   Source, Destination : Operation_Executor_Type;
begin
   --  Affect the In parameters
   Source.In_Parameter_1 := In_Parameter_1;
   --  The actual statements will be executed inside the Adjust procedure
   Destination := Source;
   --  Retrieve the Out parameters
   Out_Parameter_1 := Destination.Out_Parameter_1;
end;
```

The last problem due to ATC is the deserter problem. A task that has joined a version of a recoverable object by calling `Join_State` but then terminates (or is aborted) without calling `Discard_State` or `Commit_State` first, will compromise the correct functioning of the recoverable object, since the version cannot be discarded, nor committed, until all participant tasks have given their vote. It is therefore necessary to detect deserter tasks.

This can be done by using a controlled task attribute that stores all the versions of all recoverable objects that a task has joined so far. If the task is aborted or terminates before it has called `Discard_State` or `Commit_State` for these versions, the `Finalize` procedure of the controlled attribute will inform the recoverable objects that the task has deserted.

7 Conclusion and Future Work

This paper describes an implementation of recoverable objects that supports concurrency. Application programmers can write their own recoverable objects by extending the `Recoverable_Object` root type.

It has been shown that when designing recoverable objects that can be accessed concurrently, a deferred update scheme should be preferred over an in-place update scheme. Implementation problems due to concurrency and corresponding solutions have been presented. Special attention must be paid to task abortion, for this can lead to corrupt data structures and memory leaks (deserter problem).

Future work in this direction include adding persistence functions to recoverable objects, making it possible to save the state of an object to stable storage and (automatically) restore it after a crash failure. Plans are then to implement some kind of concurrent transactional service built upon these abstractions.

8 Acknowledgments

The authors gratefully acknowledge the contributions of Alexander Romanovsky in the preparation of this paper, and want to thank Thomas Wolf for his excellent implementation suggestions.

References

[1] Lee, P. A.; Anderson, T.: "Fault Tolerance - Principles and Practice". In *Dependable Computing and Fault-Tolerant Systems*, volume 3, Springer Verlag, 2nd ed., 1990.

[2] Taft, S. T.; Duff, R. A. (Eds.): *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*. Lecture Notes in Computer Science **1246**, Springer Verlag, 1997.

[3] Barnes, J. (Ed.): *Ada 95 Rationale*. Lecture Notes in Computer Science **1247**, Springer Verlag, 1997.

[4] Randell, B.: "System structure for software fault tolerance". *IEEE Transactions on Software Engineering* ***1**(2)*, pp. 220 – 232, 1975.

[5] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[6] Moss, J. E. B.: *Nested Transactions, An Approach to Reliable Computing*. Ph.D. Thesis, MIT, Cambridge, April 1981.

[7] Xu, J.; Randell, B.; Romanovsky, A.; Rubira, C. M. F.; Stroud, R. J.; Wu, Z.: "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". In *FTCS-25: 25th International Symposium on Fault Tolerant Computing*, pp. 499 – 509, Pasadena, California, 1995.

[8] Rogers, P.; Wellings, A. J.: "State Restoration In Ada 95: A Portable Approach to Supporting Software Fault Tolerance". *Technical Report YCS-98-297*, Department of Computer Science, University of York, York, UK, 1998.

[9] Jähne, B.: *Digital Image Processing, Concepts, Algorithms and Scientific Applications*. Springer, New York, 3rd ed., 1995.