Overview

• Design Issues
  • Encapsulating Input / Output Parameters
• Extensible Voters
  • Exact Majority Voter
• Version Execution
  • Exception handling
  • Non-pre-emption vs. pre-emption
• Interface for the Programmer
• Discussion
Desired Properties of Libraries (1)

- General purpose
  - Provide everything that is not application-specific
- Easy to use for a programmer
  - Can a programmer use your library without an excessive amount of work?
  - How invasive is your library?
  - Does the programmer need to change his existing design significantly to use your library?
  - Does using your library restrict the programmer in any way?
- Easy to use interface
  - Minimize required programming effort (algorithmic complexity / code quantity) to use the library
Desired Properties of Libraries (2)

• Safe to use
  • Do not give the programmer the opportunity to make mistakes!
    • If possible, correct use should be enforced by the compiler
    • Otherwise, use exceptions to signal misuse
  • Library must behave correctly, even if programmer uses “advanced” programming constructs such as multi-threading, exceptions, reflection, proxies, aspect-orientation, ...  

• Safe interface
  • Rely on generics / type checking
  • Signal (checked) exceptions if misuse is detected at run-time
N-Version Programming

• Application-specific
  • Input data
  • Output data
  • Code of the different versions

• Generic
  • Execution infrastructure
    • Threads that execute the versions
    • Synchronizing the threads
    • Distributing the input
    • Collecting the output
    • Voting
  • Voters

• The generic library needs to handle application-specific input and output, and needs to call application-specific code!
Application Data Requirements

• The infrastructure must be able to copy input parameter values
• The voters must be able to compare results
• Some voters might want to sort results
• Some voters might want to perform calculations on numerical results
• Ada does not provide reflection, e.g. some general means of dealing with unknown data
  \[ \Rightarrow \] we’ll use an object hierarchy
Application Data Hierarchy

- The programmer must implement a class that encapsulates his data as a subclass of `ApplicationData` or `ComparableData` and implement the required operations, or just use `NumericalData`.

```plaintext
ApplicationData
  Copy
  "="

ComparableData
  "<"

NumericalData
  NtoFloat
  FloatToN
```
Voter Requirements

• Voter implementation is application independent ⇒ make voters reusable!

• Which voter is most appropriate for determining a correct result is highly application dependent ⇒ the user should be able to configure the n-version support with the appropriate voter ⇒ if needed, the user should be able to write his own voter and use it with the infrastructure

• Voters are accessed concurrently!
Parallel Design Diversity Concept

Voter

SubmitResult
WaitForResult
Vote

ExactMajorityVoter  MeanVoter  ...  UserDefinedVoter
Voter Class Specification (1)

package Voters is
  type Voter (Number_OfVersions : Positive) is
    abstract tagged limited private;
  type Any_Voter is access all Voter’Class;
  procedure Submit_Result(Voter : in out Voter;
    R : in Application_Data; N : Positive; Round : Positive);
  procedure Wait_For_Result(Voter : in out Voter; R : out Application_Data);
  Decision_Failure : exception;
private
  ... -- Synchronizer defined on the next slide
  type Result_Array is array (Positive range <>) of ApplicationData;
  type Voter(Number_OfVersions : Positive) is tagged record
    Results : Result_Array (1 .. Number_OfVersions);
    Collected_Results : Natural := 0;
    Sync : Synchronizer;
  end record;
  procedure Vote (Voter : in out Voter; Result : out Positive) is abstract;
end Voters;
Voter Class Specification (2)

package Voters is
   ... -- continued from previous slide
private
   protected type Synchronizer is
      procedure Submit_Result(R : in Application_Data;
         N : in Positive; Round : in Positive;
         Voter : Any_Voter);
      entry Wait_For_Result(R : out Application_Data;
         Voter : Any_Voter);
      procedure Set_Result (N : in Natural);
      procedure Signal_Failure;
   private
      Chosen_Result : Natural := 0;
      Current_Round : Natural := 0;
      Failure : Boolean := False;
   end Synchronizer;
   ... -- contents on previous slide
end Voters;
package body Voters is

procedure Submit_Result(Voter : in out Voter;
    R : in Application_Data;
    N : in Positive;
    Round : in Positive;) is
begin
    Voter.Sync.Submit_Result (R, N, Round, Voter);
end Submit_Result;

procedure Wait_For_Result (Voter : in out Voter;
    R : out Application_Data) is begin
    Voter.Sync.Wait_For_Result (R, Voter);
end Wait_For_Result;

end Voters;
package body Voters is

protected body Synchronizer is

procedure Submit_Result (R : in Application_Data; N : Positive;
                         Round : Positive; Voter : Any_Voter) is

begin
  if Round = Current_Round then
    Copy(R, Voter.Results(N));
    Collected_Results := Collected_Results + 1;
    Voter.Vote;
  end if;
end Submit_Result;

entry Wait_For_Result (R : out Application_Data; Voter : Any_Voter)
  when Chosen_Result > 0 or Failure is

begin
  if Failure then Failure := False; raise Decision_Failure;
  else Copy(Voter.Results(Chosen_Result), R); Chosen_Result := 0;
  end if;
end Wait_For_Result;
procedure Set_Result (N : Natural) is begin
    Chosen_Result := N;
    Current_Round := Current_Round + 1;
end Set_Result;

procedure Signal_Failure is begin
    Failure = True;
end Signal_Failure;

end Synchronizer;
end Voters;
Dynamic Majority Voter Specification

```plaintext
package Voters.Majority_Voters is
  type Majority_Voter (Number_Of_Versions : Positive) is
    new Voter with private;

private
  type Majority_Voter (Number_Of_Versions : Positive) is
    new Voter with record
      EqualCount : array (1 .. Number_Of_Versions) of Positive;
    end record;

  procedure Vote (Voter : in out Majority_Voter;
    Result : out Positive);

end Voters.Majority_Voters;
```
Dynamic Majority Voter Implementation

procedure Vote (Voter : in out Majority_Voter;
   Result : out Positive) is
   Majority : Natural = Voter.Number_Of_Versions / 2;
begin
   if Voter.Number_Of_Versions mod 2 = 1 then
      Majority := Majority + 1; end if;
   if Voter.Collected_Results >= Majority then
      Voter.EqualCount := (others => 0);
      for I in 1 .. Voter.Number_Of_Versions - 1 loop
         for J in I + 1 .. Voter.Number_Of_Versions loop
            if Equal (Voter.Results(I), Voter.Results(J)) then
               Voter.EqualCount(I) := Voter.EqualCount(I) + 1;
               if Voter.EqualCount(I) >= Majority then
                  Voter.Sync.Set_Result (I); return;
               end if;
            end if;
         end loop; end loop;
      if Voter.Collected_Results = Voter.Number_Of_Versions then
         Voter.Sync.Signal_Failure;
      end if;
   end if;
end Vote;
N-Version Implementation

• Abstraction of a version

```pascal
type Version is access procedure
  (Input : in ApplicationData;
   Result : out ApplicationData);

type Versions is array (Natural range <>) of Version;
```

• Each version is executed by a task. For efficiency reasons, one task is created for each version, and kept alive for successive runs.

```pascal
task type Version_Executor
  (Version_Number : Natural;
   My_Control : access Version_Controller;
   My_Voter : access Voter) is
  entry Start (I : in Application_Data; R : in Positive);
end Version_Executor;
```
Exception Handling

- Should exceptions be treated as valid results, or should an unhandled exception be treated as a version failure?

If user-defined exceptions are to be supported, exceptions should be treated as valid results.

We decided to encapsulate exceptions in the ApplicationData hierarchy.
N-Version Specification

generic
  Number_Of_Versions : Positive;
  Algorithms : Versions (1 .. Number_Of_Versions);
  Voter : Voters.Voter;
package N_Version_Support is
  type N_Version is limited private;
  procedure Execute (N : in out N_Version;
                     Input : in Application_Data;
                     Output : out Application_Data);
  Decision_Failure : exception renames Voters.Decision_Failure;
private
  type N_Version is new Ada.Finalization.Limited_Controlled
    with record
      Version_Executors : array (1 .. Number_Of_Versions)
        of Version_Executor;
      Control : aliased Version_Controller;
      Voter : aliased Voter;
    end record;
  procedure Initialize (N : in out N_Version);
  procedure Finalize (N : in out N_Version);
end N_Version_Support;
N-Version Implementation (1)

```vход
package body N_Version_Support is
    procedure Initialize (N : in out N_Version) is
    begin
        for I in N.Version_Executors loop
            N.Version_Executors (I) := new Version_Executor
                (I, N.Control'Access, N.Voter'Access);
        end loop;
    end Initialize;

    procedure Finalize (N : in out N_Version) is
    begin
        for I in N.Version_Executors loop
            Abort (N.Version_Executors (I));
        end loop;
    end Finalize;

    ... -- continued on next slide
end N_Version_Support;
```
N-Version Implementation (2)

package body N_Version_Support is
    ... -- previous slide

procedure Execute (N : in out N_Version;
    Input : in Application_Data;
    Round : in Positive;
    Output : out Application_Data) is

begin
    for I in N.Version_Executors loop
        N.Version_Executors(I).Start (Input, Round);
    end loop;
    N.Voter.Wait_For_Result (Output);
end Execute;
end N_Version_Support;
Non-Preemptive Version Execution

• Each version runs to completion, and then submits its result to the voter

• Advantages
  • Simple to implement
  • The state of the versions remains consistent

• Disadvantages
  • Wasted time (sometimes a decision can be made without waiting for all results)
  • “Endless looping” versions will lead to failure
Preemptive Version Execution

• Still running versions are interrupted / notified as soon as a correct result has been determined

• Advantages
  • No wasted time
  • Can handle “endless looping” versions

• Disadvantages
  • Needs special language or OS support
  • Often results in high run-time overhead in fail-free mode
  • Consistency problems
  • Hard to prove correctness
Preemptive Version Controller

```plaintext
protected type Version_Controller is
  entry Wait_Abort;
  procedure Signal_Abort;
private
  Abort_Signaled : Boolean := False;
end Action_Controller;

protected body Version_Controller is
  entry Wait_Abort when Abort_Signaled is
    begin
      if Wait_Abort'Count = 0 then
        Abort_Signaled := False;
      end if;
    end Wait_Abort;
  procedure Signal_Abort is
    begin
      Abort_Signaled := True;
    end Signal_Abort;
end Version_Controller;
```
task body Version_Executor (Version_Number : Natural,
    My_Control : access Version_Controller;
    My_Voter : access Voter) is
    Input, Result : Application_Data; Round : Positive;
begin
    loop
        accept Start (I : in Application_Data; R : in Positive) do
            Copy (I, Input); Round := R;
        end Start;
        select
            My_Control.Wait_Abort;
        then abort
            begin
                Algorithms(Version_Number) (Input, Result);
                Submit_Result (My_Voter, Result, Round, Version_Number);
            exception
                when E : others =>
                    Submit_Result (My_Voter, new Exceptional_Outcome
                        (Exception_Identity (E)), Round, Version_Number);
            end;
        end select;
    end loop;
end Version_Executor;
Voter Synchronizer with Exceptions

package body Voters is

protected body Synchronizer is

procedure Submit_Result (...) is
begin
  -- same as before
end Submit_Result;

entry Wait_For_Result (R : out Application_Data; Voter : Any_Voter)
  when Chosen_Result > 0 or Failure is
begin
  if Failure then Failure := False; raise Decision_Failure;
else
  Copy(Voter.Results(Chosen_Result), R); Chosen_Result := 0;
  if R in Exceptional_Outcome’Class then
    Raise_Exception(R.Exception_Identity);
  end if;
end if;
end Wait_For_Result;
User-Defined N-Version Unit

with N_Version_Support;

type Element_List is new Application_Data with ...

procedure Bubble_Sort (Input : in Element_List;
                         Result : out Element_List);

procedure Shaker_Sort ...;
procedure Quick_Sort ...;

Sorting_Algorithms : Versions :=
    (Bubble_Sort’Access, Shaker_Sort’Access, Quick_Sort’Access);

package My_N_Version is new N_Version_Support
    (3, Sorting_Algorithms, Majority_Voter);

declare
    Reliable_Sort : My_N_Version.N_Version;
    Input : Element_List := new Element_List (...);
    Result : Element_List;
begin
    -- build input
    Reliable_Sort.Execute(Input, Result);
end;
Discussion (1)

• Ease of use
  
  + All application independent code is provided by the infrastructure, e.g. the programmer only has to implement the code for each version
  + The infrastructure provides all well-known voters, and allows the programmer to implement a custom voter, if necessary
  + Can be used as is in a recursive context
  - The programmer has to encapsulate input parameters and results in subclasses of ApplicationData
Discussion (2)

• Safety of use
  • Initialization of the n-version support is done automatically at instantiation time
  • Thread creation / destruction / synchronization is handled exclusively by the infrastructure
  • The infrastructure deals with user-defined and unhandled exceptions
  • The interface enforces coherence, e.g. the number of versions N always corresponds to the number of implemented procedures
  • The interface enforces correct voting, e.g. every version votes exactly once
  • Thread-safe
  • Failure to determine a result is signalled to the outside