Abstract Object State and Version Recovery in N-Version Programming

Alexander Romanovsky
Abstract Object State and Version Recovery in N-Version Programming

Alexander Romanovsky

Department of Computing Science, University of Newcastle upon Tyne
alexander.romanovsky@newcastle.ac.uk

Abstract

This paper deals with the use of software diversity, specifically, N-version programming (NVP) in object-oriented (OO) systems. We formulate the problem of faulty version recovery and show how our NVP scheme, developed recently, can be extended to solve it. Our approach relies on using the abstract version state, which represents a common general description of the states of all correct version objects. The recovery consists in mapping the state of a correct version onto the state of the faulty version via the abstract state. We introduce a formal description of our model and show that many ideas related to object state abstraction can be found in the existing research on OO programming. We discuss extensions of LAYOM and PSL as promising practical approaches for developing recovery features in OO programming. As an alternative solution, we propose a meta-object architecture and a related protocol which can facilitate the solution of the object version recovery problem. The paper finishes with a brief discussion of engineering steps which have to be done for developing recoverable version objects and of some approaches which can improve the re-usability of the scheme proposed.

Keywords: software fault tolerance, software diversity, N-version programming, version recovery, object-oriented programming

1. Introduction

In spite of all effort devoted to improving the quality of software systems, the goal of meeting high dependability requirements cannot be achieved without accepting that there are always design faults in software, that hardware can fail, operators can misuse systems and environment can misbehave. Analysis of failures in the existing systems shows that the ratio of failures caused by software faults (bugs) is increasing [1]. All experience in using different software engineering techniques and paradigms demonstrates that it is not possible to develop a reasonably large program without bugs. Object-orientation may not be an exception, either [2]. Fault tolerance techniques are traditionally used for dealing with such problems. The main underlying idea in recent development of fault tolerance techniques for object-oriented (OO) systems is to apply well-known techniques to this particular area and to make use of the peculiarities of these systems (e.g. the fact that these systems are structured out of objects, i.e. entities which combine code and data; their re-usability, etc.) [3, 4].

N-version programming [5] was the first technique developed for employing software diversity to tolerate software faults. N versions of the same (sub-) program are developed diversely from the same specification (there are many ways of achieving diversity here, e.g. the use of different programmers, languages, algorithms, etc. [6]). The versions run in parallel and the results of their execution are adjudicated by a special component which calculates the majority result to be returned as the result of the execution of the diversely implemented program.
Recently several authors [3, 7, 8] have developed OO schemes for N-version programming. The units of diversity in these schemes are classes/objects. The idea is to implement several class versions which meet the same specification of the basic class. The authors of [3] call this approach class diversity, as opposed to method diversity which assumes diverse implementation of the same method. In class diversity the implementation of the basic class is built out of several object versions.

2. Faulty version recovery

It is accepted that the generally popular N-version programming (NVP) has several disadvantages (see, for example [9]), one of which is that the versions which are in minority and likely to be in erroneous states cannot be easily recovered and, generally speaking, should be ignored in the following computations (this is why this technique is sometimes called masking diversity), which causes the degradation of system availability. The only approach proposed for dealing with faulty version recovery is community error recovery [10]. The general idea here is to use the states of the healthy (correct) versions to recover the faulty ones. To follow this approach, version programmers have to insert inside each of the modules (that an N-version program is built of) the so-called cross-check (cc-)points, where a set of variables from all versions are compared and the adjudicated results passed to all versions for recovery. At the exit from the module the so-called recovery (r-)points are established, where the complete states of healthy versions are passed to the faulty versions if the recovery at cc-points was not successful; special application-dependent algorithms are used at r-points to map version states. Later, many researchers agreed that using inter-version mapping is the only realistic way to recover versions (see [3, 8]). Our analysis shows that community error recovery has several disadvantages and cannot be directly used in OO programming. First of all, this structuring is not suitable for OO systems; secondly, this approach contradicts data encapsulation and imposes additional restrictions on version development; thirdly, generally speaking, the claim that the recovery of a subset of version variables should be able to help when the state of the whole faulty version is corrupted is not and cannot be grounded; lastly, version coordination is executed by versions themselves which unnecessarily complicates the job of version developers.

Figure 1. Internal structure of a diversely-implemented object
Recently, we proposed a general OO scheme for N-version programming and outlined an approach for version recovery [8]. Within our scheme, versions are developed in a special way by inheriting from the basic class and from a special abstract service class which extends version interfaces to allow faulty version recovery. This abstract class has two methods: Give_State and Correct_State. The implementation of the basic class includes an NVP manager as its central (re-usable) component, which coordinates version executions as well as version recovery (Figure 1 shows the execution of application method M1). The manager can call the adjudicator to compare either the output results of a method call or the complete version states (using service method Give_State of each version to extract their states). The latter decreases the latent period of errors and is similar to the purpose of r-points in [10]. The same method Give_State, if necessary, is used for recovery after adjudication: it returns a state of the healthy version to be transferred to a faulty one (see Figure 2). This state is presented in an intermediate (abstract) format common for all versions. Our scheme is demonstrated in Ada 95, an OO language which has well-developed concurrency features. We have designed a set of abstract classes and re-usable components which help in applying diversity on the class level.

Inter-version mapping, in general, and abstract state development, in particular, is a complex error-prone problem, which, if not solved properly, can undermine the whole idea of faulty version recovery. Unfortunately, this problem has never been discussed before and the responsibility seems to have been left with application programmers. We would like to avoid any ad hoc programming here. The general purpose of our research is to propose a development method which would allow programmers to design object versions in a special way which makes them mappable and recoverable.

We believe that recent research in OO can serve as a sound ground for developing these systematic approaches. We do not have a final solution for this problem yet but we have found that OO programming provides very useful features and techniques which can help us to attack the problem. In this paper we would like to discuss some of them.

3. Formal description

Each object has a state and an abstract state, which is a projection of the object state. Methods work with data, which are parts of 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 diversely-implemented versions describe the
same phenomena, and their behaviour is the same; 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 one of its functions (for each version object).

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_i 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_i of 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 V_j changes from S_j' to S_j'' as a result of the execution of any method M_i with input parameters I_ij; this can be described as function G_ij:

\[ S_j'' = G_{ij}(I_{ij}, S_j') \]

We introduce \textit{abstract state} E_j of object V_j as a set of data reflecting (abstracting) the current state of V_j. These data are to be seen from outside: they represent 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_l be a faulty version (this fact is detected by the adjudication). We can guarantee the recovery of faulty version V_l if there is a correct version V_m from \{V_k\} such that:

1. there is \textit{copy function} C_m on V_m, which calculates the current abstract state E_m of V_m (C_m corresponds to method Give_State above):

\[ E_m = C_m(S_m) \]

2. there is \textit{recovery function}, R_l of V_l (R_l corresponds to method Correct_State above) such that:

\[ S_l = R_l(E_m) \]

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). These functions always exist but their development may be a very complex task which requires detailed knowledge of how each particular version works. The abstract state is calculated after the adjudication which allows us to detect which versions are correct and which are faulty: 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_l to correct its internal state S_l.

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 it true:

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

That, in particular, means that we can use any correct version object, which has a copy function, for recovering any faulty one. We will call this state \textit{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. R_k and C_k are parts of the version interface, and version designers should develop these as object...
methods. To do this, they need a specification of the abstract version state which is designed for each abstract class A that is to be developed using class diversity.

4. OO programming and faulty version recovery

We believe that OO programming can considerably help in developing approaches for version recovery. First, recovery should rely on manipulations with data, and data are part of structuring units in OO programming. Secondly, OO programming secures a clear separation of specification/interface and implementation: the interface describes operations which can be performed (including ones related to recovery). Abstract (virtual) classes and inheritance allow us to re-use some components and to extend interfaces in a standard way to make version objects mappable and recoverable. Research in OO programming is an evolving issue in which enormous effort is invested by researchers and practitioners; this is why there are many novel techniques developed here, e.g. multiple interfaces, reflection, delegation, to mention just a few.

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 performs version recovery. The data are hidden inside the basic object which is developed diversely. The type of these data 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 version object methods Give_State and Correct_State. This is why we need only one instance of this class for each basic 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 this state 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, all of them use the same type, describing the abstract version state, to work with this state in these two methods.

Abstract version state is based on abstracting different implementations of the same class. It is a general description of the states of all version objects implementing this class. Let us consider several examples. Assuming that there are several implementations of a set object (using hash table, list, array), the abstract version state can be a record containing all elements which are in a set now. Another example is the use of language diversity while implementing a class. Let us assume that there are three versions implemented in Ada, C++ and Java and that they implement the same algorithm. If the NVP manager is programmed in Java, we can use a sufficient subset of internal data of the Java version as an abstract version state, the mapping functions for the Ada and C++ versions should take into account some rules for type conversions between Java and Ada and between Java and C++. A third example is a collection of elements in which the main internal data structure (storing all the elements) is presented in the same way in all versions (for example, as an array). Let us assume that a sort method is implemented differently in different versions. If one of them fails, we have to pass a sorted array from the correct version to a faulty one. This will suffice for version recovery.

We have found several models in which some abstractions of the object state are introduced. They are different in many respects and used for different purposes. One may need them to formulate the predicates on the object states, pre- and post-conditions, object invariants, etc. Note that predicates are introduced into Lamport's TLA [11] 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 [12], with the idea that all predicates should be expressed using abstract states, so that all correct (consistent) implementations of a class are viewed as identical if they conform to the same pre- and post-conditions and class invariants.

In the following, we briefly outline three possible approaches to be used for implementing and supporting the idea of abstract version states: two of them rely on extending the existing approaches, the third is developed as a meta-object protocol. Our analysis of current trends in developing advanced object-oriented techniques shows that many researchers realise that it is important to be able to work with an abstract representation of the concrete object state. We will be discussing our proposals on a very general level; clearly, more research will have to be done to make them practical but we believe that they shed some light on the ways in which version recovery problems can be solved and demonstrate that further progress in this area of research is feasible.

5. LAYOM-based approach

Bosch [13] introduces the concept of the abstract object state (different from ours) to allow clients to access the object state in a disciplined manner, to make it possible to change the interface of objects dynamically, to build a concept of active object states upon it and to facilitate the modelling of complex dynamic behaviour of objects. This concept defines an abstraction of the object state which is placed at the object interface. The author points out that this state is a conceptualisation of the concrete object state and that, generally speaking, it is less complex than the latter in that it has fewer dimensions and smaller domains associated with dimensions. The layered object model (LAYOM) is developed to allow an extended description of objects.

The abstract object state can be described in the LAYOM together with several other additional components extending the conventional object model (layers, categories, active states, etc.). The abstract object state has several dimensions, called states, each of which is calculated as a (mapping) function on the concrete object state and is to be developed together with the object itself. The abstract state is declared in a class interface. Within this approach, the concrete state of an object is built out of abstract states of all objects declared inside it.

This approach allows us to deal with the abstract object state in a disciplined, unified and structured way. It is introduced formally and addresses important linguistic issues related to declaring abstract states, interfaces and classes. This research shows that it is natural and beneficial for an object to have an abstract state which conceptualises its concrete state.

Although the intentions for introducing this concept in the LAYOM clearly differ from ours, we can use many ideas discussed above and the general framework itself. To do this, we need to extend the model by introducing recovery functions. Besides, we need a much more general view on the concrete states of several versions: the abstract version state should capture what they all have in common. We would like to note that although the LAYOM itself is not intended for this, it is very general, and our "extension" can be viewed as one of its applications. The development of the abstract version state should obey different rules: it should follow the development of versions and be based on a sophisticated 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 this version programmers are to develop copy and recovery functions. In this approach the abstract state of version objects can be accessed only by the NVP manager from the basic class where they are all to be declared.
The proposed approach based on the LAYOM offers a structured way for introducing faulty version recovery into OO programming. It can be used as an extension of the LAYOM and can follow its development rules (with some adjustments). Although, clearly, this approach cannot help programmers to define the abstract version state common for 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; the latter, however, would contradict the idea of version design independence which is vital for achieving version failure independence [5].

6. PSL framework

The PSL [14] 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. The PSL specifications consist of logical and temporal rules relating situations to one another, each of which describes potential states with respect to the roles of components, role attributes, and issuance and reception of events. 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 operations; a role is an instance of an interface which can be instantiated in its turn. Attributes (a concept similar to Lamport's state function [11]) are used in this framework for describing abstract properties of several role instances. The framework is general, and attribute implementation is not part of it. Attributes are abstract functions of instances of roles and other value types. In the PSL they are declared as auxiliary functions within a protocol module.

The underlying idea behind this approach corresponds with the purposes of our research. We would like to "open" the implementation of version objects in a disciplined way because we need this openness for version recovery. The idea is to extend the interface of versions (without changing the interface of the diversely implemented object itself). By doing this, we can add new properties (related to version recoverability) to version objects.

It seems to be possible to extend the concept of attribute to describe the abstract version state and introduce this concept into the specification phase, although this will require some effort. 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.

7. Reflection

Another solution we would like to consider is based on using reflection (in the form of the meta-object protocol - MOP [15]). Meta-objects of all versions will contain an abstract object state (which has to be the same for all healthy versions). When a faulty version is detected, the abstract state of this version is corrected using a copy of the abstract state of the correct version kept in its meta-object. 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. Another thing which this type of reflection should be able to do is to update the abstract version state when the state of the base level object has been updated during the execution of a method. To do the latter,
one can, for example, use reflection upon all accesses/updates of the object states (e.g. assignments), which many MOPs allow to do. 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 FRIENDS system [4]). This requires a re-structuring of our approach presented in Section 2. In this case we will reflect upon all application calls of the diversely implemented object and intercept them. The meta-object of this object is the NVP manager encapsulating the adjudicator object (Figure 3). 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 (see Figure 4).

It is clear that reflection is 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 will be incorporated into the implementation of the ways reflection works. They are application-specific and can be designed only after the abstract version state (common to all version objects) has been designed.

There are many ways of improving the performance of this approach. For example, we may use lazy mapping functions (lazy reflection, so to say) which are called only when necessary (either for comparing the abstract states or for recovering the faulty version - in the latter case we need to execute the copy function for only one version).

Figure 3. System architecture and MOP implementing class diversity
8. Practical issues

The phases of developing faulty version recovery are shown in Figure 5. Different programmers are involved in these phases. Phase B is performed by N independent programmers (teams) using, if necessary, some ways of enforcing diversity (different languages, libraries, algorithms, etc.). In phase C another independent programmer analyses these designs and develops the abstract version state; s/he may have separate individual meetings with each version designer to assess the feasibility of the mapping function development. Afterwards, version programmers extend their objects by adding these functions in phase D. It can be said that in this phase they introduce the abstract version state into the specification of version objects.

Mapping functions can be quite complex, and, to achieve better error detection in their work, we can use a natural diversity which exists in the system: if the adjudicator has identified the majority of versions, then there are always several correct objects; this is why we can apply copy mapping functions for several of them (provided they have them). The results (the abstract states) must be the same, so a simple adjudication would allow us to choose the correct abstract state and to ignore the states which were calculated using copy mapping functions with faults (although their versions are proven to be correct).
9. Discussion

First we would like to discuss the problem of re-use. The general NVP framework for introducing diversity [8] relies on using conventional inheritance. A set of general rules should be followed here. If the interface of the diversely implemented object has been modified (e.g. extended), then all version objects should be modified to conform to this new interface. Version developers should re-use the old code here. The situation is more complex with respect to the means of version recovery. Generally speaking, changes in the interface can cause changes in representing the states of some, or all, of the versions. The first approach here would be to try and re-implement mapping functions only, without changing the abstract version state (phase D in Figure 5). But if not all version programmers have found this feasible, the re-implementation should start from phase C.

We would like to emphasise that using the same abstract state for all versions does not cause failure mode dependency. This is due to the fact that the application methods of version objects are to be designed independently, without knowing about the abstract version state.

In this paper we have formulated the problem of version object recovery in OO N-version programming. We have formalised the concept of faulty version recovery using the abstract version state concept, proposed several system approaches for implementing this sort of recovery in OO systems and outlined a methodology of developing recoverable object versions. Our implementation schemes take advantage of some existing OO techniques (the LAYOM, PSL, reflection) and rely on clear system structuring and separation of the application-related problems from the problems related to object version recovery. In the future we will concentrate on further elaboration of our approach, in particular, on detailed development of the proposed implementation techniques and of the methodology outlined.

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

References.


