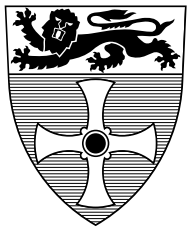
UNIVERSITY OF
NEWCASTLE



COMPUTING SCIENCE

Class Diversity Support in Object Oriented Languages

Alexander Romanovsky

TECHNICAL REPORT SERIES
No. CS-TR-661
February 1999

Contact:
alexander.romanovsky@ncl.ac.uk
www.cs.ncl.ac.uk/people/alexander.romanovsky/home.formal

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

Class Diversity Support in Object-Oriented Languages

Alexander Romanovsky

Department of Computing Science, University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

We start with a general approach to introducing software fault tolerance (SFT) into object-oriented (OO) systems [Xu *et al.* 1995] and proceed in two directions. The first one is the use of SFT schemes within standard OO languages. New questions which arise when we are dealing with these languages are addressed. Our intention is to thoroughly analyse all engineering steps which allow diversity to be introduced in systems programmed in these languages. Some new general problems are spotted and discussed as well. The second direction is dealing with version concurrency and distributedness in a general way. We investigate providing SFT by class diversity, which is the most general way of designing diverse software in OO systems. We concentrate on N-version programming and give an exhaustive discussion of this approach. One of the main reasons for this choice is that we have come to believe that the general approach which allows a unified discussion of all SFT schemes is rather restrictive because it does not properly address the differences between these schemes which represent their essences and the most difficult parts of their implementation and support. Our intention is to discuss the use of N-version programming in OO terms and to outline all novelties arising from this. The re-usability of SFT features is a key point in our approach. One of the conclusions we have arrived at is that, generally speaking, the entire states of version objects should be compared to detect and mask the faulty one. We propose unifying in one component features dealing with adjudication and faulty object recovery because these functionalities have a lot in common. Our approach is demonstrated using Ada 95.

1. Introduction

1.1 Software Fault Tolerance

It has been growing clearer in the last years that a considerable number of software faults (design bugs) are left in systems after their implementation has been completed; moreover, they cause an increasing portion of the total of errors occurring in applications. That is why a systematic inclusion of SFT is considered to be a must for many applications. The most general way to provide tolerance to design faults is

software diversity which is based on an extra diverse application code intended for error detecting and recovering. Two canonical SFT schemes were invented in 70-ties.

The recovery block (RB) scheme [Randell 1975] assumes that a set of *alternates* (*versions*) of the application code (more often of its part) is designed by independent application programmers. The *acceptance test* is used to check the correctness of the execution of these versions. This is essentially a dynamic scheme: versions are executed sequentially and if, on completing a version, the acceptance test is ensured, the RB has been executed. Otherwise the system is rolled back and the next alternate is tried. N-version programming (NVP) [Avizienis 1985] is a static scheme: all versions are executed in parallel and their results are compared by an *adjudicator* to find the correct ones and to mask the execution of the faulty versions by majority voting. Each of these schemes has its own drawbacks and advantages, and their application fields differ (e.g. depending on the type of the redundant resource available). The interested reader is referred to [Avizienis 1985, Lee & Anderson 1990, Bishop 1995] for a complete discussion and comparison and for brief summaries of the applications in which they have been used.

A whole range of SFT schemes intermediate between these two have been proposed since then. Some of them, even though they are static, do not require all versions to be executed sequentially (e.g. self-configurable optimal programming - SCOP [Bondavalli *et al.* 1993]).

1.2 Existing Approaches

To be used in OO systems and benefit from all the advantages involved (re-usability and extendability, first of all), these schemes should be designed and used in special ways. Diversity, control mechanisms, comparing or checking results (the adjudicator or acceptance test), state restoration for RBs, input/output data, etc. should be thought of in OO terms. The papers [Chang & Dillon 1994, Tso & Shokri 1995, Xu *et al.* 1995, Xu *et al.* 1996] take this approach. In particular, the paper [Xu *et al.* 1995] proposes a general OO framework and thoroughly discusses different types of diversity. It introduces the main idea of the re-usability of service SFT components (classes) and offers three pre-defined classes: `variant`, `adjudicator` and `controller`, which allow constructing any of the existing SFT schemes. We fully agree with the authors in that an undisciplined introduction of design diversity can be error-prone; our intention is to demonstrate how OO features can make it a simple and routine job. However, two approaches in [Tso & Shokri 1995, Xu *et al.* 1995] are discussed on a very general level, and it is due to this, we believe, that they lack important details without which their ideas cannot be applied directly. Moreover,

software architecture [Tso & Shokri 1995] has never been discussed in published literature to the extent sufficient to understand and to evaluate it. The conceptual linguistic framework [Xu *et al.* 1995] cannot be used directly in practice because it concerns only the specifications of the pre-defined classes; moreover, the authors do not seem to have intended to give these details. We had a lot of difficulties when trying to use it for Ada 95 [Intermetrics 1995] and Java [Sun 1995]. Moreover, this framework cannot be used directly even in C++ (though a C++ notation is used) because this language has no features for concurrent programming. While applying this framework, we became aware of many details and problems which believe to be of general importance for practitioners.

Class diversity (as opposed to method diversity) is the most general way to introduce SFT because classes are the units of system design in OO programming. We agree with the authors of [Chang & Dillon 1994] that method diversity is more appropriate for supporting RBs while NVP is better supported by class diversity. The reason for this is that the use of class diversity for the RB scheme would involve moving all version objects into new states after any of them ensures the acceptance test, which is very inefficient and diminishes all advantages of dynamic SFT. On the other hand, it is more difficult to provide SFT support for NVP because all versions are to be executed in parallel and because the recovery of the faulty version is a non-trivial problem which cannot be solved transparently for application programmers (as opposed to state restoration required for RBs).

The approach in [Xu *et al.* 1996] is based on method diversity: classes have only one method and it is not made clear in this framework whether class or method diversity is considered. Within this approach, objects have no states. Moreover, the strong typing of the language is not assumed. We believe that exceptions should be propagated by the exception mechanism but not signalled as the return code. Generally speaking, output parameters should be of application-dependent types, that is why the adjudicator cannot be fully application-independent. We believe that this approach should be adjusted and made more concrete for class diversity.

Unfortunately, the papers [Tso & Shokri 1995, Xu *et al.* 1995] do not discuss the problem of concurrent version execution, which is not only a difficult and error-prone task, but a problem which is very important for the entire SFT scheme. The problem of version distribution for object oriented languages does not seem to get the necessary attention, either. Many modern languages (e.g. Ada 95 and Java) allow us to do this on the language level because they have features for distributed programming. Note that if versions are distributed, their control should be concurrent and its consistency should be guaranteed.

The paper [Rubira & Stroud 1994] addresses a number of SFT problems, but it lacks generality and is too C++ oriented. Concurrent version control is not and cannot be discussed in C++. There is no attempt made to have reusable classes and components or to clearly separate application functionalities from SFT ones. There is no clear distinction between the NVP and RB schemes in this paper. Class diversity is not discussed, either. However, the idea of using an abstract state to unify the state representations of all versions, proposed in the paper, seems extremely promising.

Within the Arche system [Issarny 1993], versions can be declared as different classes and viewed as different implementations of a given (abstract) application type. SFT is introduced on the language level here because Arche relies essentially on the underlying run time system providing coordinated calls of versions; that is why the use of NVP consists only in designing versions and the adjudicating method.

The approach in [Purtilo & Jalote 1991] allows SFT (both the RB and NVP schemes) to be programmed using some standard programming languages (e.g. Pascal, C, Prolog) for designing version and adjudicator procedures and an auxiliary *module interconnection language* for specifying SFT. It is impossible to apply this approach to standard OO languages because all SFT support is hidden in the run time system and because it is not expressed in OO terms.

2. Formulating Problems

A thorough analysis of the existing proposals [Rubira & Stroud 1994, Tso & Shokri 1995, Xu *et al.* 1995] shows that it is not always easy to apply them in practice. There are many reasons for this: an orientation towards a particular language, a lack of many important topics and an omission of many details, a rather high level of unification of all SFT schemes within one framework. For example, it is clear that very different service class libraries should be provided for RBs and NVP schemes (state restoration for RBs and faulty version recovery for NVP, or the acceptance test for RBs and results comparison for NVP). Our intention is to design a SFT scheme of a particular - viz. NVP - type, and to have a set of programming conventions which would be properly detailed and oriented towards standard languages having no explicit support for NVP. The version execution should be discussed in such detail as to make the approach practical. In particular, we believe that version classes are essentially different in the RB and NVP schemes and that although unifying them in one general model is good for general understanding, scheme-specific research is to follow. We will give a detailed description of NVP in OO languages and discuss an implementation in standard languages (a set of conventions and re-usable components).

One of our main intentions is to introduce class diversity and its support in an object-oriented way; this is why we have to re-formulate the general NVP scheme and all the problems involved (diversity, adjudication, comparing results, etc.) in OO terms. One of the problems we shall address is result adjudication. OO programming relies on data encapsulation, so it is not enough to compare method results, i.e. output parameters. The concept of *results* should be extended by including the object state. For example, method `sort` of class `list` can have no output parameters at all. Another example could be as follows. Let us consider an object that has methods `Prepare_Missile` and `Launch_Missile` without any output parameters. It would be extremely desirable to use software diversity in designing objects of this sort, and if we are using class diversity, then the states of all versions should be compared and adjudicated after the first method has been completed and before the call of the second one.

We would like to clearly separate the application-dependent code from class diversity support. The latter can be made re-usable and, to a great extent, hidden from application programmers. That is why we do not want to make application programmers change the version class very much.

Our scheme should take into account the strong typing of the language; thus, versions should be derived from the same abstract class and results should be of application-dependent classes.

Our approach uses concurrency, which is essential for the majority of SFT schemes (even in sequential applications). We would like to make this important step and try and improve the paper [Xu *et al.* 1995] in this respect because expressing the concurrent execution of versions is impossible in C++. We are going to investigate the problems of concurrent version execution in great detail: how to start the execution of several versions in parallel, how to synchronise them when they are finished, how to guarantee the consistency of the adjudicator execution when several versions work in parallel. We want to outline all steps and trade-offs of the NVP scheme control in OO languages and to find out which of the components can be programmed to be re-usable.

Another of our intentions is to offer detailed practical methods of faulty version recovery.

Our basic idea is twofold: to make it possible to introduce software diversity in a very disciplined way but without either a special language construct or a run time system (although the use of a special run time support, similar to DEDIX [Avizienis 1985], should be able to make our approach simpler). We would like to have a model which

would be applicable to standard languages. It would be much easier if a language had class diversity features but there are no languages like this and there will hardly be any in the foreseeable future. That is why our scheme will be presented as a set of programmers' conventions, templates and standard re-usable components. We believe that object-orientation makes it possible to outline disciplined, routine and error-free engineering steps of using class diversity.

Complexity control and separation of concerns while producing critical applications is of great importance for using NVP. OO programming itself gives an obvious basis for this. We fully agree with the view expressed in [Xu *et al.* 1996] that this separation is at the same time a division of programmers' responsibilities. In this paper all programmers are grouped into three categories: *system programmers* develop different re-usable SFT components to be used and adjusted by *FT designers* who create fault tolerant application-specific objects to be used by *users*. Our classification differs in one important point: we believe that those subcomponents of SFT components which are application-dependent, e.g. versions, should be designed by *application programmers* rather than by FT designers. These application programmers should know nothing about the fact that they are designing subcomponents for NVP. A minor distinction is that we will not concentrate on the differences between system programmers and FT designers because, as we have explained, our approach is not so general as that in [Xu *et al.* 1995, Xu *et al.* 1996].

3. General Model

The computational model of NVP in OO systems is as follows. We assume that a number of class versions was designed using the same abstract specification of the application class. An additional class called the *NVP manager* class, which has the same interface, is programmed by the FT designer. An object of this class (the *manager object*) manages the execution of the entire NVP scheme. Version objects are known only to the manager object and are asynchronously called by it. All scheme control is provided by the NVP manager class. Each time a method of the manager object is called, a concurrent call of N methods from N version objects happens. The same input parameters are passed on to all versions, which guarantees the 'consistency of initial conditions' [Avizienis 1985]. When the executions of these methods are completed, the manager gets the results and calls the adjudicator which returns the correct output results and the list of faulty versions. The latter can be recovered by the manager (if appropriate measures are taken). This OO model avoids many of the problems of guaranteeing version determinism.

Version classes are to be implemented by different programmers using different algorithms or even languages (e.g. Ada classes can be used in Java programs with the help of Appletmagic or by using CORBA interface) [Avizienis 1985]. Diversity is hidden from the caller. All version objects are invisible from the outside and are accessed by the manager class only: they have to be declared in the private part (body) of this class.

Our general model assumes that version objects have their states, so version objects cannot be created each time a method is called and have to be declared in the manager class implementation.

In spite of the fact that concurrency control is inevitable for NVP, none of the existing SFT approaches intended for OO programming addresses the problems of parallel version execution. The problems of data consistency are very important. That is why the NVP manager and the adjudicator should be programmed in a special way to protect the consistency of their data and guarantee the consistency of results. We need concurrency for several purposes: to start the execution of several versions asynchronously, to synchronise them when they have been completed, to adjudicate results and to recover the version state in a consistent way. Moreover, if we want to have a distributed execution of versions, we have to call them asynchronously and collect the results in a consistent way in one location (or to use some distributed agreement algorithms). One more purpose is to be able to abort a version asynchronously if it cannot be finished and its time-out is broken. Finally, we believe that version calls, adjudication, faulty object recovery should be parallelised as much as possible.

4. Conceptual Proposals

4.1. Manager

The manager class has the external interface of the application class and the entire diversity control is hidden in its implementation. To guarantee this, we assume that the application class specification is known and exists in the form of an abstract (virtual) class; the manager class is derived from it.

The responsibilities of the manager class are to start the execution of several versions in parallel, to catch all exceptions which can be raised by versions, to control the version execution, to synchronise them when they are finished, to signal the failure exception if there are too many faulty versions, to guarantee the consistency of version states, to call the adjudicator and to initiate the faulty version recovery. It is

clear that implementing class diversity requires the parallel execution of versions. Because of this the manager has to introduce (spawn) concurrency and to call the same methods of all versions asynchronously. After they are completed, the manager has to synchronise all of them and to collect their results in a consistent way. That is why we propose to split the manager into several parts: each method of this class is implemented as a part of the manager and an additional centralised part (controller) controlling the synchronisation of versions when they are finished. Each manager method should have a standard structure. The monitor-like construct seems to be very appropriate for implementing the controller. It allows better parallelisation and a guarantee of NVP consistency. Generally speaking, the manager can be made largely application-independent and will only have to be adjusted (maybe by using templates, generics or parameters) to the names of version objects and their methods. The manager class is implemented by the FT designer.

The manager can provide either synchronous or asynchronous version control depending on the application requirements and on the ability of the language to interrupt the concurrent activity asynchronously (say, Ada 95 [Intermetrics 1995], unlike Ada83, has asynchronous transfer of control). Asynchronous control can be used to interrupt the version if it has exceeded its time constraints.

A lot of choices depend on the language features, but, generally speaking, each method or manager has or forks N internal execution threads (tasks) which can be started either when the object is created or when the method is called and each of which calls one of the versions (one of the solutions could be asynchronous calls of all versions, another one will be discussed in Section 5). They are synchronised by the controller and report version results to the adjudicator. When all of the versions have completed, these tasks receive the adjudicator's decision and adjudicated results. The decision can be of three sorts: recover the faulty version, raise the failure exception or proceed with the correct results.

The manager class should be made re-usable as much as possible: classes for the entire manager class and for the controller should be provided. If it is not possible, then templates, generics or parameters should be used to make programmers' job less error-prone.

4.2. Versions

As we have mentioned before, we do not want to make application programmers change the version class interface very much or to let them know that they are implementing versions. They should have the application class specification at their

disposal. Version classes, as well as the manager class, should be derived from the same abstract (virtual) class. The abstract class serves as a formal specification for all version designers. This allows the adjudicator to compare their results (output parameters) and to produce the correct ones, which would be otherwise of incomparable types (because of the strong typing).

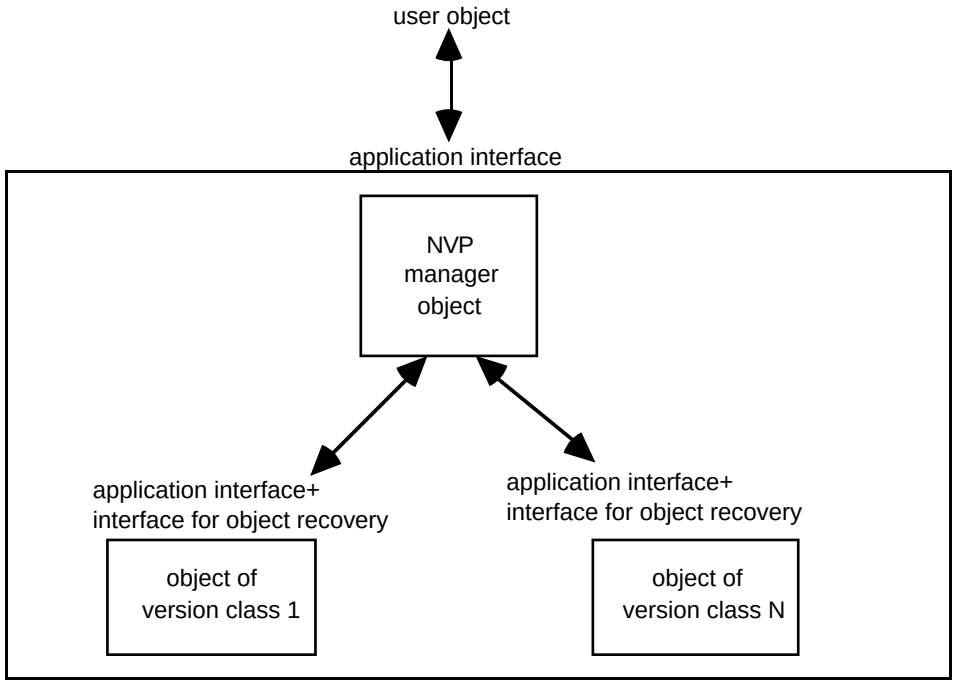


Figure 1. Class and version interfaces.

Version implementation should be made re-usable and extendable as much as the language allows; however, this has nothing to do with SFT.

We will show in Section 4.4 that, apart from application methods, each version class should have auxiliary features for an object to be recovered if its results are found to be incorrect. These features should be accessible via version class interface (see Figure 1).

4.3. Adjudicator Object

Within the general approach, the adjudicator collects all version results and returns the correct result (besides, it can return an indication of the faulty versions which can be recovered later). Generally speaking, this idea should be adjusted for the class diversity approach when it is not all object methods that return the results. In OO programming, the intention is to encapsulate data in classes but not always pass them as parameters. So, it is not always possible to detect an error by comparing the output

parameters of the methods (as proposed in [Xu *et al.* 1995]). We have to compare both the version states and output parameters.

We believe that the adjudicator has to be application-dependent for two important reasons. Firstly, as we have shown, it manipulates the representation of the internal state of all versions. Secondly, the version results which should be compared are application-specific and, generally speaking, can be of any application type. The adjudicator is called/used after each version has completed, and the results and the internal state of the version (which should be represented in a unified, comparable way for all versions) are passed on to it. Moreover, in principle the adjudicator should also be method-specific, so, the adjudicator object has to have the corresponding adjudicating method for each application method. Having said this, we still believe that a standard OO language can help to introduce some re-usable components by using templates, generics, etc. because general adjudicating algorithms can be re-used, as was rightly pointed out in [Avizienis 1985].

One important aspect which is not usually discussed is the execution of the adjudicator in a concurrent environment. Its consistency and that of its data should be guaranteed when several versions work in parallel. It is obviously important to use some suitable concurrent programming features to design the adjudicator and to find a proper and less restrictive way to coordinate the execution of the adjudicator and the manager class. We regard a monitor-like construct as very suitable for programming the adjudicator, because it allows data encapsulation and a mutually exclusive access to them.

An asynchronous NVP scheme can be programmed (which can be useful for distributed systems, in particular), where the adjudicator produces the final correct results without waiting for all versions provided a majority of them agree on those. The results of the belated versions are used only for identifying the faulty ones. The manager should guarantee that all versions from the previous NVP call have been completed before the next call starts execution.

4.4. Faulty Version Recovery

We believe that local error detection and recovery (e.g. exception handlers) should be implemented and used within each version. But the adjudicator produces the correct results even if they fail. At the same time it detects the faulty version(s) whose results are ignored and which should be either masked or recovered. Obviously, the latter is much more preferable. This can be done only with the help of the correct versions (note that the backward recovery of the faulty object does not help). The situation is

complicated by the fact that the data representing the version state are encapsulated into each version and hidden from the outside. Basically, the faulty version should be recovered by moving it to the state corresponding to those of the correct versions (although this contradicts, to some extent, our idea of version programmers unaware of the fact that they design different versions). This is why the conventional forward recovery techniques cannot be applied here.

Generally speaking, recovery is feasible if it is possible to find the relation (mapping) between the internal data of different versions [Rubira & Stroud 1994, Xu *et al.* 1995]. The following approach has been chosen. We assume that the programmer of each version has to implement two additional methods: `Give_State` and `Correct_State`. The first one is called in a correct object after a faulty one has been detected. It returns the complete state of this version in an *intermediate general abstract format.* Method `Correct_State` of the faulty version is called afterwards, all information representing the state of the correct version is passed to it. This method recovers the state of the object.

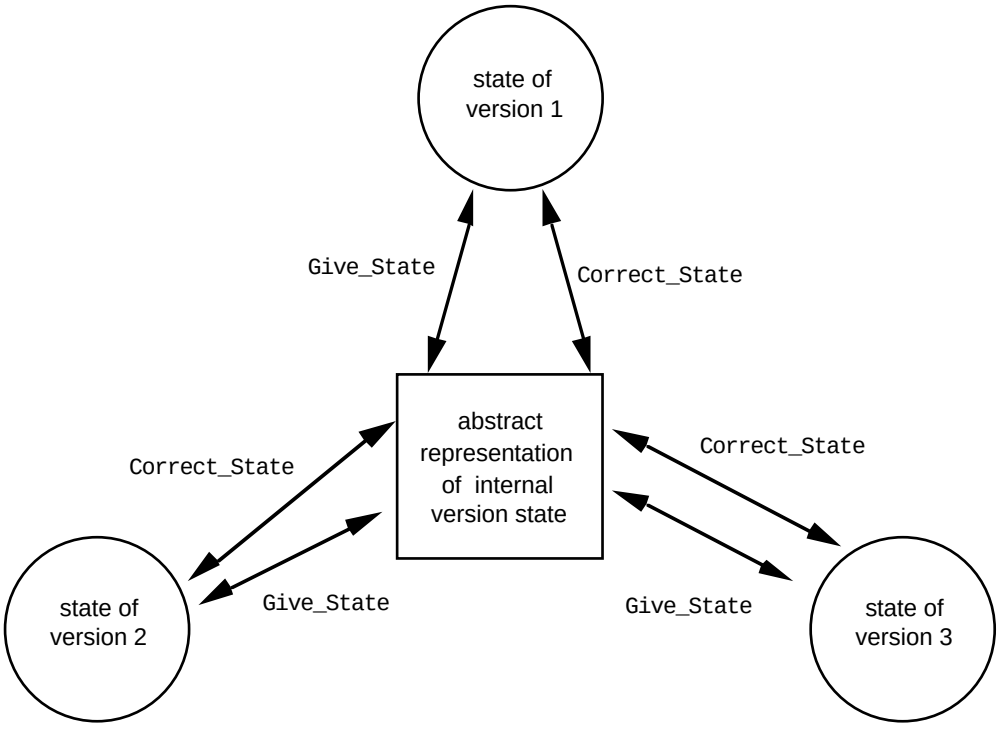


Figure 2. Faulty version recovery based on the intermediate state representation.

We can envisage several approaches to presenting the internal states of different versions in a uniform way. One idea is to use components of only basic types (e.g. `real`, `integer`, `boolean`, `string`) which are kept in some record types known to all version programmers. Another approach could be to use the internal data of one of the versions as this intermediate data representation. Finally, a unified class could be

used which would be provided by FT designers; objects of this class are returned when service method `Give_State` of each version object is called. The problem is that, with any of the approaches above, there should be enough information in the intermediate state representation to recover the corrupted internal states of any version (one-to-one mapping). This should be guaranteed by each application programmer when designing method `Correct_State` . Figure 2 illustrates our approach.

A very simple extension of this approach could allow the backward recovery of the faulty version. Method `Give_State` can be used to obtain the state of the version before the method call. If it is saved in stable storage, we can roll the faulty object back and re-try its method as a means of recovery.

Our approach is somewhat similar to the initial idea in [Rubira & Stroud 1994] about using an abstract state of the variants, but ours is not intended either for ensuring the consistency of versions (when they are called in parallel) or for backward recovery schemes. We have given a detailed analysis of the approach which is intended for NVP only. In the next section we will show how this approach can be extended to allow recovery and adjudication to be merged.

Our approach makes it possible to extend the community error recovery scheme [Tso & Avizienis 1987], which was originally intended for versions with the same internal data. A reasonable scheme extension can be designed using this proposal when/if these data are mappable (e.g. a list is presented as an array and as a hash table) and an intermediate format exists.

4.5. Unified Approach to Adjudicating and Mapping

It is not difficult to see that our general approaches to version state comparison in Section 4.3 and to object state mapping in Section 4.4 have a lot in common, and we believe that it is conceptually right that state comparison and mapping are similar. Method `Give_State` (Section 4.4) can be used for both of them because it returns an intermediate representation of version internal states which can be either compared or used for correcting the faulty version. Our idea is for these two functionalities to be provided by the adjudicator. It uses the intermediate representation of version states to compare them; the same representation can be used to recover the faulty ones if they are detected. This approach can be applied only if the version class interface is extended by including two new methods allowing the current internal state to be output in a uniform format and the corrupted version state to be corrected by using the state of a correct version. These methods should be implemented by version programmers.

Our approach helps to solve a very important problem of the atomicity of method execution. It should be guaranteed by the NVP scheme that if the method execution is not successfully completed and the failure exception is raised, then this execution has no effect on the encapsulated version objects. The initial version abstract state should be kept in stable storage and used by the manager to roll the state of all versions back before raising the failure exception.

4.6. NVP in Distributed Systems

Many modern OO languages have features for programming distributed applications (e.g. Ada 95 - Distributed Annex [Intermetrics 1995] and Java - Remote Method Invocation (RMI) [Sun 1996]). This allows tolerating not only design faults but hardware faults as well. Our approach to NVP programming and class diversity can be easily applied for these systems because it is obviously much easier to distribute version objects than diversely designed methods. Generally speaking, a standard OO language should allow object distribution and a simple remote method (procedure) call, which are basic features for all distributed OO systems.

As we have mentioned before, we believe that it is much safer to encapsulate version objects into the manager class. The only way to use NVP in distributed systems is to locate versions in different nodes, which makes them visible to the outside world and increases the chances for information smuggling and for version misuse. We believe that some additional care should be taken to guarantee that only the manager can call them (e.g. an additional parameter, a sort of manager identifier, can be used in each call, or a proxy object can be introduced in each node which encapsulates the version and checks all calls going through it to the version). Our approach does not require either atomic or ordered broadcast because only the manager is supposed to call the versions.

This implementation can be made even simpler if exceptions can be propagated through remote method calls (as is the case for both Ada 95 and Java).

Our approach is essentially centralised because the manager and the adjudicator control the execution of all versions. Additional care should be taken to allow tolerating the hardware faults of the nodes where these components are located; a replication mechanism can be used.

4.7. General Structure and Class Hierarchy

Finally, we can summarise the general structure of NVP and interconnections between its different components, as envisaged in our approach, in the way shown in Figure 3.

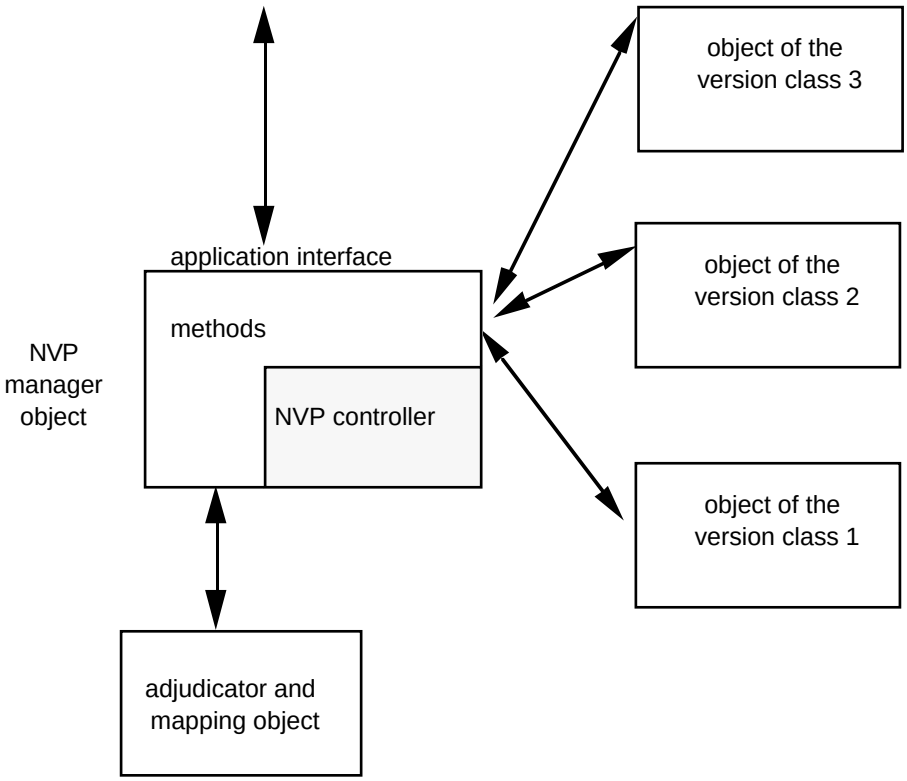


Figure 3. General structure of NVP with class diversity.

All components of our scheme are objects, which makes it possible to design any of them (the adjudicator; the controller; local objects used by versions; even, the manager object) diversely using NVP; this fact can be hidden from outside objects and the approach can be used recursively. We treat any of them as an idealised component with diverse design (the concept was proposed in [Xu *et al.* 1995]).

There could be several ways of deriving the manager class and version classes from the abstract class. We think that the most natural one is to derive each of them directly from the abstract class. A service class can be used to extend the interface of each version class by including methods for version recovery by mapping. One more standard class can make it possible to add version control to the manager class. This is summarised in Figure 4.

Generally speaking, different linguistic mechanisms can be used in different standard languages to achieve the re-usability of the NVP scheme and to allow the extendability of a particular approach: inheritance, class and package library, templates, generics, etc.

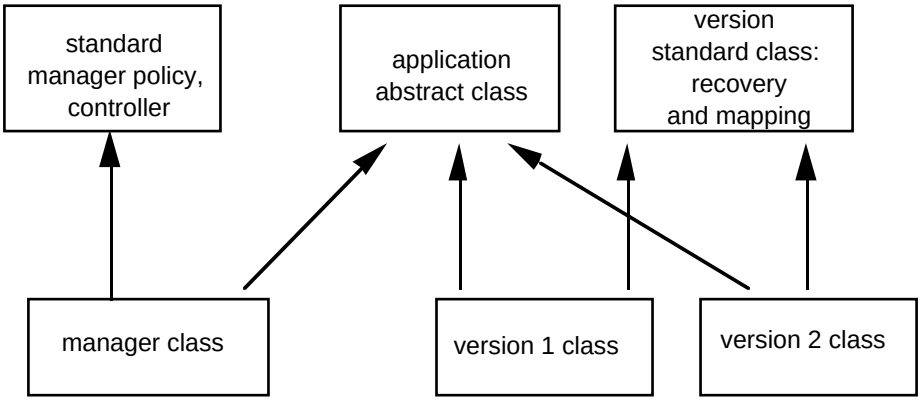


Figure 4. Using inheritance for facilitating NVP.

5. Programming NVP scheme in Ada 95

We will demonstrate our proposal using Ada 95 [Intermetrics 1995]. This gives us an opportunity to discuss it in more detail. Moreover, we believe that this detailed description can be used both for fast prototyping and directly in critical applications which are often designed using this language. Note that this Section does not offer a complete guidance for programmers; rather, it shows how all NVP components can be designed within a conventional wide-spread industrial language.

5.1. General Structure

In Ada 95 a class is programmed as a tagged type and a package including several subprogram-methods; one parameter of each of these methods should be of this type. Let us assume that there is an abstract specification of an application class (type) `list_t` which we are going to implement diversely using NVP:

```
package class_list is
   type list_t is abstract tagged limited null record;
   ...
   procedure Get_Min(l : access list_t; elem : out elem_t) is abstract;
   procedure Sort (l : access list_t) is abstract;
   ...
   list_failure : exception;
end class_list;
```

Our approach assumes that version classes and the manager class are derived from this. Manager class `ft_list_t` is designed by the FT designer and has to have concurrent tasks to execute versions in parallel. Type `list_t` is extended by adding the controller. All abstract methods are made concrete. The external specification of this class is the same as the specification of abstract class `list_t`:

```
package class_ft_list is
```

```
   type ft_list_t is tagged limited private;
   ...
   procedure Get_Min(l : access ft_list_t; elem : out elem_t);
   procedure Sort (l : access ft_list_t);
   ...
   ft_list_failure : exception;
private
   type ft_list_t is new list_t with
        record
          nvp_c: NVP_controller(version_max); -- controller
        end record;
end class_ft_list;
```

In the next section we will explain how object `nvp_c` of protected type `NVP_controller` works.

All versions, which are objects of the diversly-implemented classes, are declared in the private part of the manager class and hidden from the outside world. Moreover, the adjudicator object of type `list_adjudicator_t` is declared here as well. The manager implementation looks as follows:

```
package body class_ft_list is
   List_V1 : aliased version_1_t;
   List_V2 : aliased version_2_t;
   List_V3 : aliased version_3_t;
   l_adj : list_adjudicator_t;
   ...
   procedure Get_Min(l : access ft_list_t; elem : out elem_t) is ...
   procedure Sort (l : access ft_list_t) is ...
   ...
end class_ft_list;
```

5.2. Concurrent Version Execution

As we explained in Section 4, versions are started in parallel by each application method. Versions must know nothing about the parallelism of their execution, all concurrency should be provided on the manager level. The body of each application method in the manager class implementation includes N tasks which are spawned when the method is called. Each of the tasks calls the corresponding method of its version, passes the output parameters and the version state to the adjudicator object, synchronises its execution with those of other versions by calling entry `version_finish` of the controller. It is the responsibility of each task to recover the state of its version when the adjudicator lets the task know that its version is faulty. The adjudicator returns the correct output results to the method body which is executed concurrently with the tasks. If the correct results are not found, the failure exception is raised.

```
   procedure Get_Min(l : access ft_list_t; elem : out elem_t) is
     decision : adjudicator_decision;
     task Get_Min_LV1;
     task Get_Min_LV2;
```

```
 task Get_Min_LV3;

task body Get_Min_LV1 is
   My_number: constant :=1;
   elem_out: elem_t :=default;
   abstract_list : abstract_list_state_t;
   decision : adjudicator_decision;
 begin
    Get_Min (List_V1'Access, elem_out);
    l_adj.Get_Min_Keep_Outs (My_number, true, elem_out);
    Give_State (List_V1'Access, abstract_list);
    l_adj.Keep_State (My_number, abstract_list);
    l.nvp_c.version_finish;
    l_adj.Get_Min_Adjudicate (My_number, decision, elem_out);
    if decision=recover then
          l_adj.Give_Correct_Abstract_List_State (abstract_list);
          Correct_State (List_V1'Access, abstract_list);
    end if;
    exception when others =>
          l_adj.Get_Min_Keep_Outs (My_number, false);
          l.nvp_c.version_finish;
          l_adj.Get_Min_Adjudicate (My_number,decision,elem_out);
          if decision=recover then
                l_adj.Give_Correct_Abstract_List_State (abstract_list);
                Correct_State (List_V1'Access, abstract_list);
          end if;
end Get_Min_LV1;

task body Get_Min_LV2  is  ...
task body Get_Min_LV3  is  ...
begin
    l.nvp_c.version_finish;
    l_adj.Get_Min_Adjudicate (version_max, decision, elem);
    if decision=failure then
          raise ft_list_failure;
    end if;
end Get_Min;
```

Each task calls the corresponding version object and controls further execution by synchronising with controller `nvp_c` and by calling the appropriate methods of adjudicator `l_adj` and of the version (to get the version state and to recover it), see Sections 5.3 and 5.4 for details. The general control is provided by a centralised controller which can be easily implemented in Ada 95 as a parameterised protected object:

```
protected type NVP_controller (N : positive) is
    entry version_finish;
private
    entry synch_exit;
    active : integer := ;
    let_go : boolean := false;
end NVP_controller;

protected body NVP_controller is
    entry version_finish when true is
    begin
          active := active+1;
          requeue synch_exit;
    end version_finish;
    entry synch_exit when synch_exit'count=N+1 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_controller;
```

Parameter `N` represents the number of versions; as a result, type `NVP_controller` is reusable.

As we mentioned in Section 4, an asynchronous version completion may be required. Ada 95 has features for asynchronous transfer of control [Burns & Wellings 1995], which allows it to be programmed in a simple way. It is clear that the interrupted version is in a faulty state and should be recovered afterwards.

The drawback of this approach for some systems with expensive task creation could be that all N tasks are created each time a method is called (though this is in line with the main NVP ideas). Ada makes it possible to program different approaches to fork concurrency (although they would have much more complex conventions to be followed by FT programmers). For example, N tasks can be created when an NVP object is created; each of these has an endless loop and a set of accepts to trigger each method of the corresponding version.

5.3. Version Design

Let us consider version classes in detail. Derived class `version_1_t` declaring a version class looks as follows:

```
package class_version_1 is
   type version_1_t is tagged limited private;
   ...
   procedure Get_Min (l : access version_1_t; elem : out elem_t);
   procedure Sort (l : access version_1_t);
   ...
   version_1_failure : exception;

   procedure Give_State (l : access version_1_t;
                         state : out abstract_list_state_t);
   procedure Correct_State (l : access version_1_t;
                         state : in abstract_list_state_t);
private
   type version_1_t is new list_t  with null record;
end class_version_1 ;
```

Version classes are designed by different application programmers. As we have said before, two additional methods `Give_State` and `Correct_State` are used to compare version states and to recover the state of this version if it is found to be faulty.

5.4. Adjudicating and State Mapping Class

The adjudicator is called in the concurrent environment and should behave consistently. We propose to design it as Ada protected type `list_adjudicator_t`, all methods of which are executed with mutual exclusion. An instance of this type is created in the manager body. For each application method (e.g. `Get_Min`), the adjudicator has two service methods which are called by the manager when each version execution is completed. For example, there are procedures `Get_Min_Keep_Outs` and `Get_Min_Adjudicate` corresponding to method `Get_Min`. The first one is called by each concurrent task, saves the output parameters and the version number; value `false` is passed as parameter `except` if the version has failed and an exception has been raised. The second procedure is called by each task and by the method body, it returns the decision and correct output parameters (if the adjudicator has got the majority vote). Our example in Section 5.2 demonstrates this. It is clear that though the adjudicator has to be designed by the FT designer, the functionalities described above require some application knowledge about the output results and about the abstract representation of version states.

If a faulty version has been found by the adjudicator, it is recovered by the corresponding task; if there is no majority vote, the failure exception is raised by the method body. To do the former the manager has to know the abstract representation of the internal version state. That is why the adjudicator has two service procedures `Keep_State` and `Give_Correct_Abstract_List_State` that provide the recovery of the faulty version. The first one is called by the manager when the execution of each version is completed; it transfers the current version state to the adjudicator. The second method is used to get this state from the adjudicator and to recover the faulty version.

The FT designer programs the package with the protected type in the following way:

```
package list_adjudicator is
   type adjudicator_decision is (recover, failure, ok);
   protected type list_adjudicator_t is
        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;
```

We believe that this approach to adjudication and version recovery is the only general one. But, any opportunity to adjust it and to make it more flexible would be of great use for its applicability. To this end, simple extensions of our scheme can allow FT designers to choose the most suitable way of doing recovery. It is clear that there is always a trade-off between early error detection and the efforts put in it; this approach makes it possible for the FT designer to control this trade-off.

One opportunity is to compare the results only after calls of some application methods (but not after each of them). This may be important when it is known that the results of some of them have 'higher priorities' and can affect critical calculations.

For some methods, only output parameters should be compared and the comparison of the entire version states may be omitted.

Sometimes the data required for version state comparison are different from those required for state recovery, and as comparison should be executed much more often and for all versions, it can be very effective to separate these two states and make the first take up as little memory as possible. To allow this, the application programmer should provide the following service operations in the version interface:

```
procedure Give_State_to_Compare  (l : access version_1_t;
                                  state : out abstract_list_state_t);
procedure Give_State_to_Recover  (l : access version_1_t;
                                  state : out extended_abstract_list_state_t);
```

Another opportunity to make our approach cheaper could be to mark a version as the faulty one and ignore it in further execution or to delay the recovery until certain time redundancy appears.

5.5. Version Distribution

Our scheme can be moved into distributed systems without serious modifications with the help of the Ada 95 Distributed Annex [Intermetrics 1995]. The Ada 95 distribution model specifies partitions as the units of distribution. They comprise aggregations of library units (e.g. packages). Ada distinguishes between several categories of these. For example, *pure* packages are those consisting of types and constants, *remote call interface* ones basically include several subprograms called from other partitions, *shared passive* ones are used for managing shared global data, etc. We can distribute versions by categorising packages with version objects as remote call interface packages. Package `class_list` should be categorised as pure. Note that Ada 95 exceptions are propagated through remote procedure calls; thus, the manager will catch them as it does in the scheme above. We will not go deeper in the

discussion of these problems because this would require going into finer detail of Ada 95.

The manager can be distributed in the same way by categorising package `class_ft_list` as the remote call interface one. But it would be more difficult to distribute the adjudicator because it is implemented not as a procedure, package, or class but as a protected object: Ada 95 does not allow remote entry calls (implementing it as a class would require some class-oriented synchronisation which is not provided by the language).

To deal with the problems of the manager or adjudicator replication, the approaches to package replication thoroughly discussed in [Burns & Wellings 1995] can be used. We believe that this service should be implemented as an underlying level hidden from our scheme.

6. Discussion

In distributed and concurrent OO systems when several methods of a diversely implemented object can be called at the same time, special care should be taken to guarantee version consistency. We do not address this problem; our assumption is that only one method of the object can be executed at a time. To tackle this problem in the most general way, the manager has to guarantee the same deterministic order of method calls for all versions. In this case the states of all versions will be the same.

Another problem arises in using NVP in concurrent OO languages which allow application-dependent synchronisation constraints. We believe that the best solution is to keep them only on the manager class level; versions should be called directly from the manager without restrictions caused by the constraints. The constraints should be manipulated in a consistent way because all versions are supposed to do this: only the modifications made by the versions which have been proved to be correct should be taken into account.

One more problem is the recovery and consistency of objects used by versions. There are basically objects of two kinds: objects internal for each version, which are part of the version design, and external objects, which are used by all versions (the same sets of them should be used by all versions). Objects of the first kind are encapsulated in the version, and it is not difficult to guarantee their properties. It is much harder to tackle objects of the second kind. They have to be replicated N times, and each version should manipulate its own replica. It may be better to pass these objects as parameters to the diversely implemented objects because our mechanism replicates

them automatically and returns one correct set of results. Otherwise, it is much more difficult to recover the objects used by the faulty version. Moreover, it is getting hard for the outside software to use these replicated objects: one solution could be to replicate them each time a method is called and to leave only (the correct) one of them after the method is completed. If all replicas are locked during the execution of a method, then object replication makes it possible to guarantee that the executions of all versions are isolated and no *information smuggling* among versions and between them and the outside world is possible.

Apart from the problems mentioned above, in our future research we would like to address the following issues: implementing managers for several dynamic SFT schemes (e.g. SCOP [Bondavalli *et al.* 1993]) which are based on parallel version execution; introducing features which allow FT designers to choose from these schemes; making the adjudicator object more re-usable by using generics or parameters; applying our general approach to Java [Sun 1995], which has concurrency on the language level (newly-created threads can be used to start the execution of versions, objects with synchronised methods can be used for implementing controllers and version synchronisation, etc.); implementing the distributed Ada 95 and Java NVP schemes (the latter will rely on the Java RMI [Sun 1996]). Another important direction will be designing meta-object protocols for introducing NVP in OO languages [Stroud & Wu 1995]: we believe that our detailed analysis of the NVP execution makes it clear which functionalities can be implemented on the meta-level and which concerns should and can be separated.

7. Conclusions

We regard our approach as an important step in applying the general framework in [Xu *et al.* 1995] to standard OO languages and in investigating practical approaches to using SFT in them. We concentrated on class diversity only because here the units of software diversity are the units of system structure and design. This makes it possible to introduce SFT on the object level and to explain all details of NVP in OO terms. Moreover, as objects can be of different scale, diverse software of any granularity can be designed in a very flexible way. One of the implications of using NVP in OO systems is that, generally speaking, the adjudicator has to compare version states because comparing version results is not always able to prove that the execution has been correct (moreover, the output results may even not exist because of data encapsulation adhered to in OO programming).

We have given a complete discussion of the NVP scheme and, in particular, a detailed analysis of the concurrent aspects of NVP support (our proposals are demonstrated

using Ada 95, a language which is widely used for safe-critical applications). It has been shown how to start the execution of several versions in parallel, how to synchronise them when they are completed, how to guarantee the consistency of the adjudicator execution when several versions work in parallel. We have offered a practical approach to the faulty version recovery based on the intermediate representation of the states of all versions and explained why it seems natural to extend the traditional functionality of the adjudicator for it to deal with this recovery. We have shown in which respects the NVP adjudicator has to be application-specific.

Our intention is not just to design re-usable components (the adjudicator, as we have demonstrated, is not really re-usable very much) but to separate concerns clearly. In particular, the adjudicator is separated from the controller and from versions; the application methods of versions are not changed; versions and the manager are derived from the same application abstract class. This separation is at the same time a division of responsibilities among programmers of different groups.

Acknowledgements. This research has been supported by DeVa (Design for Validation) ESPRIT project and by EPSRC/UK DISCS project. Thanks go to my colleagues in these projects.

References.

[Avizienis 1985] A. Avizienis, “The N-version Approach to Fault Tolerant Systems”, *IEEE Trans. Software Eng.*, 11 (12), pp.1491-501, 1985.

[Bishop 1995] P. Bishop, “Software Fault Tolerance by Design Diversity”, in *Software Fault Tolerance* (M. R. Lyu, Ed.), John Wiley & Sons, 1995. pp.211-30.

[Bondavalli *et al.* 1993] A. Bondavalli, F. Di Giandomenico and J. Xu, “Cost-Effective and Flexible Scheme for Software Fault Tolerance”, *Comp. Syst.: Science and Eng.*, 8 (4), pp.234-44, 1993.

[Burns & Wellings 1995] A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge University Press, 1995. 396p.

[Chang & Dillon 1994] E. Chang and T. S. Dillon, “Achieving Software Reliability and Fault Tolerance Using the Object Oriented Paradigm”, *Comp. Syst.: Science and Eng.*, 9 (2), pp.118-21, 1994.

[Intermetrics 1995] Intermetrics, *Information technology - Programming languages - Ada. Language and Standard Libraries. ISO/IEC 8652:1995(E)*, Intermetrics, Inc., 1995.

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

[Lee & Anderson 1990] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, Dependable Computing and Fault-Tolerant Systems, 3, Springer-Verlag, Wien - New York, 1990. 320p.

[Purtilo & Jalote 1991] J. M. Purtilo and P. Jalote, "An Environment for Developing Fault-Tolerant Software", *IEEE Trans. Software Eng.*, 17 (2), pp.153-9, 1991.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Trans. Software Eng.*, SE-1 (2), pp.220-32, 1975.

[Rubira & Stroud 1994] C. M. F. Rubira and R. J. Stroud, "Forward and Backward Error Recovery in C++", *Object Oriented Syst. J.*, 1 (1), pp.61-85, 1994.

[Stroud & Wu 1995] R. J. Stroud and Z. Wu, *Using Metaobject Protocols to Satisfy Non-functional Requirements*, Computing Dept., University of Newcastle upon Tyne, N°TR 533, 1995.

[Sun 1995] Sun, *The Java Language Specification. Version 1.0 Beta*, Sun Microsystems, Inc., 1995.

[Sun 1996] Sun, *Java Remote Method Invocation Specification. Revision 0.9.*, Sun Microsystems, Inc., 1996.

[Tso & Avizienis 1987] K. S. Tso and A. Avizienis, "Community Error Recovery in N-Version Software: a Design Study with Experimentations", in *FTCS-17*, (Pittsburg, USA), pp.127-33, 1987.

[Tso & Shokri 1995] K. S. Tso and E. H. Shokri, "ReSoFT: a Reusable Software Fault Tolerance Testbed", in *Int. Pacific Rim Fault Tolerance Symposium*, pp.98-103, 1995.

[Xu *et al.* 1995] J. Xu, B. Randell, C. M. F. Rubira and R. J. Stroud, "Toward and Object-Oriented Approach to Software Fault Tolerance", in *Fault-Tolerant Parallel and Distributed Systems* (D. Avreski, Ed.), IEEE CS Press, 1995. pp.226-33.

[Xu *et al.* 1996] J. Xu, B. Randell and A. Zorzo, "Implementing Software-Fault Tolerance in C++ and OpenC++: An Object-Oriented and Reflective Approach", in *Int. Workshop on Computer Aided Design, Test, and Evaluation for Dependability*, (Beijing, China), pp.224-9, International Academic Publishers, 1996.