Diversely Designed Classes for Use by Multiple Tasks

Alexander Romanovsky

Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK
email: alexander.romanovsky@newcastle.ac.uk
tel: +44+191+222+81+35

This paper proposes a new N-version programming (NVP) scheme which allows several caller tasks to jointly use components which are designed diversely. Diversity is applied here at the level of classes in such a way that several version classes (objects) are developed separately and independently, and are encapsulated into a diversely designed object. Such objects are to be implemented in a special stylised way to incorporate a controlling mechanism which would deal with task and version synchronisation, adjudication of version output parameters and states, faulty version recovery, etc. The general approach is demonstrated using Ada. We outline the characteristics of applications which benefit from using such NVP scheme, discuss the engineering of diversely designed objects and of the software which uses them and describe several possible extensions of the scheme.

Keywords: software design diversity, N-version programming, object-orientation, re-use, Ada

1. Introduction

There are many reasons why mistakes are made in designing computer systems. Unfortunately, experience confirms the idea, which has been obvious to some researchers for years, that it is impossible to deliver realistically large and complex software which is faultfree. To this end many software fault tolerance techniques have been developed which employ software diversity to overcome design faults. Two approaches were proposed in the 70-ies and have been investigated thoroughly since then: *recovery blocks* [1] and *N-version programming* [2]. We do not intend to compare them here (for that, readers are referred to [3]), but we would like to note that each of these approaches has its own pros and cons, has been used in practice several times, and that, generally speaking, they are the only general ways to secure tolerating software design faults. To apply the recovery block scheme, programmers have to develop several versions (alternates) of a program block and an acceptance test to check the correctness of the execution of any version. The first version (the primary alternate) is run and its results are checked by the acceptance test. If they are not ensured, the state of the program is rolled back to the state it had when the block started, and the next version is run. This continues until a version produces results ensuring the acceptance test. Note that versions are tried sequentially and that one needs state restoration features for this technique. Recently an Ada recovery block scheme [4] has been proposed which uses a specially-developed state restoration feature.

In this paper we focus on N-version programming (NVP). In this approach, N *versions* of a program (or, a module) are developed independently by different programmers, to be run concurrently. Their results are compared by an *adjudicator*. The simplest way is to use majority voting here: the results produced by the majority of versions are assumed to be correct, the rest of the versions are assumed to have errors (i.e. their faults to have been triggered in the execution). This technique requires a special support which controls the execution of versions and of the adjudicator and passes the information among them (we refer to it as the *controller*). In particular, it synchronises the version execution and obtains information (e.g. output results) from all of them to pass to the adjudicator.

Independent version design is vital in applying NVP because version designers tend to make similar mistakes which can cause several versions to fail on the same inputs. Special methodologies to help ensure this independence are proposed in [5].

A general framework for applying diversity in object-oriented (OO) systems [6] clearly demonstrates that NVP suits class diversity better, while recovery blocks are easier to apply to implementing method diversity. Versions, adjudicators and the controller are classes in this framework. Diversity is hidden inside a diversely designed class, and versions have interfaces identical to its interface. The applicability of the general framework is demonstrated in the C++ language (although the authors had to resort to an unspecified underlying mechanism to allow concurrent execution of versions). This implementation is presented as a set of re-usable classes. The first approach [7] to introducing class diversity was developed for the Arche language. This scheme does not hide diversity, so the user has to program many functions of the controller; in particular, s/he explicitly declares version objects and calls version objects and the adjudicator. The approach relies on an experimental run-time support which takes care of version concurrency and synchronisation; among other things, this support has a special feature for concurrent call of version methods.

2. Class Diversity in Ada

In their paper published in 1985, L.Strigini and A.Avizienis [8] emphasise the importance of developing various supporting mechanisms for NVP. Several experimental settings have been developed since then (DeDiX [2], DELTASE [9], Arche [7]). Unfortunately, none of them can be directly used in practice; not only because of their experimental nature but because they were built on top of OSes which are not used any more or because they are not readily available for practitioners. Our decision here is to follow B.Randell's idea; he argues the importance of using existing practical languages for implementing diversity schemes ([10]); in particular, he emphasises the usefulness of advanced OO programming features which can considerably facilitate scheme re-use and help with diverse system structuring.

Our class diversity scheme [11] relies on the general framework previously developed at Newcastle University [6]. The intention is to develop a NVP scheme suitable for concurrent OO languages: we want to explicitly address difficult questions of version synchronisation and concurrency and to develop re-usable components to perform these functions. For many reasons, it was Ada that was chosen to demonstrate and to experiment with the framework . Ada (Ada 95) is the first standard mainstream OO language which has features for concurrent programming. The expressiveness and power of its concurrency features is superior to those of other concurrent programming languages: it successfully combines features of both process-oriented and object-oriented concurrent programming. By demonstrating our scheme in Ada we are able to propose general solutions and to check them, so that they would be useful for developing similar schemes in other environments and languages. Most importantly, Ada itself is used in many industrial applications with high dependability requirements (e.g. those found in the aerospace industry).

In our scheme [11] each diversely designed (DD) class is implemented independently by N application (version) programmers. We assume that the initial abstract specification of the class is given. For example:

```
package list_class is
       type list_t is abstract tagged limited null record;

       procedure Init (l : access list_t) is abstract;
       procedure Add (l : access list_t; elem : in elem_t) is abstract;
       procedure Del (l : access list_t; elem : in elem_t) is abstract;
       procedure Get_Min (l : access list_t; elem : out elem_t) is abstract;
       procedure Sort (l : access list_t) is abstract;
       function Check (l : access list_t; elem : in elem_t)
                                         return boolean is abstract;
       list_failure : exception;
end list_class ;
```

The concrete DD class inherits from this class by adding a protected object synchronising N internal tasks and by implementing the controlling code in all application methods. This implementation comprises the NVP controller which works with N version objects and the adjudicator (an Ada protected object) declared inside the DD object (see Figure 1). N internal service tasks are forked when any method of the DD class is called: each task calls the corresponding method of a version object (Figure 2). Each method of the DD object has the same structure incorporating declarations on N service tasks and the same sequence of operations which includes calls of the synchronising object, version objects and the adjudicator. Version classes inherit from the initial class (`list_class`, in our example) and make it concrete by implementing all application methods. Programmers of different types are involved in developing DD objects: the system programmer designs the synchronising object and templates for developing its methods, the application programmers develop version objects, the fault tolerance programmer develops the adjudicator object and assembles all components of the DD class together following the templates proposed. Their responsibilities are clearly separated in our framework. Our scheme incorporates features for faulty version recovery and for advanced error detection based on version state adjudication [11-13]. To allow this, the fault tolerance programmer designs an *abstract stat*e of the DD object (a general description of the object state), which a version programmer will use to develop two mapping functions to get the current abstract state of the version and to map the proven correct abstract state of a healthy version onto the current internal state of the version. The functionalities of the adjudicator are extended by including version state comparison. Re-use comes in different flavours in this framework: synchronising objects are the same for all DD objects, inheritance is used to develop the specification of version classes and of the DD class itself, the implementation of the DD object and of the adjudicator follows the templates proposed, etc.

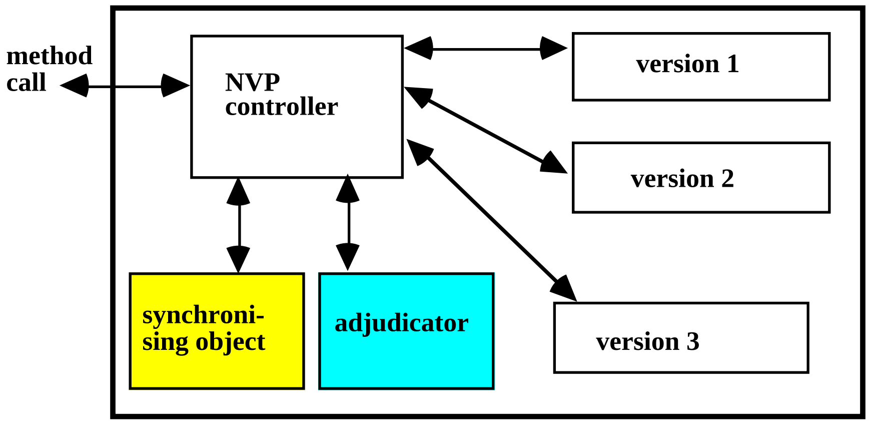

Figure 1. The architecture of the diversely designed object

To the best of our knowledge, the only existing Ada NVP scheme is the concurrent corresponding module scheme [14], which is developed for Ada 83 and which adheres to procedure diversity. It extends the original NVP scheme by allowing versions (all of which are produced from the same specification) to be in one of the following categories depending on their relation to the primary version: duplicate, reciprocal, residual; this requires a more complex version control than the conventional NVP. We believe that this scheme is important because it hides diversity inside the diversely designed procedure and is presented using an existing practical language. Our analysis shows, however, that the scheme, apart from adhering to procedure diversity rather than class diversity, has other disadvantages. First, it does not clearly separate the work of programmers of different types which are to be involved in design or address re-usability issues. Secondly, it does not properly address problems of informing the caller of failures and of returning output parameters. Lastly, from our point of view, using task abort as a regular (rather than exceptional) means is conceptually wrong.

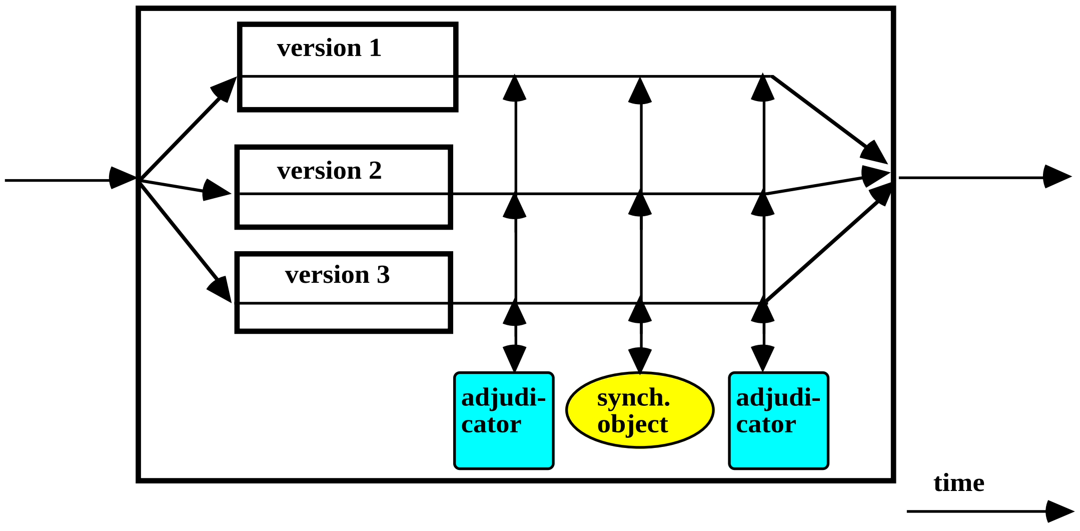

Figure 2. Execution of the diversely designed object

3. Motivations

The existing OO NVP schemes which are oriented towards practical languages [6, 11, 15] assume that there is only one caller requesting the results from a DD object. This is not the best approach to many practical situations for several reasons. First of all, it relies on concurrency forking/joining inside a DD object which can be expensive, in particular when there are many DD object calls (in which case using replicated tasks which call the same DD object together gives a better performance). Secondly, it can be important for some applications to allow several tasks to call a DD object in a synchronised fashion when they all need its results. In this case, each of multiple callers is able to use reliable results in its own way and to perform other computation when it is not involved in calling the DD object. This offers more flexibility and allows us to better utilise system resources. Thirdly, it is well known that it is often difficult to formalise and to properly use dynamic task creation, as well as to analyse systems which dynamically create tasks [16]. To answer all the concerns above, the DD object and, in particular, the controller should be able to allow N tasks to issue N replicated calls of any of its methods. This should be supported by appropriate caller task synchronisation inside each DD object. Note that experimental settings (Arche [7], DeDiX [2], DELTASE [9]) allow multiple/replicated tasks to use DD software but, as we have mentioned before, they cannot be easily used in practice as applications can run on only one experimental platform and they are not programmed at the concurrent language level.

In this paper we will demonstrate how DD objects should be developed to be able to serve N caller tasks. We will address this problem in general terms and discuss an Ada implementation: a re-usable OO scheme which separates the responsibilities of different programmers and adheres to principles of structured system design will be presented. Our particular emphasis is on discussing the functionalities of the supporting software and its re-usability. Although the new scheme bears a lot of similarity to our initial scheme discussed in Section 2, we believe that it is better to give its complete description, which is easy to understand and to use, rather than present it as a modification of the initial scheme; the two schemes are compared in Section 6.

4. The Scheme

We say that a diversity scheme follows the principles of *structured diverse programming* [13] if design diversity can be applied to the main structuring units of system design, diversity is encapsulated inside such units, the scheme can be applied recursively (e.g. it allows unit nesting) [17], the scheme supports independent design of version units, the responsibilities of the programmers of different types (version/application, system and fault tolerance programmers) are clearly separated. All this makes it easier to deal with system complexity, facilitates the management of diverse software and promotes re-use.

In our new scheme diversity is introduced at the class/object level: each diversely designed object has versions, the adjudicator and the synchronising software, which are hidden inside. This object is to be called by N concurrent caller tasks: it synchronises these tasks and calls N version objects. We rule out other approaches, within which each task calls its own version object, because this exposes diversity and is more difficult for the programmers of caller tasks to use.

When N caller tasks call a method of a DD object, the controller synchronises these N calls using a synchronising object: when all N of them have arrived, it allows them to continue (Figure 3). At the next step each of them calls one of the version objects and passes the output results and the abstract state of this version to the adjudicator. The synchronising software gets all tasks synchronised after these calls have been completed. Afterwards each task requests the correct output results from the adjudicator and returns them back to the caller. If the adjudicator cannot reach a consensus, an interface failure exception is signalled to all caller tasks.

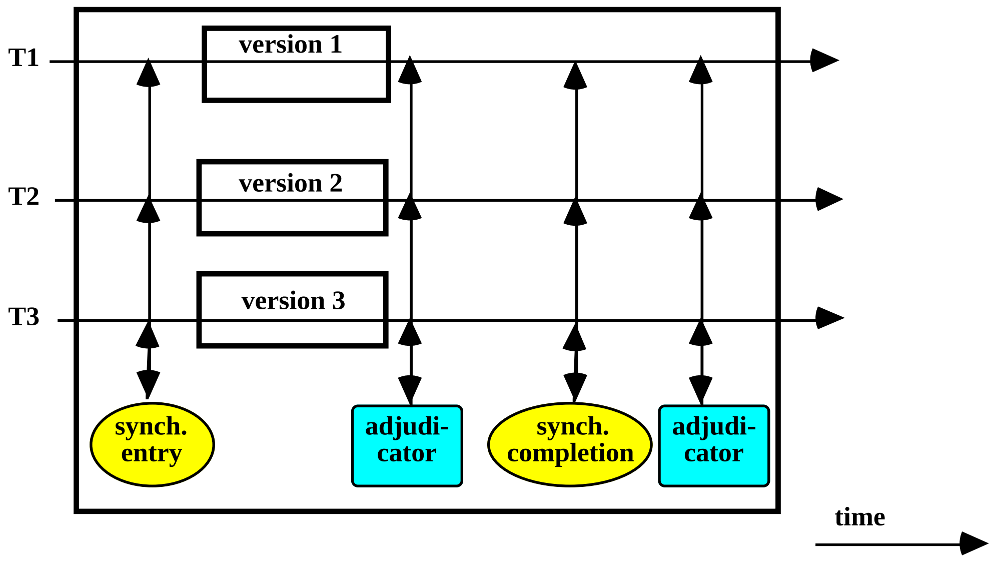

Figure 3. Execution of a diversely designed object by multiple tasks (T1, T2, T3)

The original abstract class (e.g. that in Section 2) has to be extended for the version classes by adding two new methods for comparing version states and for the faulty version recovery:

```
package list_class_version is
        type list_version_t is abstract tagged limited private;

        procedure Give_State (l : access list_version_t ;
                            state : out abstract_list_state_t) is abstract;
        procedure Correct_State (l : access list_version_t ;
                                state : in abstract_list_state_t) is abstract;
 private
        type list_version_t is abstract new list_t with null record;
end list_class_version ;
```

The design of the application methods of versions is in no way affected by the fact that these objects will be used as versions of the DD object, although these version objects are to be declared inside the DD object. The DD class `ft_list_class` is inherited from the initial abstract `class_list` class:

```
package ft_list_class is
        type ft_list_t is tagged limited private;
        procedure Init (l : access ft_list_t);
        procedure Add (l : access ft_list_t; elem : in elem_t);
        procedure Del (l : access ft_list_t; elem : in elem_t);
        procedure Get_Min(l : access ft_list_t; elem : out elem_t);
        procedure Sort(l : access ft_list_t);
        function Check(l : access ft_list_t; elem : in elem_t) return boolean;
private
        type ft_list_t is new list_t  with
                record
                      nvp_s: nvp_synchroniser_t(version_max);
                end record;
end ft_list_class ;
```

The NVP control is separated from version objects. The parameterised protected type `nvp_synchroniser` synchronises caller tasks when they are inside the DD object before and after calling version objects (Figure 3):

```
package nvp_synchroniser is
        protected type nvp_synchroniser_t (N : Positive) is
                entry caller_entry (your_number : out version_number);
                entry version_finish;
        private
                entry synch_entry (your_number : out version_number);
                entry synch_exit;
                active : integer := 1;
                let_go : boolean := false;
   end nvp_synchroniser_t;
end nvp_synchroniser;

package body nvp_synchroniser is
        protected body nvp_synchroniser_t is
                entry caller_entry (your_number : out version_number)
                                        when true is
                begin
                        active := active+1;
                        requeue synch_entry;
                end  caller_entry;

                entry version_finish when true is
                begin
                        active := active+1;
                        requeue synch_exit;
                end version_finish;

                entry synch_entry (your_number : out version_number)
                              when synch_entry'count = N or let_go is
                begin
                    active := active-1;
                    your_number := version_number(active); -- type conversion
                    if active /= 1 then let_go := true;
                    else let_go := false; end if;
                end synch_entry;

                entry synch_exit when synch_exit'count = N or let_go is
                begin
                        active := active-1;
                        if active /= 1 then let_go := true;
                        else let_go := false; end if;
                end synch_exit;
        end nvp_synchroniser_t ;
end nvp_synchroniser ;
```

The order in which tasks call the DD object is irrelevant; each task receives a unique number (output parameter `your_number`) from the synchronising object and calls one of the version objects. Each method of the DD object is implemented following this simple template:

```
procedure Get_Min(l : access ft_list_t; elem : out elem_t) is
       decision : adjudicator_decision;
       My_number: version_number;
       elem_out: elem_t := default;
       abstract_list : abstract_list_state_t;
begin
       l.nvp_s.caller_entry(My_number);
       begin
          case My_number is
              when 1 =>    Get_Min (List_V1'Access, elem_out);
                           Give_State(List_V1'Access, abstract_list);
              when 2 =>    Get_Min (List_V2'Access, elem_out);
                           Give_State(List_V2'Access, abstract_list);
              when 3 =>    Get_Min (List_V3'Access, elem_out);
                           Give_State(List_V3'Access, abstract_list);
          end case;
          l_adj.Get_Min_Keep_Outs(My_number, true, elem_out);
          l_adj.Keep_State(My_number, abstract_list);
       exception when others  =>
              l_adj.Get_Min_Keep_Outs(My_number, false);
       end;

       l.nvp_s.version_finish;
       l_adj.Get_Min_Adjudicate(My_number, decision, elem_out);
       if decision = failure then
              raise list_failure;
       elsif decision = recover then
              l_adj.Give_Correct_Abstract_List_State(abstract_list);
              case My_number is                         -- version recovery:
                 when 1 => Correct_State(List_V1'Access, abstract_list);
                 when 2 => Correct_State(List_V2'Access, abstract_list);
                 when 3 => Correct_State(List_V3'Access, abstract_list);
              end case;
       end if;
end Get_Min;
```

As we have explained before, all N tasks are synchronised twice: first, by calling entry `caller_entry` of the synchronised object after they have entered the method, the second time by calling entry `version_finish` after they have passed output results and the version abstract state to the adjudicator. The controller catches any exception propagated outside versions and treats it as a version failure. The failure exception (`list_failure` in our example) is in the interface of the DD object and of each version.

The adjudicator is declared inside the DD object. It is implemented as a protected type to guarantee the mutual exclusion of its procedure execution:

```
package list_adjudicator is
       type adjudicator_decision is (recover, failure, ok);
  protected type list_adjudicator_t is
         -- for each applied method:
       procedure Get_Min_Keep_Outs (verion_N : in version_number;
                    except : in boolean; elem : in elem_t : = default);
       procedure Get_Min_Adjudicate (verion_N : in version_number;
                    decision : out adjudicator_decision; elem : out elem_t);
       -- ... similar for all methods

       procedure Keep_State(verion_N : in version_number;
                     state : in abstract_list_state_t);
       procedure Give_Correct_Abstract_List_State (
                     state : out abstract_list_state_t);
  end list_adjudicator_t;
end list_adjudicator ;
```

Each task calls procedure `Get_Min_Adjudicate` after all tasks have been synchronised on entry `version_finish`: this call returns the adjudicated output parameters if consensus has been reached. In this case it may return value `recovery` in parameter `decision` if the version which has been called by this particular task needs recovery. The adjudicated correct output parameters are returned by each task to the caller context.

5. Discussion

5.1. Nested Calls of DD Objects and DD Object Visibility

As opposed to the initial NVP scheme [11] in which DD objects can be declared inside versions, the proposed scheme does not allow such static nesting because each version is called here by only one task (so this version does not have many tasks inside). This means that a DD object cannot be declared inside version objects. The general rule is that such object should be visible for all contexts in which it is used by N caller tasks. This obviously allows nested calls of any DD objects from (some of the) versions of other DD objects. The Ada compiler will report any violations of this rule.

We realise that this rule does not completely agree with the idea of hiding diversity because one should guarantee that such objects are accessible by all caller tasks (so, generally speaking, they all should be aware of this fact). Although diversity is clearly encapsulated inside DD objects in our scheme and its control is hidden, the scheme may require changing the location where a DD object is declared. It seems clear that one cannot have NVP schemes with N caller tasks which work without such restriction.

5.2. Desertion

Usually the deserter task problem is considered in the context of atomic actions [18], when a participant fails to enter or to reach the end of an action, so that the rest of participants are stuck if appropriate measures are not taken. It is clear that similar problems can affect systems which employ DD objects: a task can fail to call a DD object, or a version can hang. We have found these similarities between the execution of DD objects with N callers and of atomic actions both fascinating and deep: to the best of our knowledge, they have not been discussed before. It is only reasonable that we will be using existing solutions here. Our scheme can be modified to employ some of the solutions proposed for dealing with deserters in the Ada distributed atomic action scheme presented in [19].

A hang of any version can be easily detected by the Ada asynchronous transfer of control. We can extend the NVP controller in the following way which allows version hangs to b treated as its failures:

```
case My_number is
    when 1 =>
           select
                 delay get_min_time_max;
                  raise list_failure;
           then abort
```

```
                Get_Min (List_V1'Access, elem_out);
                Give_State(List_V1'Access, abstract_list);
            end select;
        when 2 =>
            -- the same for all case alternatives ...
    end case;
```

If we want the NVP controller to detect a task that fails to call a DD object (the so called entry desertion), we have to use time-outs, which can be done by incorporating a watchdog task in the controller. An extended NVP controller will set a predefined time-out when the first task calls a method. After the failure to enter is detected, the controller signals the failure exception to all tasks which have been waiting on entry `caller_entry`. Another approach can be used if we do not want to use task creation. In this case the faulty task will be detected outside the DD object: N caller tasks synchronise their execution before calling a DD object using a timed entry in their select operators (in which case we might drop the entry synchronisation from the NVP controller functionality).

5.3. Problems and Future Research

5.3.1. Diversely Designed Adjudicators

Developing effective adjudicators can be a very complex and, as such, an error prone task. The general framework for applying diversity in OO systems proposes [6] treating adjudicators as conventional components to which design diversity can be applied if necessary. Unfortunately, this can be done only if all problems of concurrent version execution are ignored. In our implementation, the adjudicator is an Ada protected object with two interface procedures for each application method of the DD object and two additional procedures: `Give_Correct_Abstract_List_State` and `Keep_State`. They are both used for faulty version recovery; procedure `Keep_State` is used to pass the abstract version state to the adjudicator for comparison. We need this object to be protected to guarantee the mutual exclusions of all of its method executions. This is why we cannot apply our scheme directly to diverse design of adjudicators.

It is worth mentioning that our class diversity scheme cannot be used for employing diversity in Ada protected objects for many reasons, one of which is that it should be allowed for N caller tasks to execute procedure or entry calls concurrently.

5.3.2. Distributed DD Objects

The proposed approach can be further developed to be used in distributed systems and to allow version distribution. This can allow hardware faults to be tolerated because a DD object will be able to produce correct results even if the nodes in which the minority of versions are located have crashed. Moreover, this can be beneficial for decreasing the correlation of version failure modes when version partitions are implemented on different (hardware and/or software) platforms. A new scheme would use the standard features of Ada distributed programming.

This scheme has to break diversity encapsulation and rely on locating version objects in different partitions. This requires a new distributed NVP control as all communication between versions and the controller has to be remote. Version objects will be declared in separate partitions (e.g. of the `Remote_Call_Interface` category). This scheme may allow adjudicator and synchronising object distribution, but, being protected objects, they have to be declared inside the corresponding partitions and called through additional interface procedures. The Ada distribution model permits exceptions to be signalled via remote procedure calls, which means that version exceptions will be raised in the controller context as they are in the single-computer scheme presented before. A special care should be taken to detect version partition crashes: in many situations this will cause the raising of the predefined exception `Communication_Error` in the controller context but we may need to watchdog all remote calls as well (e.g. using the Ada asynchronous transfer of control, similarly to the approach shown in Section 5.2).

5.3.3. DD Object Consistency

We would like to briefly discuss some possible solutions of DD object consistency problems which should be addressed for two reasons: these objects are used cooperatively by several tasks calling the same methods and they are also used in a concurrent (competitive) environment, when different sets of tasks can call methods concurrently. Mutual exclusions of these calls should be guaranteed to avoid any violation of DD object state consistency. In our current implementation we assume that it is guaranteed at a higher application level by task coordination outside the DD objects.

A simple extension of the NVP control above could keep the next set of N caller tasks waiting on the `caller_entry` entry until the current set of tasks has completed any manipulations with versions (including their state recovery if it is necessary). Unfortunately this solves only part of the problem: for example, it still allows N tasks calling different application procedures of the same DD object to be executed.

In our current scheme we have one synchronising object for all object methods. To guarantee a mutual exclusion of different method executions, we should introduce a synchronising object into each application procedure and a general synchronising object which lets only one set of tasks to execute one procedure at a time. Although this solution is more general than the first one, it does not solve all problems because tasks from different sets can intermix in calling the same application procedure. To solve this problem, we have to extend this solution and have all tasks from the same set carry a unique identifier (an input parameter) to allow the NVP controller to distinguish tasks from different sets.

5.3.4. N caller tasks for DD objects with M versions

A very natural generalisation of the approach proposed is to allow M caller tasks to call DD objects with N versions. We will outline a new scheme of this kind here, yet it is clear that more effort should be invested into finalising it and making it usable. First, this scheme should allow either dynamic or static choice of caller numbers. To do the former, we should extend the interface of the DD object by a service procedure which is called to inform the object about the number of callers in the following method call (M being the input parameter). The latter approach is similar to the one used in the scheme presented in

Section 4. Secondly, we should program cases when M is greater than N and when it is smaller than N differently. For the first case we should let only N tasks call version objects, the remaining M-N tasks should bypass all calls of these objects and of the adjudicator. This can be programmed at the level of the template used for application methods of the DD object (procedure `Get_Min` in our example above) without modifying the synchronising object or the adjudicator. We will need task forking/joining to program the second case. The controller should allow M caller tasks to call first M version objects and should create N-M new tasks, each of which calls one of the remaining version objects in the way we do in our original scheme [11] (which can be characterised as a static scheme with one caller task and N version objects).

6. Concluding Remarks

Although the proposed solution is to some extent based on our previous scheme [11], we have found that the modifications are not straightforward since the task synchronisation patterns are different. We were able to re-use some parts of the previous scheme: version design and version class specification were not changed, the specification of the DD class is developed in exactly the same way, adjudication is basically the same. But the NVP controller works in a completely different fashion in the new scheme; this is why the synchronising object, as well as the templates describing how the DD object is developed, are different. Using Ada concurrency features helped us a lot in designing these new controlling components and in making them re-usable.

Another important difference lies in the way DD objects are used. We had to investigate general questions of employing DD objects intended for N tasks: visibility rules, DD object nesting, task synchronisation inside DD objects, application of DD objects, etc.

It is very important, from our point of view, that the scheme promotes structured diverse programming, supports object-oriented way of applying NVP (both the diversely designed units and versions are classes) and hides diversity inside classes.

The most important novelty of this scheme is the ability of DD objects to serve multiple application tasks. First of all, it allows DD objects to be used in new application contexts. Secondly, it offers better performance because the NVP controller does not need to fork service tasks inside DD objects: rather, it re-uses the existing threads of controls. Thirdly, it facilitates the modelling, analysing and understanding of the system by avoiding task creation. And lastly, the templates for implementing DD objects are significantly simpler and less error-prone as compered with the scheme which requires task forking.

Acknowledgement. This research has been supported by the EPSRC/UK DISCS ("Diversity in Safety Critical Software") Project.

References

1 **Randell, B** System Structure for Software Fault Tolerance, *IEEE TSE* **1**(2) (1975) 220-232

2 **Avizienis, A** The N-version Approach to Fault Tolerant Systems, *IEEE TSE* **11**(12) (1985) 1491-1501

3 **Lee, P A, Anderson, T** *Fault Tolerance: Principles and Practice,* Springer-Verlag, Wien - New York (1990)

4 **Rogers, P, Wellings, A J** State Restoration in Ada 95: A Portable Approach Supporting Software Fault Tolerance, *J. of Systems and Software* **to appear** (1999)

5 **Lyu, M and Avizienis, A** Assuring design diversity in N-version software: a design paradigm for N-version programming, *Proc. 2nd Dependable Computing for Critical Applications Symposium,* (1992) 197-218

6 **Xu, J, Randell, B, Rubira, C M F, Stroud, R J** Toward and Object-Oriented Approach to Software Fault Tolerance, in *Fault-Tolerant Parallel and Distributed Systems* **Avreski, D (Ed.)** IEEE CS Press (1995)

7 **Issarny, V** An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards Reusable, Robust Distributed Software, *J. of Object-Oriented Programming* **6**(6) (1993) 29-40

8 **Strigini, L, Avizienis, A** Software Fault-Tolerance and Design Diversity: Past Experience and Future Evolution, *Proc. the 4th Int. Conf on Computer Safety, Reliability and Security,* Como, Italy (1985) 167-172

9 **Ciompi, P, Grandoni, F, Strigini, L** Software Fault Tolerance, in *Delta-4: A Generic Architecture for Dependable Distributed Computing* **Powell, D (Ed.)** Springer-Verlag (1991)

10 **Randell, B** Approaches to Software Fault Tolerance, *Proc. the 25th Annual LAAS Conference,* Toulouse, France (1993) 33-42

11 **Romanovsky, A** Class Diversity Support in Object-Oriented Languages, *J. of Systems and Software* **48** (1999) 43-57

12 **Romanovsky, A** Abstract Object State and Version Recovery in N-Version Programming, *Proc. TOOLS 29,* Nancy, France (1999) 86-95

13 **Romanovsky, A** Faulty Version Recovery in Object-Oriented N-Version Programming. Department of Computing Science, University of Newcastle upon Tyne, , CS-TR-679 (1999)

14 **Lee, P-N, Tamboli, A** Concurrent Correspondent Modules: A Fault Tolerant Ada Implementation, *Proc. the 8th Annual Int. Phoenix Conf. on Computers and Communications,* (1989) 300-304

15 **Rubira, C M F, Stroud, R J** Forward and Backward Error Recovery in C++, *Object Oriented Systems* **1**(1) (1994) 61-85

16 **Wichmann, B A** Guidance for the use of the Ada programming language in High Integrity Systems, *Ada Letters* **18**(4) (1998) 47-94

17 **Randell, B** Recursive Structured Distributed Computing Systems, *Proc. the 3rd Symposium on Reliability in Distributed Software and Database Systems,* Florida, USA (1983) 3-11

18 **Kim, K H** Approaches to Mechanization of the Conversation Scheme Based on Monitors, *IEEE TSE* **8**(3) (1982) 189-197

19 **Mitchell, S E, Wellings, A J, Romanovsky, A** Distributed Atomic Actions in Ada 95, *The Computer J.* **41**(7) (1998) 486-502