Mutation-Like Oriented Diversity for Dependability Improvement: A Distributed System Case Study

Conference Paper *in* Lecture Notes in Computer Science · October 2004

DOI: 10.1007/978-3-540-30182-0_92 · Source: DBLP

Avelino F. Zorzo
Pontifícia Universidade Católica do Rio Grande do Sul
138 PUBLICATIONS **1,212** CITATIONS
SEE PROFILE

Eduardo Bezerra
Federal University of Santa Catarina
111 PUBLICATIONS **464** CITATIONS
SEE PROFILE

Flávio M De Oliveira
43 PUBLICATIONS **228** CITATIONS
SEE PROFILE

Mutation-like Oriented Diversity for Dependability Improvement: A Distributed System Case Study[⋆]

D. O. Bortolas, A. F. Zorzo [⋆⋆], E. A. Bezerra, and F. M. de Oliveira

Hewlett-Packard/PUCRS - Research Centre on Software Testing (CPTS)
and Research Centre on Embedded Systems (CPSE)
Faculdade de Informática (FACIN/PPGCC)
Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681 – Prédio 30, 90619-900, Porto Alegre, RS, Brazil
{dbortolas,zorzo,eduardob,flavio}@inf.pucrs.br

Abstract. Achieving higher levels of dependability is a goal in any software project, therefore strategies for software reliability improvement are very attractive. This work introduces a new technique for reliability and maintainability improvement in object-oriented systems. The technique uses code mutation to generate diverse versions of a set of classes, and fault tolerance approaches to glue the versions together. The main advantages of the technique are the increase of reliability, and the proposed scheme for automatic generation of diverse classes. The technique is applied to a distributed application which uses CORBA and RMI. First results show promising conclusions.

1 Introduction

The use of computer systems is growing fast and becoming more pervasive in every day life. This is frequently called ubiquitous computing [1]. As a consequence, computers are each time more needed and it has become difficult to picture the modern world without them. Computers are so widespread that they are used even in places where responsibility is huge and failures may result in a tragedy, e.g. airplanes, trains or air traffic control systems. Computer high reliability is essential in these type of systems, and fault tolerance is a strategy used to minimize the effects of possible faults occurring in such systems.

Redundancy is the main mechanism employed in the construction of fault-tolerant systems at both, hardware and software levels. Replication of a hardware component ensures that when a failure happens, another component replaces the defective one [2].

The main problem in hardware components are the physical failures, e.g. broken wires or short cuts. Therefore, a failure in a hardware component does not imply that another "copy" of the same component will produce the same

[⋆] This work was developed in collaboration with HP Brazil R&D.
[⋆⋆] Partially supported by CNPq Brazil (grant 350.277/2000-1).

failure. This same strategy cannot be applied directly to software components. If a software component is replicated, its faults (*bugs*) will also be replicated. This is true as faults in software components are classified as *design faults* [2], and each time a problem happens, the replaced component will have the same behaviour. Although hardware faults could be also the result of design mistakes, this is not common and therefore will not be considered in this work.

Design diversity is an approach used to overcome the replication problem in software [3]. Components conceived diversely have the same interface and functionality, but are designed and implemented using different techniques by separate groups of programmers, working independently. It is expected that groups working independently do not make the same mistakes. Thus, redundancy of software components might not present the same faults under the same environmental conditions.

Although design diversity has clear benefits to software dependability improvement, they do not come cheap. The high costs of this technique are mainly due to the need of several development teams working simultaneously. Therefore, this research is directed towards the automation of some steps of diversity programming, in order to minimize its cost.

The main objective of this work is dependability improvement of object-oriented systems using automatic code generation mechanisms. Dependability of a system is the ability to deliver services that can be trusted [4]. The concept includes several attributes [4], but this work targets mainly "reliability" and "maintainability". It will be shown also that automatic generation can help to ease the development and maintainability activities in software fault-tolerant systems, and can even decrease the total cost of these systems. Automatic generation can be used to reduce the amount of faults introduced during the implementation of diverse versions of a software.

In the proposed approach, traditional software fault tolerance mechanisms as, for instance, N-version programming (NVP) [5] and recovery block [6], are used to improve the reliability of software systems. The innovative aspect introduced in this paper is not the use of these fault tolerance mechanisms, but the whole process for automatic generation of new diverse classes of a system.

This approach is called Mutation-like Oriented Diversity (MOD). It uses a controlled sort of mutation, and not the one existing in nature and employed as a model in the traditional Mutant Analysis testing technique [7]. The approach's central idea is the modification of some specific parts of the source code of an application, targeting the generation of new diverse versions. Existing components are used as basic building blocks, and the generation of diverse versions is performed by applying *ad-hoc* mutant operators to the original code. The whole process was conceived to ease the automation process, and a tool aiming this objective is under development.

The remaining of this paper is organized as follows. Section 2 presents some related work. Section 3 describes the proposed approach. Section 4 discusses a distributed system used as a case study, presenting some interesting results. Section 5 presents the conclusions and future work.

2 Related Work

Mutation Analysis [7] is a testing technique that has as goal the evaluation of a set of test cases selected to a particular application. The technique uses a large number of modified programs, called mutants, that may be automatically generated by performing small modifications in the original code, producing new syntactically correct programs. The modifications are performed by mutant operators [8], which are specifically designed to certain constructions of a language and, consequently, each language has its own set of mutant operators. These operators have as objective to produce a wrong behaviour in resultant mutants, which are used to distinguish good from bad test cases. In this way, the set of test cases can be reduced and, as a consequence, the testing time is also shortened. However, this is not always true, as it is the case of the equivalent mutants [9].

MOD has some similarities to Demillo's Mutation Analysis, as the objective is to automatically create new diverse versions of some classes of an application, and these versions are generated by a set of mutant operators. Differently, however in MOD the new versions are produced to have a similar behaviour as the original part of the code from which they were generated. In MOD, mutant operators perform replacements in statements of the original application source code, by statements that are intended to have the same functionality as the original one.

Software fault tolerance applied to object-oriented systems is not a new research topic in Computer Science. For instance, [10] states that software fault tolerance cannot be achieved by just implementing traditional fault tolerance schemes in object-oriented systems. They propose a new system structure to deal with problems that arise from the fault tolerance applied to object-oriented software. Their framework to develop fault-tolerant software is based on diversely designed components that provide two types of redundancy: masking redundancy and dynamic redundancy. [11] proposes a new N-version programming strategy in which the diversity of the components is applied at the class level. The classes are developed separately and independently, and are encapsulated into a diversely designed object. This new NVP approach uses the general framework proposed in [10].

MOD tries to tolerate faults in a similar way to those approaches. It provides support for masking and dynamic redundancy, and it is also applied to the level of classes. The main difference in MOD is that the diverse classes are generated automatically. This feature reduces the complexity of incorporating new designed components to the system during its lifetime.

3 Mutation-like Oriented Diversity (MOD)

The proposed approach has as a main objective to significantly enhance the reliability and maintainability figures of object-oriented (OO) systems. Figure 1 shows the design of a non-fault-tolerant OO software, which is divided in layers to better describe some particularities of the strategy. The *component classes* *layer* is where the basic building classes, or basic building components, of the system are located. The *intermediate classes layer* is composed of classes that will be diversely replicated. Finally, the *user classes layer* contains the remaining classes of the system. The starting point to understand MOD is to comprehend that it needs to be applied to an initial piece of software. This means that it is necessary to have, at least, the components shown in all of the three layers of Figure 1. The process is then carried out in two stages. First the mutants are generated, and next an extra layer, the *controller classes layer*, is added to the system, in order to attach the newly created mutant classes.

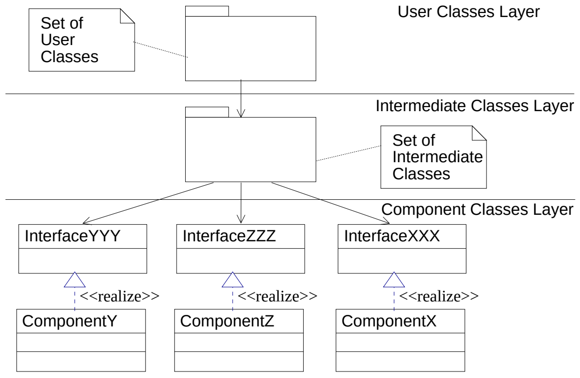

Fig. 1. Software Design.

3.1 Mutant Generation

In this phase the set of classes in the intermediate layer of the original software are replicated diversely, generating the intermediate layer of the new fault-tolerant system. Figure 2 shows the class diagram of a fault-tolerant system generated using MOD. The new *controller classes layer* shown in Figure 2 is discussed in the next section.

The generated classes in the intermediate layer are called *mutants*. They represent the new diverse versions obtained from the mutation of the intermediate classes layer. In the mutation process, *mutant operators* are used to replace constructions and instructions in the source code.

Changes in single instructions as, for instance, arithmetic or relational expressions, are not sufficient to introduce diversity to the newly generated mutated software versions. For this reason, a coarse grain mutation should be performed. In MOD, the mutation takes place at the class level, which means that the mutated instruction is an entire class. These classes are represented by elementary components in the component classes layer (Figure 1).

The elementary components in the component layer are essential to the approach. They must implement the interfaces used by the intermediate layer classes, and they must be developed according to the design diversity methodology. In this work it is assumed that at the time of applying the proposed technique the diverse versions of components to be used are already available. An important point is that the source code of the components is not required, which means that commercial-off-the-shelf (COTS) components can be used.

An interesting point to be noticed is that since system diversity is provided by elementary components, the more components used the better the diversity of the new generated classes. Another point is that, since the process of generating mutants is automatic and therefore cheaper, new elementary components can be easily incorporated to the system during its lifetime, thus opening the possibility for new reliability improvements.

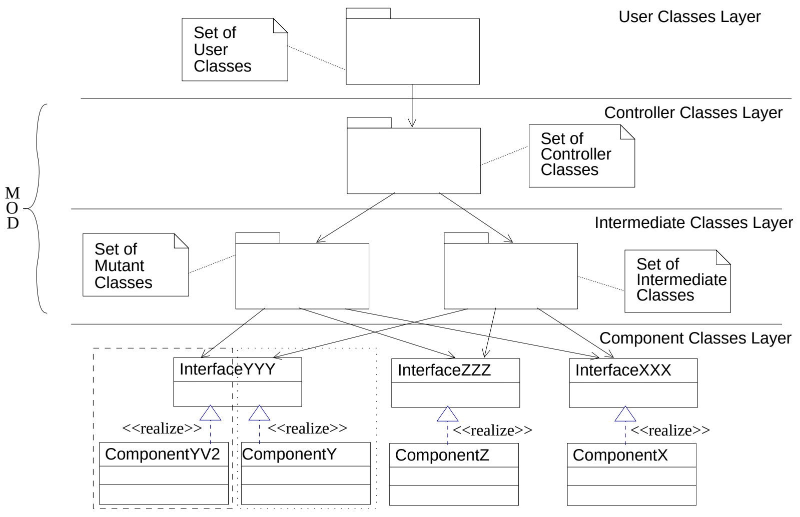

Fig. 2. Mutant Generation.

3.2 Controller Construction

Having the new intermediate classes layer, the next step is to join the several mutants and to use them in a software fault tolerance mechanism. The controller layer implements the software fault tolerance mechanism used to join the generated mutants. Any technique based on diversity can be used as, for instance, N-version programming or recovery block.

The controller layer must have exactly the same interface as the intermediate layer in the original system, since the user layer is not modified at all. The process can be better structured using, for example, reflection as in [12]. This will certainly reduce the modification that the approach has to perform in the original system. This may also improve the final result as it will allow the user interface to be any class that makes access to the intermediate classes. This alternative is under investigation and is the subject of a future work.

4 Using MOD in the Design of a Distributed Application

The selected case study to explain MOD's usage, and also for its initial evaluation, is a Reverse Polish Notation (RPN) calculator for a distributed environment. The motivations for choosing this case study are straightforward. First, the model of an RPN calculator matches perfectly with MOD. Second, an RPN calculator is not difficult to implement, as the basic building block is a stack. Finally, it has some facilities to test and make measurements from resulting data.

Although the case study is not complex, it can be seen as an ambitious implementation. Two types of distributed objects are used, RMI and CORBA. Specific mutant operators have been created in order to accomplish with the task of translating a user class of RMI objects into a user class of CORBA objects.

Considering the architecture shown in Figure 1, the original version of the case study has three main classes: the *MainProgram* class in the user layer; the *Calculator* class in the intermediate layer; and the *StackImpl* class, which is a component in the component layer.

The case study is implemented in Java and an obvious choice to distribute the application is the use of Java RMI. Thus, the *StackImpl* class is an implementation of an RMI object and the *Calculator* class is a user of this distributed RMI object. Having the first version of the calculator program, the next step is to get hold of additional diverse components (Stack programs) that implement the *Stack* interface. After that, the program mutation in the Calculator class is conducted aiming the generation of diverse Calculator classes. This mutation is performed based on the new elementary components. Finally, the controller classes (controller layer) are created.

4.1 Diverse Components (Component Layer)

Diverse components are required in the component layer in order to introduce diversity to the intermediate layer. As discussed in Section 3.1, COTS could be used in this layer. However, for this case study, a faster option was an in-house implementation of extra versions for the Stack interface. In addition to the original RMI *StackImpl* class, five extra components have been built: two more RMI classes and three CORBA classes. A total of six diverse components are used in this case study.

A stack implementation is relatively simple and it is not likely to present a faulty behaviour. As it might be difficult to observe reliability improvements, it has been decided to inject faults in the source code of each implemented stack. The fault injection was designed aiming a failure rate of 2% in all method calls of the different stacks. A possible observed failure in a pop call is the situation in which instead of the correct value, a new integer is randomly generated and returned to the caller of the method. Another situation is when instead of the value, an exception is returned to the caller of the method. A problem in a push call could be when pushing the correct value, another number is randomly generated and pushed back into the stack.

4.2 Mutation Approach (Intermediate Layer)

The mutation happens in the classes belonging to the intermediate Layer, in order to generate new diverse replicated classes. Two mutant operators are very common and may take part in almost any system that uses MOD. They are the *Instantiation Operator* (*InOp*) and the *Import Operator* (*ImOp*).

In general, the *InOp* operator creates a new mutant by replacing component instantiations in intermediate layer classes, by new diverse component instantiations. For example, if the Java code in the original program is "*Stack st = new StackImplA();*" the *InOp* operator will change this line into "*Stack st = new StackImplB();*".

The *ImOp* operator is used to replace or to insert Java *import* instructions in intermediate layer classes. Each time a new package is requested by the application, an *import* instruction is inserted in the source code. When a package is not any longer needed, the mutant operator removes the respective *import* instruction from the code. This operator is usually applied right after an *InOp*. For example, an intermediate class that uses a package called *StackA* has the instruction "*import br.pucrs.cpts.StackA.*;*". As a result of a mutation performed by an *InOp* operator, a new package *StackB* is needed, and the previous package has to be removed. The *ImOp* operator will make the change, replacing the original import instruction by "*import br.pucrs.cpts.StackB.*;*".

In the distributed application, in this paper, the classes belonging to the intermediate layer do not create instances of objects, instead, they obtain instances of distributed objects from some sort of service directory. As a result, the *InOp* operator could not be used straight away and it had to be split into two operators, the RMI to RMI operator (*R2ROp*) and the RMI to CORBA operator (*R2COp*). *R2ROp* reads an intermediate layer class having RMI object references and creates a mutant intermediate layer class having other RMI object references. Basically, *R2ROp* performs the replacement of the URL of the original intermediate layer class by the URL of the new mutant class.

The *R2COp* operator reads an RMI intermediate layer class and produces a CORBA intermediate layer class. The translation of an RMI object reference into a CORBA object reference is not direct. In this case, the *R2COp* operator translates an RMI instruction into a set of CORBA instructions, in order to modify the calls to get the object references. The translation is not straightforward, but it is feasible to be accomplished. Another aspect to be observed is that an RMI object reference has a different type when comparing to a CORBA object reference. The original *InOp* operator is used here to solve the differences.

For the case study, the mutation process generates a total of six versions of the intermediate layer classes. These new classes are obtained after having applied the necessary mutant operators to the original intermediate layer classes of the distributed RPN calculator program.

4.3 Adding Fault Tolerance Mechanisms (Controller Layer)

The new diverse versions generated in the previous section are used by the controller layer to introduce fault tolerance capabilities to the system. The controller layer is created using the N-version programming model. This fault tolerance technique was chosen as a consequence of the case study distributed features. The controller layer classes are built in a way that they offer the same names for the user layer, and therefore the classes do not need to be modified.

There are some options to automate the generation process of the controller layer classes as, for instance, templates for some pre-defined fault tolerance mechanisms. However, in the present version this process is not completely automated, and the controller construction is the module of MOD that needs the larger amount of manual intervention.

4.4 Preliminary Results

First, in order to observe the behaviour of the original program, a total of 10^4 test sets were created to feed the distributed calculator. Each test set is an expression having 2, 3, 4, 5, 6 or 7 operands (expression "$\underline{3}\ \underline{4}+\underline{5}+$", for instance, has three operands, which are underlined). The result of this experiment is represented by the dotted line in Figure 3.

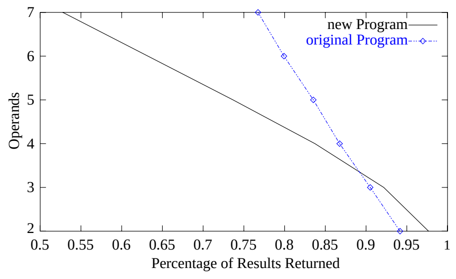

Fig. 3. All returned results vs. number of operands in an expression.

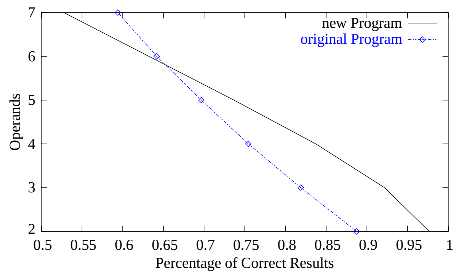

Fig. 4. Correct results vs. number of operands in an expression.

Considering the selected expressions, one could expect the stack to present a correct behaviour in 98% of the method calls. However, this figure is never observed when running the experiment, as it is necessary more than one method call to perform a calculation. It is observed also that as the number of method calls increases, so does the chance of getting erroneous output. As shown in Figure 3, the percentage of responses, both correct and incorrect, in the original program (dotted line), decreases as the number of operands increases (the number of calculations and, therefore, the method calls increase).

The dotted line in Figure 3 represents all the results that do not raise exceptions during the evaluation of an expression. However, there is yet a chance that the returned result is erroneous. For this reason a gold calculator was designed and used to verify the returned results from the original calculator program. The dotted line in Figure 4 shows the percentage of only the correct results returned from the execution of the original calculator program, employing the same 10^4 test cases executed previously.

The same set of test cases was applied to the new program generated by MOD. Observing the solid lines in Figures 3 and 4, it is possible to conclude that the number of all returned results and the number of only correct outputs are identical.

The diagram shown in Figure 3, is the most significative one as it represents all the responses provided to the user, both the correct and the incorrect ones. Comparing the original program to the new generated one, it is possible to see that for the majority of the test cases (4, 5, 6 and 7 operands), the original program returns more results than the generated one. In the worst case, the 7 operands test cases, the new generated program could only return a result in a few more than 50% of all calculated expressions.

However, it is necessary to know if the returned results (Figure 3) can be trusted. When they are compared against the correct ones (Figure 4) it is possible to observe that the percentage of results in the original program vary too much and, therefore, they are not as reliable as the returned results of the new generated one.

5 Conclusions and Future Work

This work introduces a new strategy for developing systems having high reliability requirements. First results are still incipient, but the main idea could be verified through a case study. Several enhancements are under investigation and they will be discussed in a future work.

The results of applying the proposed methodology appear to be as reliable as traditional fault tolerance software approaches. However, the advantages of applying the methodology are not only related to the level of reliability reached, but also to the ease of maintainability of code and the level of automation of the strategy. The intermediate classes of the program are replicated through the mutation mechanism, in other words, the logic of the replicated code is the same, and no extra maintaining is needed for this code. The automation is an important feature, as it can accelerate the development phase and even decrease the chance of introducing hand made modification faults.

The approach also has some disadvantages. As the methodology uses traditional fault tolerance software strategies, the drawbacks are similar. For example, the problems related to the use of fault-tolerant software in object-oriented languages [10], or even the restrictions imposed by the nature of the application [2]. Another drawback concerns shared resources. As mutant versions are generated from a single set of classes, in case several mutants have access to the same resource (e.g. file, I/O), it may result in a system failure. Other problems are mainly related to the design of the software, which must follow strict rules (see Section 3). For instance, the system shall be designed aiming high cohesion and low coupling. The mutant generation task may become difficult or even impossible to be accomplished in highly coupled systems.

Finally, there is a large amount of future work to be done. It is necessary to implement tools to support the automatic generation of the mutant mechanism, and also to design/implement new case studies. Another research issue under investigation is the possibility of increasing the reliability of a system by combining parts of different components to build a new class.

References

1. Weiser, M.: Ubiquitous Computing. IEEE Computer - Hot Topics. **26** (1993) 71–72.
2. Jalote, P.: Fault Tolerance in Distributed Systems. Prentice Hall. (1994) 432p.
3. Litlewood, B., Popov, P., Strigini, L.: Modeling Software Design Diversity - A review. ACM Computing Surveys. **33** (2001) 177–208.
4. Laprie, J. C.: Dependable Computing and Fault Tolerance: Concepts and Terminology. In Digest of FTCS-15. (1985) 2–11.
5. Avizienis, A.: The N-version Approach to Fault-tolerant Software. IEEE Transactions on Software Engineering. **11** (1985) 1491–1501.
6. Randell, B.: System Structure for Software Fault Tolerance. Proceedings of the International Conference on Reliable software. (1975) 437–449.
7. Demillo, R. A., Lipton, R. J., Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. Computer. **11** (1978) 34–41.
8. Offutt, A. J., Lee, A., Rothermel, G., Untch, R., Zapf, C.: An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology. **5** (1996) 99–118.
9. Offutt, A. J., Pan, J.: Automatically Detecting Equivalent Mutants and Infeasible Paths. The Journal of Software Testing, Verification, and Reliability. **7** (1997) 165–192.
10. Xu, J., Randell, B., Rubira-Casavara, C.M.F., Stroud, R.J.: Toward an Object-oriented Approach to Software Fault Tolerance. in Recent Advances in Fault-Tolerant Parallel and Distributed Systems (eds. D.K. Pradhan and D.R. Avresky) IEEE Computer Society Press. (1995) 226–233.
11. Romanovsky, A.: Diversely Designed Classes for Use by Multiple Tasks. ACM SIGAda Ada Letters. **20** (2000) 25–37.
12. Xu, J., Randell, B., Zorzo, A. F.: Implementing Software Fault Tolerance in C++ and Openc++: An Object-Oriented and Reflective Approach. International Workshop on Computer Aided Design, Test and Evaluation for Dependability (CADTED). (1996) 224–229.