Software Fault Tolerance
Implementing Backward Error Recovery

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University

Overview

- Simple Approach for Implementing Recovery Blocks
  - C++ [CS93]
- Designing Libraries
- Checkpoint
  - Ada [RW98]
- Recovery Cache
  - Ada [RW98]
- Recovery Block Template

Simple Recovery Blocks

- Make a copy of the original object(s) before running the algorithm that modifies their state [CS93]
- Two ways of implementing recovery blocks
  - In-place update
    - Make modifications on the original state, and replace it with a backup copy upon rollback
  - Deferred update
    - Make modifications to a copy, and replace the original upon acceptance

C++ In-place Update

```
T oldObject = object;
try {
  alternate(object);
  if (accept(object)) {
  return;
 }
} catch (..) {
 object = oldObject;
 continue;
}
```

Copy!

Requires one initialization / copy in the fault-free case

C++ Deferred Update

```
try {
 T newObject = alternate(object);
 if (accept(newObject)) {
     object = newObject;
  return;
 }
} catch (..) {
 continue;
}
```

Copy!

Requires initialization / copy and an assignment in the fault-free case

Discussion of the Simple Approach

- The object's state is copied entirely
  - Solution: redefine the copy operator
- The state is stored in "volatile" memory
- The programmer can make mistakes, since (s)he has to follow programming conventions
  - Create copies manually
  - Restore them manually
  - Make sure to handle all exceptions
    - What if (s)he forgets one?
- Idea
  - Design a library and enforce rules whenever possible

Desired Properties of Libraries (reminder)

- Ease of use of the Library
  - Minimize the amount of work a programmer needs to do by providing everything that is application-independent
- Safe use of the Library
  - Verify imposed programming conventions
  - Use static checking whenever possible, otherwise throw exceptions
- To achieve ease of use and safety of use, the library interface must be designed with great care!

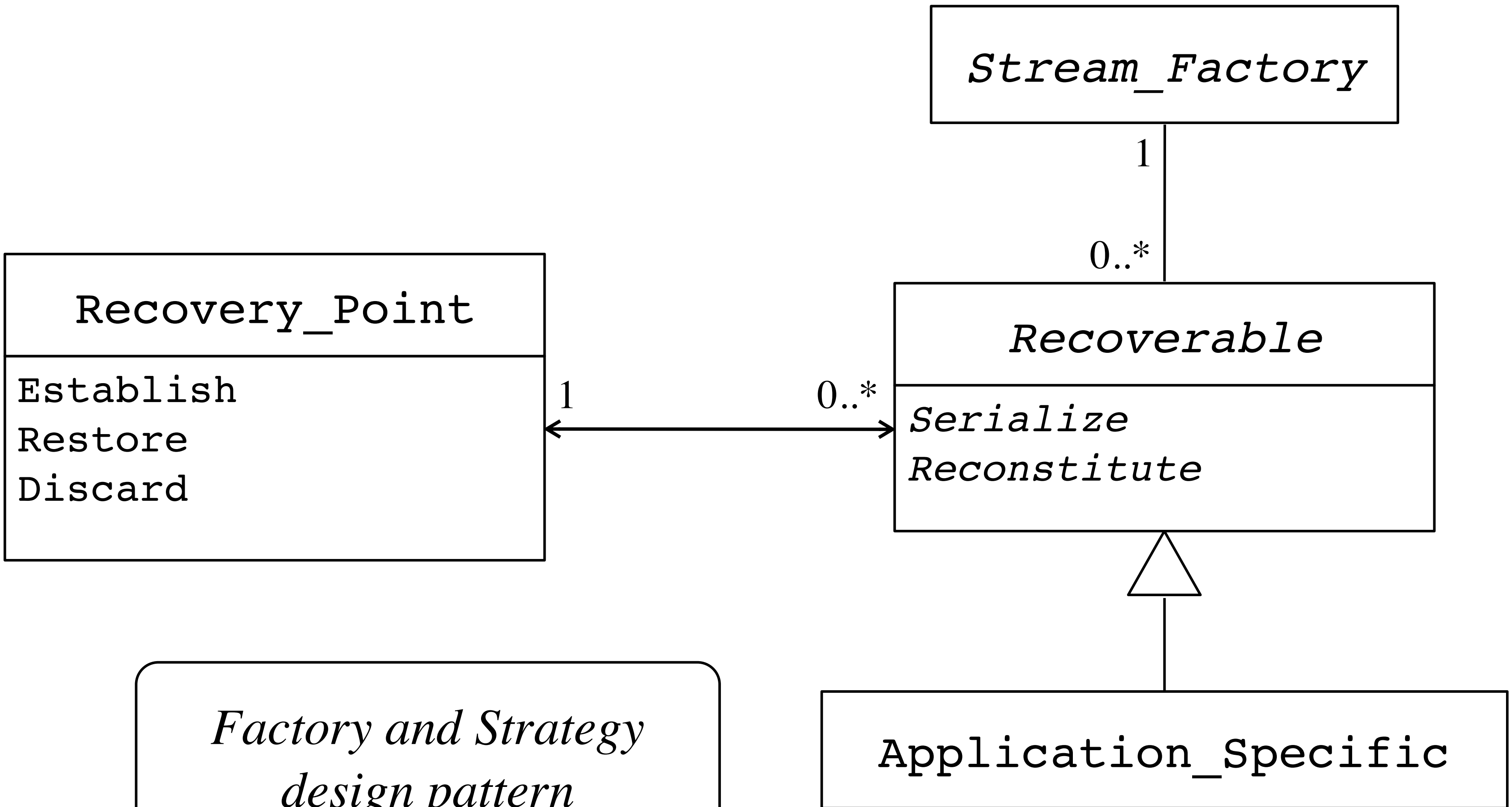
Checkpointing

- A checkpoint saves a complete copy of the (important) application state
- Implementation Requirements
  - Easy to use for a programmer
    - What does that mean for checkpointing?
    - What is generic? What is application-specific? What needs to be "configured" by the user?
  - Safe to use
    - What does that mean for checkpointing?
    - What needs to be enforced? What could go wrong?
  - Do implementation choices affect the interface?

Ada Implementation

- Recoverable objects are clearly identified
  - The application-specific objects that must be checkpointed have to inherit from the *Recoverable* class
- Object state is stored in streams
  - The storage used for saving the state of objects is customizable
- Fine-grained control over what state needs to be saved
  - The programmer can provide marshalling and unmarshalling operations to save only specific state to the stream
- Group objects that represent the relevant application state for a particular algorithm
  - A recovery point is a set of objects, whose state must be saved at certain points in time

Library Architecture



Factory and Strategy design pattern

Recovery Point Specification

```
type Recovery_Point is tagged limited private;

procedure Establish (This : in out Recovery_Point);

procedure Restore (This : in out Recovery_Point);

procedure Discard (This : in out Recovery_Point);

Level_Violation : exception;

Sharing_Violation : exception;
```

Specification of Streams

```
type Stream_Factory is tagged limited private;

type Any_Stream_Factory is access Stream_Factory'Class;

type Any_Stream is
   access Ada.Streams.Root_Stream_Type'Class;

function New_Stream (This : in Stream_Factory )
   return Any_Stream;

type Volatile_Stream_Factory is new Stream_Factory
   with private;

Default_Factory : aliased Volatile_Stream_Factory;
```

Specification of Recoverable

```
type Recoverable
 (Recovery_Object : access Recovery_Point'Class;
  Stream_Factory : Any_Stream_Factory := Default_Factory)
  is abstract tagged limited private;

type Any_Recoverable is access all Recoverable'Class;

procedure Serialize
    (Stream : access Ada.Streams.Root_Stream_Type'Class;
     This : in Recoverable) is abstract;

procedure Reconstitute
    (Stream : access Ada.Streams.Root_Stream_Type'Class;
     This : out Recoverable) is abstract;
```

Example Use

```
type Point is record
  x, Y, Z : Float;
end record;

type Recoverable_Plane is
  new Recoverable with record
  A, B, C: Point;
end record;

procedure Serialize
  (Stream : access Root_Stream_Type;
    This : in Recoverable_Plane) is
begin
  Point'Write (Stream, This.A);
  Point'Write (Stream, This.B);
  Point'Write (Stream, This.C);
end Serialize;
```

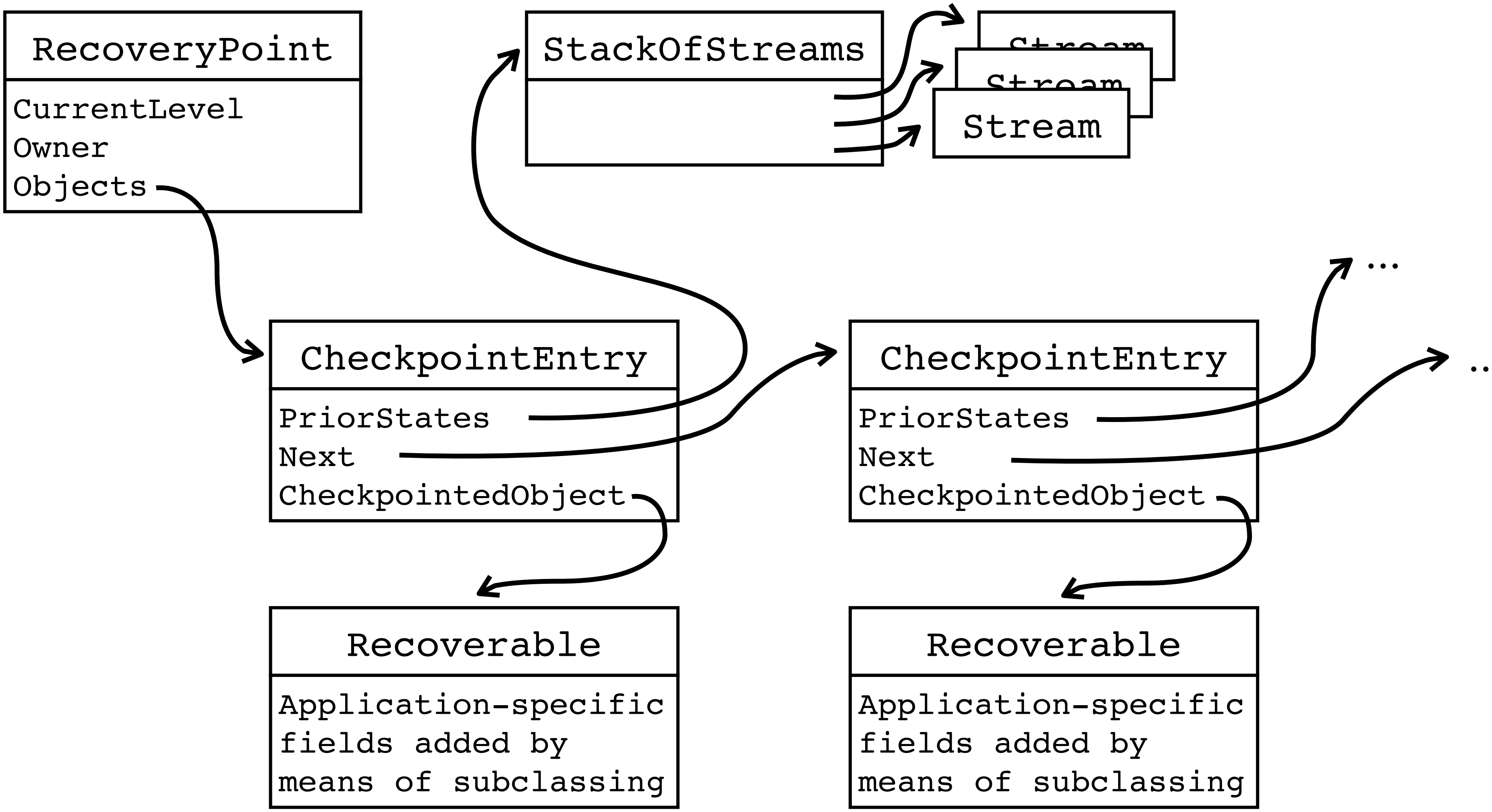
```
-- similar for Reconstitute

declare
  Prior_State :
    aliased Recovery_Point;
  Foo : Recoverable_Plane
    (Prior_State'Access);
  Bar : Recoverable_Plane
    (Prior_State'Access);
begin
  …
  Establish (Prior_State);
  …
end;
```

Implementation Concerns

- Enforce safety rules
  - Make sure that a recoverable object is associated to one and only one recovery point
    - Set up bidirectional association at declaration time
  - Make sure that recovery points are not "shared" among threads
    - Remember task id of first task that associates recoverable objects with the recovery point
  - Forbid associating a new object to a recovery point that is already in use
  - Make sure all memory is reclaimed when recovery point goes out of scope

Checkpoint Implementation Overview



Recovery Point Implementation

```
subtype Recovery_Level is Natural;
Empty : constant Recovery_Level := 0;

type Checkpoint_Entry;
type List is access Checkpoint_Entry;

type Recovery_Point is new
  Ada.Finalization.Limited_Controlled with record
 Current_Level : Recovery_Level := Empty;
 Owner : Ada.Task_Identification.Task_Id;
 Objects : List;
end record;

procedure Finalize (This : in out Recovery_Point);
```

CheckpointEntry Implementation

```
package Stack_of_Streams is
  new Unbounded_Stacks (Stream_Reference);

type Checkpoint_Entry is limited record
  PriorStates : Stack_of_Streams.Stack;
  Next : List;
  CheckpointedObject : Any_Recoverable;
end record;
```

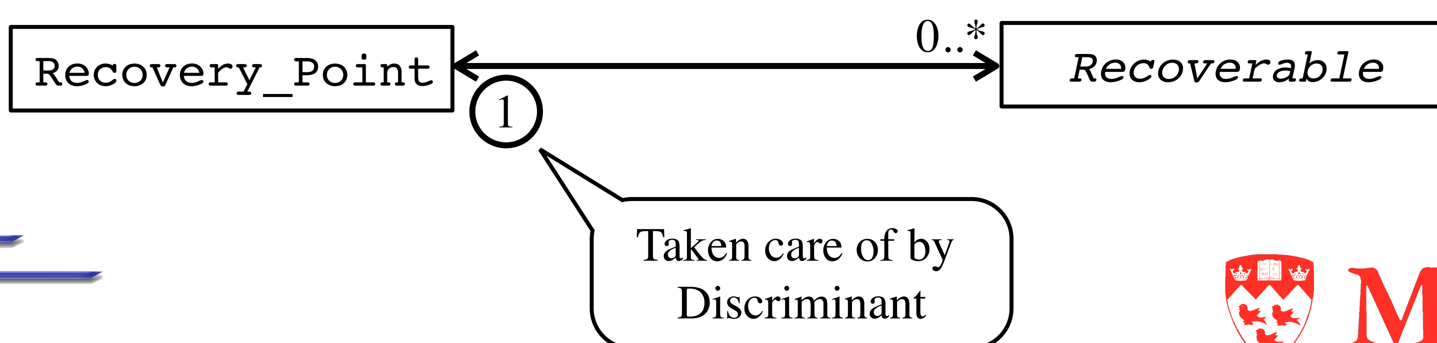
Recoverable Implementation

```
type Recoverable
  (Recovery_Object : access Recovery_Point'Class;
    Stream_Factory : Any_Stream_Factory := Default_Factory)
 is abstract new Ada.Finalization.Limited_Controlled
 with null record;

procedure Initialize (This: in out Recoverable);
procedure Finalize (This: in out Recoverable);
```

Safe Registration

- `Initialize` procedure of the recoverable object
  - Verify that the recovery point is empty, i.e. `CurrentLevel` = 0
  - Looks at the `Owner` field of the associated recovery point and sets the `id` to the one of the current task, or raises `Sharing_Violation` if it already has a different owner
  - Creates a new checkpoint entry for the object, points the `CheckpointedObject` field to the current recoverable object, and inserts the checkpoint entry into the `Objects` list of the recovery point



Establish

- Increment the current recovery level (if there are associated recoverable objects)
- For each checkpoint entry
  - A new stream instance is obtained by calling `New_Stream` of the stream factory associated with the recoverable object
  - The state of the object is stored in the stream by calling `Serialize`
  - The stream is pushed onto the `PriorStates` stack

Restore and Discard

- Restore
  - If the current recovery level equals empty, then raise `Level_Violation`
  - For each checkpoint entry
    - Restore the most recently saved state using the top stream in the `PriorStates` stack and the `Reconstitute` procedure
- Discard
  - No effect if the current recovery level equals empty, else decrement the level
  - For each checkpoint entry
    - Pop the most recently saved state from the `PriorStates` stack and deallocate the associated memory

Reclaiming Storage

(not a problem in garbage collected languages)

- `Finalize` of recoverable objects removes the corresponding checkpoint entry from the `Objects` list of the associated recovery point, and also frees any remaining prior states in the stack of streams
  - Very important if checkpoints and procedure nesting are not coordinated properly
- `Finalize` of recovery point goes though the entire data structure and reclaims all leftover storage

Performance Enhancement

- Previous checkpointing implementation saves and restores the state of all objects associated with the recovery point, even if they are not modified!
- Idea: Incremental checkpointing
  - Only objects whose values have been modified are checkpointed
- A Recovery Cache saves only the original state of objects that have changed after the latest recovery point
  - If nothing changes, nothing needs to be saved!

Recovery Cache Example (1)

```
X := 10;
Y := 20;
Establish(RP);
X := 30;
Establish(RP);
Y := 40;
X := 50;
Discard(RP);
```

Recovery Cache Stack

{X,10}	Level 1
{Y,20} {X,30}	Level 2

Recovery Cache Example (1)

```
X := 10;
Y := 20;
Establish(RP);
X := 30;
Establish(RP);
Y := 40;
X := 50;
Discard(RP);
Restore(RP);
```

Recovery Cache Stack

Recovery Cache Stack	
{X,10}	Level 1
{Y,20}	Level 2
{X,30}	

All information needed to restore level 1 is available!

Implementation Issues (1)

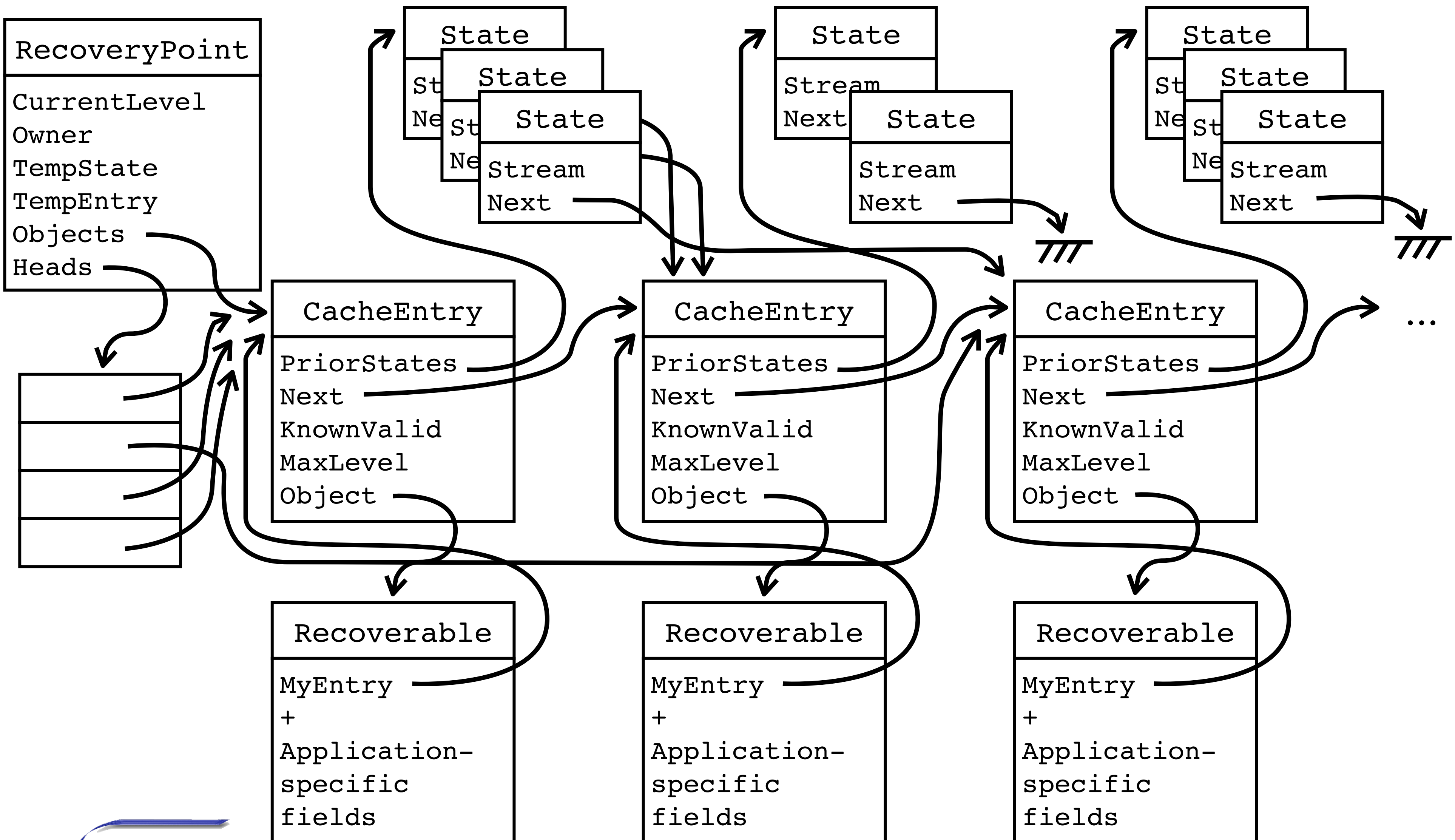
- Specification remains the same, except for `Recoverable`, which is not limited anymore in order to allow assignment
- In Ada, when an assignment statement `A := B` is executed
  - `Finalize` is invoked on A
  - Bitwise copy of all fields from B to A
  - `Adjust` is invoked on A
- Problems
  - Assignment copies the entire object
  - `Finalize` also called upon object destruction

When do we have to save the previous state?

Implementation Issues (2)

- Assignment copies the entire object
  - CacheEntry reference will be overwritten!
  - Solution: `Finalize` copies the reference into a `TempEntry` field of the recovery point, `Adjust` puts it back in place
- `Finalize` also called upon destruction
  - Solution: `Finalize` creates the new prior state, but stores a reference to it in a `TempState` field of the recovery point, `Adjust` then inserts it into the stack

Recovery Cache Impl. Overview



Establish and Restore

- Establish
  - Increment the current recovery level
    (if there are associated recoverable objects)
  - Create a new (empty) recovery region by pushing a new "head" on the stack
- Restore
  - Follow the top-most region pointer to all cache entries, and restore the value of the associated recoverable objects by calling `reconstitute`, using the most recent `Stream` as a parameter

Discard

- If there is only one level, discard it
- Else merge the level with the previous one
  - Insert each entry of the most recent level into the previous if the previous level does not contain the object already
    - Follow the previous head pointer, and, for each cache entry that is also part of the current level, i.e. `MaxLevel = currentLevel`, deallocate the most recent stream, and decrement `MaxLevel`

Implementing Recovery Blocks

(reminder)

```
ensure Acceptance Test
by Primary Alternate
else by Alternate 2
else by Alternate 3
…
else by Alternate n
else signal failure exception
```

Recovery Block Specification

- Generic procedure ensures safe use
- Works with checkpoint or recovery cache

```
type Alternate is access procedure;
type Alternates is array (Natural range <>) of Alternate;
type Acceptance_Test is access function return Boolean;

procedure Recovery_Block
 (Alternatives : in     Alternates;
  Acceptable   : in     Acceptance_Test;
  Prior_State  : in out Recovery_Point'Class);

RB_Failure, Sharing_Violation : exception;
```

Recovery Blocks Implementation

```
procedure Recovery_Block
  (Alternatives : in     Alternates;
   Acceptable   : in     Acceptance_Test;
   Prior_State  : in out Recovery_Point'Class) is
begin
  Establish (Prior_State);
    for A in Alternatives'Range loop
      begin
        Alternatives(A);
         if Acceptable then exit;
      exception
        when others => null;
      end;
       if A = Alternatives'Last then
         Discard(Prior_State);
         raise RB_Failure;
       else Restore (Prior_State);
       end if;
    end loop;
  Discard (Prior_State);
end Recovery_Block;
```

Recovery Blocks Use

```
declare
  RP : aliased Recovery_Point;
  Data : Recoverable_Plane (RP'Access);
  function Is_Horizontal return Boolean is …;
  procedure First_Try is …;
  procedure Second_Try is …;
  procedure Third_Try is …;
  My_Alternates : Alternates :=
   (First_Try'Access,
    Second_Try'Access,
    Third_Try'Access);
begin
 Recovery_Block
  (My_Alternates, Is_Horizontal'Access, RP);
end;
```

Recovery Block Discussion

- Ease of use
  - Programmer has to identify recoverable state
  - Both standard checkpointing and recovery cache can be used
  - No support for user-defined exceptions
  - The permanent association between a recoverable object and a recovery point could be made dynamic
    - Allows flexible, execution-dependent partitioning of recoverable state, e.g. defining subsets of recovery points for recursive recovery blocks
- Safety
  - Multithreaded use prohibited
  - Failure exception signals recovery block failure to the outside
- Implementation
  - The implementation of the recovery cache is slightly complicated due to Ada's way of doing `Finalize` and `Adjust`
    - A mapping between objects and cache entries based on a hash table might be more efficient

References

- [CS93]
  Calsavara, C. M. F. R; Stroud, R. J.: "Forward and Backward Error Recovery in C++", University of Newcastle upon Tyne, Technical Report No. 417, 1993.
- [RW98]
  Rogers, P.; Wellings, A. J.: "State Restoration in Ada 95: A Portable Approach to Supporting Software Fault Tolerance", University of York, Technical Report No. 298, 1998