Faulty Version Recovery in Object-Oriented N-Version Programming

A. Romanovsky
Faulty Version Recovery in Object-Oriented N-Version Programming

Alexander Romanovsky
Department of Computing Science
University of Newcastle upon Tyne

Being able to recover faulty versions in N-version programs would be vital for many long-running applications. Developing a recovery feature, however, is a very complex and error-prone task, which we believe has not received adequate attention. Although many researchers are aware of the importance of version recovery, there are very few schemes which include these features. Even when they do, they rely on an ad hoc programming and are not suitable for object-oriented systems. We believe that developing systematic approaches here is crucial. In this paper we formulate a general approach to version recovery in class diversity schemes, which is based on the concept of the abstract version state. The approach extends our recently-developed class diversity scheme (Romanovsky, 1999) and relies on important ideas motivated by community error recovery (Tso and Avizienis, 1987). Our diversity scheme includes two-level error detection which allows error latency to be controlled. To use it, special application-specific methods for each version object have to be designed, which would map the version state into the abstract state and, at the same time, form a basis for one-level version recovery. We discuss the approach in detail, compare it with the existing solutions, show additional benefits of using the abstract object state. Our intention is to outline a disciplined way for providing version recovery and thus make it more practical. We discuss several promising approaches which can be used for developing new structuring techniques incorporating the abstract object state concept.

Keywords: software fault tolerance, N-version programming, version recovery, object-orientation, structured programming

1. N-Version Programming and Object-Orientation

1.1. Software Fault Tolerance

Developing modern computer systems in many application areas is impossible unless means for tolerating faults are included. There are many areas in which system fault
tolerance is vital because high dependability requirements cannot be met without it. There is a lot of evidence to indicate that tolerating software faults is becoming predominant in providing system fault tolerance. One reason for that is that complex modern software always contains faults. In the last thirty years many techniques have been proposed for dealing with design faults in software; all of them employ some form of software diversity. Two general techniques have received the most attention: recovery blocks and N-version programming (a thorough discussion of advantages and disadvantages of these approaches and their comparison can be found in (Lee and Anderson, 1990)).

A recovery block (Randell, 1975) consists of a block of application code (the primary routine), several alternate routines executing the same functionality and an acceptance test. If the primary routine fails, which is detected by the acceptance test, the alternates are tried sequentially. This technique uses backward error recovery to return the data affected by the execution of the erroneous routine into the state they were in at its start and then runs the next alternate.

Unlike recovery blocks, N-version programming (NVP) (Avizienis, 1985) employs masking redundancy: N equivalent modules (called versions) are implemented independently and run concurrently. The results of their execution are adjudicated by a special component that defines the correct majority result and eliminates the results of the versions in which design faults have been triggered. Special care should be taken to decrease the common mode failure among versions; this is why one uses diversity in methodologies (languages, tools, etc.) while developing them. The correct functioning of an N-version program very much depends on how the run time support works, its responsibility being to run versions in parallel, synchronise their completion, pass the results to the adjudicator, etc. Many schemes based on the general ideas of NVP have been developed since then. Different forms of NVP have been used in various industrial applications (Bishop, 1995): using NVP is a practical issue today. In this paper we will concentrate on some important aspects of this technique.

1.2. Structured Approaches to Diversity Design

Most practical systems with high dependability requirements are complex by nature; employing diversity can make them even more complex, which may undermine the idea of using diversity if appropriate measures are not taken. We believe that the following issues are vital for developing diversity schemes.
First of all, the diverse components should be developed in a structured way. Which means that applying software diversity should be coupled with the structuring techniques used in system development. Units of system design/structuring should be the same as units of diversity.

Secondly, it should be possible to apply diversity schemes recursively (Randell, 1983). This allows any system subcomponent to be designed diversely, which applies recursively for any subcomponent of this component (Xu et al., 1995).

Another vital issue is diversity encapsulation. Complex systems should be developed in such a way that diversity applied to develop any components is hidden from outside components. This means that diversity control, as well as diverse software controlled by it (e.g. versions), are to be hidden inside components which offer the conventional application interface only.

The fourth important principle is guaranteeing the independence of version (or alternate) design. The scheme itself should support the independence of version development. Ideally, version developers should know nothing about the fact that they are developing one of many versions, the design of which should not be coordinated or restricted in any way.

The last principle to be emphasised is that diversity control and support should be separated as much as possible from the application code of the components.

These principles are fundamental for developing all modern diversity schemes because they make it possible to cope with the increasing complexity of the diverse system by allowing a clear separation of concerns while developing these systems, independence of the version design (which, generally speaking, should decrease the probability of the version common failures) and a high level of flexibility in achieving system dependability (components can have different dependability requirements attached). There are, for example, schemes which associate diversity with procedures (e.g. (Purtilo and Jalote, 1991)), but recent developments in modern design techniques and languages require associating software diversity with object-oriented (OO) system development. In the following, we will concentrate on topics related to using software diversity in object-oriented systems.

1.3. Software Diversity and Object-Orientation

Applying NVP in developing classes and objects, which are the units of the system design and structuring in OO systems, can offer many advantages and makes it easier to adhere to all above-mentioned structuring principles. Having analysed recent
research on using NVP in OO systems (Chang and Dillon, 1994; Issarny, 1993; Romanovsky, 1999; Xu et al., 1995; Xu et al., 1996), we can conclude that associating NVP with OO programming facilitates the system development and makes its parts (both versions and the NVP control) re-usable. To make use of this association, we have to view the units of the diverse design as the general units of system development. This makes it possible to use all advantages of object-orientation while developing diverse software without breaking the system structure or unnecessarily complicating component interfaces and with a clear diversity encapsulation. Moreover, we can have a fine granularity of diversity and associate different dependability requirements with different system components.

When we apply general NVP in the context of OO programming, we have to think about all scheme components in OO terms: the component to be designed diversely, versions, adjudicators, the scheme manager (the controlling mechanism), input/output parameters, internal version data. The crucial idea in using class diversity is to independently develop N version classes which meet the same class specification.

The paper (Xu et al., 1995) proposes a general OO framework and thoroughly discusses different types of diversity. It shows, in particular, why applying diversity on the class level is the way which best suits the concept of NVP. The authors introduce the main idea of the re-usability of service components (classes) and discuss interfaces and functionalities of three pre-defined classes for versions, adjudicators and managers.

The approach presented in (Issarny, 1993) allows using class diversity in the concurrent language Arche. This makes it possible for the authors to demonstrate many important details of the NVP scheme execution and control, which are often left out of the papers reporting their results using sequential languages. In Arche the general multi-operation feature, which is part of the run-time support, is used for calling diverse implementations of the same class. The results are built of component contributions and reported to the caller.

The paper (Rubira and Stroud, 1994) discusses how backward and forward error recovery can be used in C++. In particular, it proposes employing diversity on the level of C++ classes. We believe it is very proper to use a set of variant objects, each an instance of a different concrete sub-class of the base class.

It is extremely unlikely that any widely used languages will have any fault tolerance features in their standard support. This is why one of the vital issues for using design diversity in practical systems is the ability to do this using standard languages and environment (Randell, 1993). It may appear that this can complicate system
development because diversity control will in this case be programmers' responsibility. Fortunately, it does not seem to be the case: it is shown in (Romanovsky, 1999; Rubira and Stroud, 1994; Xu et al., 1995; Xu et al., 1996) that using software diversity in OO systems can be essentially facilitated by employing re-usable and predefined classes.

1.4. NVP in Concurrent OO Languages

We have recently developed an NVP scheme intended for concurrent OO languages (Romanovsky, 1999) which addresses the problems of version control (the synchronisation, concurrency and distributedness of the diverse class operation), proposes a clear internal structure of the diversely-implemented (DI) class and discusses interfaces and functionalities of its subcomponents. Another aim of this research is to discuss engineering steps for applying class diversity in standard languages (we use Ada).

In developing this scheme we have followed all above-mentioned principles of structured NVP. Developers of version objects use the same class specification. Versions are declared (hidden) inside the DI class. In addition, this class has the adjudicator object and the manager object inside it (see Figure 1). In our scheme we clearly separate their functionalities and describe their interfaces. The manager controls the execution of the scheme: it starts when a method is called, calls versions and the adjudicator object and returns results to the caller. Version programmers do not need to take notice or to know of other versions or the fact that they are parts of NVP. All components inside the DI object are built by inheriting from the provided standard classes and from the application class to be designed diversely. When a method of a DI object is called, the manager calls this method in all N versions in parallel and waits for their results to pass these to the adjudicator. The adjudicator compares them and returns the majority results, so that the correct output results can be passed to the caller. If there is no majority, the manager signals a failure exception.

To show how the scheme works we will go into the way the general ideas of NVP are applied here. A. Avizienis and L. Chen (Avizienis and Chen, 1977) explain that N programs possess all the necessary attributes for concurrent execution, during which comparison vectors are generated by the programs at certain points. The program state variables that are to be included in each of these vectors and cross-check (cc-) points at which the latter are to be generated are specified along with the initial specification. Apart from cc-points, recovery (r-) points (Tso and Avizienis, 1987) are specified at which the complete states of programs are compared (these states have to be transformed into a common representation for comparison).
In our approach the states of the programs are those of version objects; the comparison happens when each method call is completed. There are two kinds of comparison vectors in our scheme as well (we have two-level error detection); they have different meanings and are used for different purposes. Vectors of the first kind include all method output parameters; their comparison is considerably simplified by the outputs of all versions having the same types. The use of these vectors prevents the DI object from outputting any incorrect information (which is the prime purpose of NVP). To minimise the error latent period, one can use vectors of the second kind: they include the entire states of object versions. To compare these states we use mapping functions (similar to r-points) which should be developed by application programmers and which are called by the manager to convert the state of each version into an abstract state common to all versions. This is clearly far more expensive than comparing outputs.

We consider it a very important advantage of our scheme that it is oriented towards concurrent OO languages of a wide range, including those used in practice (e.g. Java and Ada). We demonstrate this using Ada - the first standard OO concurrent language, which is often used in developing applications with high-dependability requirements. In this scheme all interfaces and components are developed on the language level and made re-usable.
2. Previous Work on Version Recovery

2.1. General Principles

In accordance with the NVP paradigm we have to treat version results which are in the minority as faulty, which, generally speaking, means that the versions which produced them have design faults. It is possible to exclude them from further use but this worsens the availability of the system, undermines trust in DI components and, in the long run, causes system failure. It has been recognised for quite a long time that this can be a disadvantage of NVP and that special features should be developed to resolve the problem. The general approach taken by many researchers is to try and find some ways to recover the data of the faulty version and keep using it. If we can do this, we can guarantee that all versions are ready for subsequent use. This solution relies on the assumption that versions only have few faults which are hit (turned into errors) very rarely and the recovered version can serve properly most of the time.

The first recovery scheme appeared over ten years ago: it was the community error recovery (CER) (Tso and Avizienis, 1987), whose central idea is that the state of the correct version can be used for recovering faulty versions. This idea seems to be the only feasible one and all research which followed has used it. Clearly, one cannot use backward recovery here because this would involve rolling back all versions, or, at least, considerably delaying the system execution and using additional state restoration features. Another approach could be to use exception handling inside each faulty version. But this does not make use of the existence of versions which have produced correct results and are in correct states, and it is difficult to guarantee (locally) that the states of all versions are the same without knowing the state of the correct version. We believe that this is recovery on a lower level: it should be used as much as possible inside each version before it produces results.

In the CER the state of the correct version is used to recover the faulty version into a state in which it would have been had it not failed. The authors view this sort of recovery as a particular case of forward error recovery (Tso and Avizienis, 1987). We think that it has some important properties of backward error recovery (Lee and Anderson, 1990) as well: it ignores the current state of the faulty version and does not use the analysis of the erroneous program state to find out which data are erroneous to recover them by executing some application-specific procedures. The only thing it has in common with forward error recovery is that it moves the version into a known forward state (corresponding to the current state of the correct version). Moreover, although it is presented in terms of exception handling (Tso and Avizienis, 1987), we believe that this recovery uses a very particular case of exception handling, which
does not have all of its attributes: the declaration of several application exceptions, a clear notion of the exception context, exception propagation and signalling, pre-defined and interface exceptions, etc. In fact, this recovery rolls the version state forward into one which is proven to be correct and extracted from the state of the correct version - this is well-known in research on checkpointing in replicated processes as roll-forward checkpointing (Pradhan and Vaidya, 1992).

The main problem with this sort of recovery are difficulties in developing mapping functions for transferring internal states among versions. Note that making versions more diverse (which is the entire point of applying diversity) potentially complicates recovery, since the more different they get, the more difficult it is to develop mapping functions.

2.2. State of the Art

The community error recovery (Tso and Avizienis, 1987) was the first and the most developed proposal which uses two types of points at which the execution of versions is synchronised. This is a two-level error recovery. The cc-points are used not just for comparing data in cc-vectors taken from all versions but also for partial version recovery: each version receives the adjudication result including the correct cc-vector. It is used for partial recovery because data in the cc-vector present a subset of the version state. Each version consists of several modules executed sequentially; each of them has several cc-points inside. R-points are inserted between modules and used for complete version recovery if partial recovery fails. At these points the complete internal state of each version is mapped into an intermediate format common to all versions, so that they can be compared and passed to the faulty version for recovery.

The authors of a general OO software fault tolerance framework (Xu et al., 1995) discuss the problem of faulty version recovery in OO programming and emphasise its importance and difficulties related to inter-version mapping.

We have encountered problems of different sorts trying to apply the community error recovery in the context of OO programming. This recovery contradicts some of the principles of structured NVP we have discussed and restricts the version design, for example, by requiring that the data in cc-vectors should be of the same types in all versions (Xu et al., 1995). We have found that it is not a trivial task to introduce it in OO programming in general, and in our NVP scheme (Romanovsky, 1999) in particular (a detailed comparison of our proposal with the CER is given in Section 5.2). In (Romanovsky, 1999) some initial suggestions are made as to how to provide version recovery in OO N-version programming. We introduce the concept of the
abstract version state into our OO scheme as an intermediate representation reflecting
the internal states of all versions (version objects), which can be used for both
comparing their states and for their recovery. In this paper we concentrate on this
problem attempting to find general and practical solutions. We describe our proposals
in detail, compare them with the existing solutions, show additional benefits to be
gained through our approach and discuss several promising system architectures
which can be used for implementing our proposals.

2.3. Problems with Recovery

We believe that there are many problems yet to be solved if version recovery is to
become applicable in practice. First of all, developing mapping functions is a complex
and error-prone task. It is not enough to say that programmers should develop these
functions and an intermediate representation for mapping the internal version states. It
is often impossible to do so without a special supporting methodology, structuring
techniques or architectures to facilitate this development and make it more systematic.
It is wrong to assume that recovery software is simple to develop and this can be done
without faults. Special effort should be devoted to designing clear and disciplined
ways of solving this problem. Secondly, in approaching version recovery, we should
follow the principles of structured NVP. Thirdly, this recovery should fit into OO
programming and be based on NVP schemes developed for this programming.

It is our belief that OO programming can help us reduce the complexity of this task
since it relies on structuring units which contain data and code (so we can deal with
the states of version objects in a more straightforward manner), allows re-usability
and receives a lot of attention from researchers and practitioners, which results in
many novel architectures and approaches being developed that can be useful for
introducing recovery in a systematic way.

3. Object-Oriented Version Recovery

Our NVP scheme (Romanovsky, 1999) uses a two-level error detection: it compares
either output parameters or complete version states. The first approach guarantees that
no erroneous information is smuggled outside and that the results are correct. The
second is used to decrease error latency; version states are compared here via a special
unified representation. We call it the abstract version state. The version programmer
develops an additional method, Give_State, which calculates the current abstract state
of the version using its internal state and returns it to the NVP manager (see Figure 2).
The manager passes the abstract states of all versions to the adjudicator. We use one-
level version recovery in our scheme: if the adjudicator finds a version to be faulty,
the manager passes the abstract state of the correct version into the faulty one by
calling its method Correct_State, which transforms this state into a correct internal
version state. Figure 3 offers a dynamic view of version recovery.

![Figure 3](image)

Figure 2. Recovery of a faulty version. Methods Give_State of all versions are called
before adjudication. The arrows show the directions of information flow.

We believe that the use of the same abstract version state for two purposes, to
dramatically decrease the latent error period and to recover faulty versions, not just
increases the applicability of our approach but shows a certain affinity between them.

![Figure 3](image)

Figure 3. Dynamic view on version recovery. The state of the correct version 2 is used
to recover version 1.

It is emphasised in (Xu et al., 1995) that version recovery is feasible provided a
relation (mapping) can be found between the internal data of different object versions.
In our approach we use an intermediate abstract representation of the internal states of
versions which is common to all of them, so that the internal states are mapped via
this abstract representation.
We assume that the state of the faulty version is encapsulated inside the version object, so it is enough to recover this object. The external specification, which is the same for all versions, is assumed to be correct: faults are made by version programmers. We only deal with software faults (although it is not difficult to extend our scheme for tolerating hardware faults, e.g. transient faults or node crashes, by version distribution - see a distributed implementation in (Romanovsky, 1999) - and by extending the manager functionalities). We assume that mapping functions and the abstract object state are correct; one way to achieve this is to use diversity to implement them.

The abstract version state can be described as a collection (e.g. a record) of several data representing different aspects of the current object state. Data of this type are used by the NVP manager when it controls version recovery. The data are hidden inside the DI object. Their type is application-specific and can be implemented as a class with only simple assignment methods, necessary for these data to be received and passed as input and output parameters in calling methods Give_State and Correct_State. This is why we need only one instance of this class for each DI object.

The object version interface should be extended in a systematic way to allow us to get the current abstract state of the correct version when necessary and to pass it as an input to the faulty version for its state to be recovered. To make this access systematic and disciplined, all version objects inherit from the abstract class which has the two methods. Besides, they all use the type describing the abstract version state to work with this state in these two methods.

In our scheme the adjudication of version outputs is a default; it does not require complex calculations because these outputs are of the same types for all versions. But we believe that in many situations it is safer to use the second option and compare the version states whenever a method is called (e.g., for critical objects, or when version states are not big or the computation of the abstract state is not complex). Although this can be expensive, the latent period is shorter and the probability that most versions will be correct in the next comparison is higher; otherwise many versions can go into faulty states undetected.

For some applications the manager can be made more flexible to allow more complex policies or a cost reduction. One policy is to compare the entire version states when particular methods are called, e.g., those that are more complex, so that errors are more likely to happen while executing them, or those which are executed as part of critical calculations (in which case the caller has to inform the manager of this). Another interesting policy is to separately treat situations when the outputs are the same (and as such assumed to be correct) but the version states differ. One more
interesting combination of two kinds of result comparison is by analysing method input and output parameters. For example, some methods may not have outputs if they are intended for changing the internal state of the object rather than producing results affecting the world outside the object. In this case we can either bypass adjudication or compare version states because if we bypass the adjudication we can dramatically increase the latent period. One can also choose a special policy for methods which do not change the object state at all (e.g. concerned with pure calculations), in which case we do not need version recovery. Other situations which may require special treatment involve methods which just return a subset of the object state (or a function of it) without updating it.

4. Abstract States and Mapping Functions

All approaches proposed for version recovery assume that some mapping functions have to be developed. This is the case for our scheme as well. Unfortunately, few of the authors discuss this in sufficient detail. This is a serious problem because it is the most difficult part of recovery. Our approach is based on the concept of the abstract version state: this state is used as an intermediate format by mapping functions Give_State and Correct_State. In this section we discuss the concepts of abstract version states and mapping functions and show how the development of the corresponding recovery features can be facilitated.

4.1. Informal Description

To start with several simple examples, let us consider a class list implemented diversely, for example, using arrays, hash tables, heaps or virtual files with a direct access. The abstract version state can be just a string of elements. Function Correct_State simply re-creates the internal data (including the list itself) of a version. Our analysis shows that for most basic data structures (queues, stacks, trees, etc.) the development of mapping functions is a feasible, although not always trivial, task. Another example could be classes which use algorithm diversity (bubblesort vs. quicksort) in developing methods, in which case the main internal data will be (nearly) the same. Note that there are many ways of employing diversity (Lyu and Avizienis, 1992). They include diversity in languages, compilers, operating systems, libraries, testing methods, for which mapping functions can clearly be trivial when the program (algorithm and data) diversity is not employed because it is sufficient to map different representations of the same data (most of the time they should be of the same or easily convertible types).
There are two main ideas on which we build our understanding of the abstract version state concept. First of all, this concept captures what all version objects have in common in the states of their data; they have it because they are implementations of the same class. The abstract version state is based on abstracting these different implementations. It is a general description of the states of all version objects.

Secondly, we have found that many researchers consider abstracting the object state (with different meanings) very fruitful and use it in various models. These models are different in many respects and used for different purposes. One may need a concept of this sort to formulate the predicates on the object states, pre- and post-conditions, object invariants, etc. Note that predicates are introduced into Lamport's TLA (Lamport, 1991) as boolean-valued state functions of variables, which is, again, a way of abstracting the program state. It seems reasonable to assume that some of these should be the same for all version objects diversely developed from the same specification. Another example is introducing abstract states in Eiffel (Meyer, 1988), with a suggestion that all predicates should be expressed using abstract states, so that different implementations of a class are viewed as correct if they conform to the same pre- and post-conditions and class invariants.

### 4.2. Formal Model

Each object has a state and an abstract state, which is a projection of the object state. Methods operate with the state. The abstract object state describes a conceptual state of all version objects which have been developed from the same specification of the abstract basis class. All versions describe the same phenomena and have the same behaviour; this is why they are, in a sense, equivalent and must have the same abstract state. Generally speaking, the abstract state is not just part of the object state, it is a function of it.

Let us assume that there is a set of objects \{V_j\}. Let S_j be the current state of object V_j (hidden from outside). I_{ij} and O_{ij} are the input and output parameters for method M_{ij} of object V_j (we view an in/out parameter as a pair of input and output parameters). Parameters are not part of the state. For each method M_{ij} of object V_j, output parameters are calculated as function F_{ij}:

\[ O_{ij} = F_{ij}(I_{ij}, S_j'). \]

where S_j' is the state of the object before the call of M_i.

The current state of object V_j changes from S_j' to S_j" as a result of the execution of any method M_{ij} with input parameters I_{ij}; this can be described as function G_{ij}:
\[ S_j'' = G_{ij}(I_{ij}, S_j'). \]

We introduce *abstract state* \( E_j \) of object \( V_j \) as a set of data reflecting (abstracting) the current state of \( V_j \) (see Figure 4). These data are to be seen from outside; they present a projection (usually reduced) or mapping of the internal object data. \( E_j \) is not part of object state \( S_j \); neither is it part of output/input parameters, because it is not related to any particular method call.

Let us consider a set of version objects \( \{V_k\} \) implementing abstract class A. Let \( V_1 \) be a faulty version (this fact is detected by adjudication). We can guarantee *the recovery of faulty version* \( V_1 \) if there is a correct version \( V_m \) from \( \{V_k\} \) such that:

1. There is a *copy function*, \( C_m \) of \( V_m \), which calculates the current abstract state \( E_m \) of \( V_m \) (\( C_m \) corresponds to method \( \text{Give\_State} \) above):
   \[ E_m = C_m(S_m). \]

2. There is a *recovery function*, \( R_l \) of \( V_1 \) (\( R_l \) corresponds to method \( \text{Correct\_State} \) above), such that:
   \[ S_l = R_l(E_m). \]

![Figure 4. Internal and abstract object states.](image)

We are using the state of a correct version object \( V_m \) and its copy function \( C_m \) to calculate \( E_m \) and recovery function \( R_l \) of the faulty version \( V_1 \) to calculate its correct internal state \( S_l \). States \( S_l \) and \( E_m \) can be thought of as collections of all data.
describing the current object state and the current abstract state. Generally speaking, function $R_l$ is not necessarily a one-to-one function because $E_m$ describes the conceptual state of a correct object version (which means that $R_m(C_m(S_m))$ may not be equal to $S_m$). The abstract state and copy functions must be developed in such a way that for any two correct versions $V_k$ and $V_m$ for which there are functions $C_k$ and $C_m$ the following is true:

$$\text{if } (E_k = C_k(S_k) \& E_m = C_m(S_m)) \rightarrow (E_k = E_m).$$

This means, in particular, that we can use any correct version object which has a copy function for recovering any faulty one. We will call this state the abstract version state.

It is clear that, to allow recovery of any faulty version, we need recovery functions $R_k$ for all version objects $\{V_k\}$ but there is no need in developing $C_k$ for all $V_k$. Copy and recovery functions can always be developed but their development may be a very complex task which requires detailed knowledge of how each particular version works. $R_k$ and $C_k$ are parts of the version interface; to develop these, one needs a specification of the abstract version state which is designed for each abstract class $A$ that is to be developed using class diversity.

### 4.3. Building Abstract Object State

There is an obvious contradiction between our intention to recover the internal state of a faulty version by using the internal state of a correct one and object state encapsulation. We propose a systematic way for developing the abstract version state, which is a complex application-dependent task to be solved by an independent programmer after the versions have been designed. Only this guarantees that the independence of the version development is not undermined and that mapping functions do not contain the bugs the version has. The programmer has to analyse these diverse implementations thoroughly and come up with a general abstract description of what the data in all these versions have in common such as to make the design of copy and recovery functions for each version feasible.

Several considerations can help here. One of them is the requirement that all versions must satisfy the invariants, and the pre- and post-conditions of the DI object. That means that these conditions should be formulated in a very general implementation-independent way, so it seems possible to build the abstract version state from abstractions used for describing these conditions.
There are several straightforward approaches to implementing the abstract version state. One is to use components of only basic types (e.g. real, integer, boolean, string) and keep them in a record type. Another approach could be to use the internal data of one of the versions as the abstract version state.

The most general approach is to design a special methodology for developing object versions in such a way that they have their states abstracted. This can give real benefit because it can simplify the development of recoverable versions. Although more research will have to be done in this direction, we have found several promising approaches based on recent research in OO system modelling.

The first approach is based on extensions of the layered object model (Bosch, 1995), which introduces the concept of the abstract object state (different from ours) to achieve several purposes - allowing clients to access the object state in a disciplined manner among them. This concept defines an abstraction of the object state which is placed at the object interface. The abstract object state has several dimensions, called states, each of which is calculated as a (mapping) function of the concrete object state and is to be developed together with the object itself. This state is a conceptualisation of the concrete object state and, generally speaking, is less complex than the latter in that it has fewer dimensions and smaller domains associated with dimensions. The general framework can be used to suit our purposes although the author's reasons for introducing this concept clearly differ from ours. For this, we need to extend the model by introducing recovery functions. Besides, we need a far more general view on the way the concrete states of several versions are abstracted. The development of the abstract version state should obey special rules in our case: it should follow the development of versions and be based on an analysis of their internal data and algorithms - only this makes it possible to extract what they all have in common and propose an abstract state, after which one can develop copy and recovery functions.

The PSL (Lea and Marlowe, 1995) is a framework for specifying dynamic and architectural properties of open systems. It extends the conventional way of describing object interfaces by introducing several new abstractions. In this framework an interface, which provides a basis for specifying capabilities in open systems at various levels of precision and formality, is a view on a family of components in terms of supported operation; attributes, which are used for describing abstract properties of several instances of an interface, are declared as auxiliary abstract functions. The underlying idea behind this approach agrees with the purposes of our research: we would like to "open" the implementation of version objects in a disciplined way to allow version recovery. In the solution based on the PSL we need to extend the interface of versions (without changing the interface of the DI object
itself). By doing this, we can add new properties (related to version recoverability) to version objects. It seems possible to extend the concept of attribute to describe the abstract version state and introduce this concept into the specification phase. All version objects should have the same attribute associated with their states, and the values of this attribute should be the same for all of them (implementations differ for different versions, though). Developing attribute functions corresponds to developing copy functions in our approach.

Another solution we would like to consider is based on using reflection (in the form of the meta-object protocol (Kiczales et al., 1991)). Meta-objects of all versions will contain an abstract object state (the same for all correct versions). When a faulty version is detected, the abstract state of this version is corrected on the meta-level using the abstract state of a correct version. We need a special type of two-dimensional reflection upon object data here, in which any updates of abstract data on the meta-level cause automatic updates of the base level data because abstract data on the meta-level describe and reflect upon the state of the base level object. Moreover, this reflection should be able to update the abstract version state when the state of the base level object has been updated during the execution of a method. In practice, one can implement this using reflection upon all accesses/updates of the object states, which many systems allow. Note that not all updates of the state of the base level object cause updates of the abstract version state. The most practical approach is to implement all NVP managing (including the faulty version recovery) on the meta-level (similar to the object replication support in the FRIENDS system (Fabre and Perennou, 1997)). In this case we will reflect upon all application calls of the DI object and intercept them. The meta-object of this object is the NVP manager encapsulating the adjudicator object. The application interface of version objects remains unchanged: it does not include copy and recovery methods which are now hidden inside the implementation of the reflection capability. If, after version objects have completed the execution of methods, the manager learns from the adjudicator that there is a faulty version, it corrects its abstract state using the abstract state of a healthy version. Reflection is clearly just a structuring mechanism, so employing the mechanism proposed still requires the implementation of mapping functions between the state of the base level object (version) and its abstract state. These mapping functions are application-specific and will be incorporated into the implementation of the ways reflection works.

These approaches allow us to develop structuring frameworks which include concepts required for version recovery, impose a disciplined way of providing recovery and separate recovery-related concerns from application ones. Clearly, none of these approaches can assist programmers in defining the abstract version state common to
all versions (a complex task whose solution requires human intelligence and expertise in analysing developed versions). We think it is hardly possible to automate or support this phase in any way unless we are prepared to restrict version developers in their design by imposing constraints describing what all versions should have in common; this, however, would contradict the idea of version design independence which is vital for achieving version failure independence (Avizienis, 1985).

### 4.4. Applications of Abstract Object State

When developing mapping functions and abstract object states, there are several attractive features we can provide, apart from version recovery and defensive programming (the latter due to a decrease in the error latent period). First of all, we can do background run-time testing of versions which are not currently executed by comparing their abstract states. Secondly, if we can save the abstract state in stable storage we can use backward error recovery of the faulty version. Method Give_State can be called to obtain the version state before the method call. If this state is kept in stable storage, we can roll the faulty object back and re-try its method as a means of recovery. Another approach could be to roll all versions back and re-try them (which can be useful if there is no majority). A very sophisticated scheme which uses recovery blocks and NVP interchangeably (because we can re-create the state of any version if we have a correct abstract state of only one of them) can be developed as well. Another possibility would be to develop a roll-forward recovery scheme (similar to the scheme in (Pradhan and Vaidya, 1992) but for tolerating design faults). In 3VP it will work as follows: first, the methods of two versions are called and their results compared; if they differ, we re-create a previous state of the third version using the abstract state derived from that in which one of the two versions was before the method was called, and call its method to obtain its result in order to find the majority and the faulty version.

### 5. Discussion

#### 5.1. Our Approach

Our scheme extended by version recovery conforms to all principles of structured NVP programming formulated in Section 1.2. Classes/objects are units of diversity, each of them can have objects developed diversely inside. Diversity is hidden from the caller and clearly encapsulated inside classes. Each version confines all of its errors and is viewed as a recovery region. Versions are designed independently in this scheme and they do not coordinate their execution with other versions. The NVP
manager (a re-usable component) controls the execution of versions including their recovery. All information flowing from and to versions passes the manager, which guarantees that erroneous information does not return to the caller. All version functionalities are provided through their interfaces. In this approach we do not restrict version developers in any way.

One extension of our approach could be to introduce two kinds of abstract states: one for error detection and another for recovery. They can be specially-developed for these purposes and capture version application-specific characteristics better. Apparently, the first one will be simpler and provide a better error coverage than comparing method outputs (whose purpose is to prevent outputting erroneous information rather than quick error detection).

Mapping functions can be very complex, and it may be important to use diversity to tolerate faults here. The general approach would be to design each of them using NVP. But it seems reasonable to pay special attention to developing these functions. For example, one can test them intensively at the development stage using circles of mapping functions. Here are two examples:

Give_State(V1), Correct_State(V1);

Give_State(V1), Correct_State(V2), Give_State(V2), Correct_State(V1).

In both cases the new state of version V1 is likely to be different from the old one but the recreated V1 should behave in the following computation as if no mapping functions had been applied.

Let us compare our approach with one that relies on developing functions which directly map an internal version state into the internal state of another version. Our approach offers many advantages: it relies on principles of structured NVP, allows a simple structure of the DI object, facilitates adjudication, etc. The following consideration shows, however, that it requires more mapping functions for objects with 3 versions; but is better for those with more versions. Let us assume that we have N versions and that K of them can fail during the execution of a method. In the worst scenario, for systems without the abstract object state one needs K mapping functions to recover a faulty version, which gives a total of (K*N) functions. For those with the abstract state one needs N functions for mapping the abstract state to the state of each version and (K+1) functions for mapping the version states to the abstract state, which amounts to (K+1+N) functions. Our approach is worse for 3VP (K=1): it requires 5 functions against 3 functions for the first approach. For 5VP (K=2) we require 8 functions against 10. For 7VP the numbers are: 11 and 21.
5.2. Comparison with Community Error Recovery

Our approach shares several basic principles with the community error recovery (CER), but we have found that the latter has some problems and cannot be easily applied in OO programming.

In the CER the units of diversity are not clearly defined, they are not units of system structuring: there are N diversely-implemented programs, which consist of modules of the same functionality executed in the same order (it is assumed that they meet the same specification and are, as well as the N programs, implemented diversely). This approach is not recursive; it restricts the version design and does not allow finer granularity of diversity. Here diversity is not encapsulated because modules have to be split into the same functional steps, so that each of these steps finishes with the execution of a cc-point at which versions are synchronised. There must be the same number of cc-points in modules of different versions and they must include the same set of data (which have to be of the same types in all versions). The design of N programs is restricted as well because they have to have the same modules, the execution of which finishes with the execution of r-points.

Our scheme uses a two-level error detection: comparison of outputs and of version states. Each of them has a clear meaning: the first prevents the object from outputting erroneous information, the second decreases error latency as much as possible given the OO structure of the system. The former does not restrict the version design in any way because the same data are always output by all versions. The latter relies on abstract version states and mapping functions which we use for version object recovery as well.

The CER uses a two-level error detection as well but with a different meaning. At r-points it basically does what our scheme does while comparing abstract version states. We believe, however, that there are some problems here with identifying the program state to be used at r-points: it is not feasible to deal with the state of the entire program and there is no structuring support to clearly define which subset of these data should be used. At cc-points the CER compares some subsets of version data (cc-vectors) which have to be the same for all versions (as opposed to our scheme in which outputs are compared). It is not clear how to choose these subsets and we believe it is a serious restriction to require that all versions should use a subset of the same data. Moreover, these data have to be "very important" because partial recovery in the CER relies on them. This recovery is based on assigning correct values to a subset of data of a faulty version. It seems to be really difficult to choose a subset of data which can help in both error detection and partial recovery (these requirements may even
contradict each other). We believe that the concept of partial recovery in the CER is not generally applicable and contradicts the idea of structured NVP because the entire version has to be recovered since it is the error-confining region. In our scheme we recover the complete state of the version object.

In OO programming we have only one point at which adjudication and recovery can take place: completion of method calls. This is why we allow two kinds of error detection at these points but rule out the possibility of partial recovery: it always has to be a complete recovery. We use a one-level recovery which is similar to the CER recovery at r-points: the internal data of a correct version are transferred into an intermediate abstract form common to all versions and passed to the faulty version by the executive. In our scheme, however, the data to be recovered are clearly identified.

6. Conclusions

It is our belief that for many applications the entire idea of using NVP can be undermined if version recovery is not provided. The three main contributions of this paper are as follows. First, we propose a general framework for version recovery in OO systems which is based on a two-level error detection and a one-level error recovery. Secondly, the concept of the abstract version state is developed, formally described and analysed in the context of OO programming. Thirdly, practical approaches are proposed for developing architectures incorporating the concept of the abstract version state.

One intention of this paper is to emphasise the importance and the complexity of version recovery. We believe that it is dangerous to expect these problems to be solved in an ad hoc manner. The solutions we have proposed are not simple because the problems are extremely complex. More research will have to be done in many directions discussed in this paper, but we hope that it forms a sound basis for further work.

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

References.


