A Taxonomy of Faults for UML Designs

Trung Dinh-Trong, Sudipto Ghosh, Robert France[1]*
{*trungdt,ghosh,france*}*@cs.colostate.edu*
Benoit Baudry, Franck Fleury[2]
{*bbaudry, ffleurey*}*@irisa.fr*

[1] Computer Science Dept., Colorado State University, Fort Collins, CO 80523
[2] IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. As researchers and practitioners start adopting model-based software development techniques, the need to rigorously evaluate design models is becoming apparent. Evaluation techniques typically use design metrics or verification and validation approaches that target specific types of faults in the models. Fault models and taxonomies may be used to develop design techniques that reduce the occurrence of such faults as well as techniques that can detect these faults. Fault models can also be used to evaluate the effectiveness of verification and validation approaches. In this paper we present a taxonomy of faults that occur in UML designs. We also describe a set of mutation operators for UML class diagrams.

Keywords: *design quality, fault model, fault taxonomy, validation, verification, UML*

1 Introduction

Model-based software development techniques raise the level of abstraction at which developers conceive and implement complex software systems. To ensure that the developed systems are of high quality, it is important for developers to evaluate the quality of the system models. Quality may be evaluated in several ways, including (1) techniques that use design metrics and the relationships between design metrics and the fault proneness of software, (2) formal verification techniques, and (3) testing techniques.

A systematic study of faults that occur in UML designs will help us develop better design evaluation techniques. While there is a large body of literature on fault models for system implementations (code), there is a lack of research on fault models in designs. In many cases, researchers reverse engineer the code to obtain views of the design. Our goal is to develop fault models for UML designs at various levels of abstraction, not just models of the code. There will be some overlap in the fault models for designs and code. However, there is a completely

* This research was supported in part by National Science Foundation Award #CCR-0203285 and an Eclipse Innovation Grant from IBM.

new set of fault types that arise from the use of different UML diagram types that can be used to represent different views of a system.

There are many approaches to building fault models. Researchers may use reports on faults that are detected during testing or reported after deployment. An alternative approach is to define mutation operators for the language used to specify models. The theory of mutation analysis is based on the study of faults that are created by syntactic changes made in a program. The design of mutation operators is based on two key hypotheses: (1) *competent programmer hypothesis*, which states that programmers generally write programs that are close to the correct program, and (2) *coupling effect*, which states that if a test set can kill first-order mutants, they will also kill higher-order mutants.

Our approach to building a fault model for UML designs is based in part on studies of design models developed by students in our courses, and mutation analysis of UML designs. We plan to perform a study of design models developed in the industry so that we get a better idea of faults made by experienced developers, not just novice students.

In this paper, we first present a taxonomy of faults that can occur in UML designs. We then describe faults that are based on definitions of mutation operators for UML class diagrams.

2 Fault Taxonomy

Our high level categorization of quality problems (faults) in UML designs is as follows:

1. **Design metrics related:** Examples of design metrics include cohesion, coupling, DIT, LCOM, and WMC [1]. Having undesirable values for these metrics does not necessarily imply that the system has faults; the design could still be functionally correct. However, problems in understandability, testing, maintenance, and evolution may result. There are several studies that have looked into the relationships between design metrics and the fault-proneness of software modules (e.g., see [2]). We do not investigate this category any further in this paper.

2. **Faults that are detectable without execution:** Similar to compile-time faults in programs, these faults are usually based on the UML syntax defined by the UML metamodel, and are reported (even prevented) by most UML editing tools. These faults can be of two types:
   (a) *Fault in a single view:* Such faults are the result of using incorrect syntax. Examples include not providing a class name for a class, and using duplicate names in a single namespace.
   (b) *Consistency fault between two or more view types:* Such faults arise as a result of inconsistent information across different views. For example, using an operation in a sequence diagram that is not defined in the class diagram.

3. **Faults related to behavior:** Similar to run-time faults in programs, these faults are more subtle and can arise as a result of incorrect specification of (1) behavior in UML behavioral models, (2) class structure or (3) pre- and post-conditions and invariants in a class diagram.

3 Mutation Faults

We list below the mutation operators that we have identified for UML class diagrams. We do not claim that the set of operators is complete. Some of these operators may result in a design model that is determined to be faulty without requiring execution (a *category 2 fault*). Others result in more subtle behavioral faults.

Mutating classes: The attributes *isAbstract* and *Visibility* can be mutated. Making a class abstract when it is not results in a category 2 fault if the class is instantiated; static analysis will detect that the system tries to instantiate an abstract class. Making an abstract class concrete results in a design fault that is in category 2 in some cases; if the model is translated to Java code, the Java compiler will detect an anomaly.

Mutating class variables: The visibility and multiplicity of the variables can be changed. This may result in inconsistent views. For example, if a variable that was public and was being accessed in a sequence diagram is now made private, the model becomes inconsistent. Such an inconsistency results in a category 2 fault.

Mutating operations: The visibility of an operation may be changed. Making a public operation private may lead to an inconsistent view if that operation was being used in a sequence diagram. Making a private operation public may preserve the original behavior, but also allow new undesirable behavior (e.g., malicious or unauthorized access).

A redefinition of an operation in a child class can be deleted. This will result in faulty behavior because the operation in the superclass will be executed instead of the redefined one.

Another mutation consists in swapping compatible parameters in the definition of an operation. This fault cannot be detected without execution and it results in a faulty behavior when the method is called. This is equivalent to calling the operation by swapping the actual parameters.

Mutation of operations corresponds to category 3 faults.

Mutating associations: Associations have several attributes: *isOrdered*, *isUnique*, *isReadOnly*, *frozen*, *multiplicities*, *defaultValue*, *navigability*, and *aggregation kind*. The values of these attributes can be changed to mutate the association. For example, we can make an association readonly. In the case where there is behavior that requires writing, a test will be able to kill the mutant. A composition can be mutated to an aggregation. This has implications on the lifecycle of the object and the contained objects.

An association end multiplicity may be changed, for example, "1..*" can be converted to "*". Existing behavioral specifications that resulted in one or more links to objects will still be consistent with the "*" specification, resulting in an equivalent mutant.

In general, if the operator results in a weaker constraint, the mutant may be equivalent. This happens when we change a private operation into a public one, or relax the multiplicity constraints. However, even if a weaker constraint generates an equivalent mutant, the fault is still an important one in the model. For example, a designer may want to make an operation private due to a security issue. A fault that makes such an operation public will result in a model that has equivalent behavior, but is less secure. However, such a design fault cannot be detected by testing the behavior of the model because it has an equivalent behavior. Other methods, such as static analysis, are needed to detect such faults in a model.

Swap compatible role names: This can be done in two ways. Role names associated with the same class may be swapped as shown in Figure 1(a). Alternatively, roles associated with a child class can be swapped as shown in Figure 1(b).

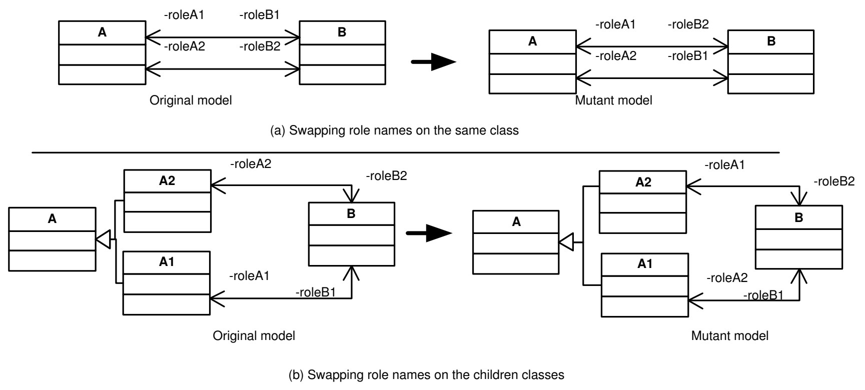

Fig. 1. Swapping Compatible Role Names

This will have an effect when the associations are actually navigated via their roles. Two different associations outgoing from a single class usually represent different concepts. Navigating one instead of the other should not lead to the same behavior and it is desirable that such faults are detected during testing. A test case that calls an operation through the incorrect role should fail.

Wrong Reference to Parent Class (WRP): An example application of this operator is described in Figure 2. This operator replaces an association by an

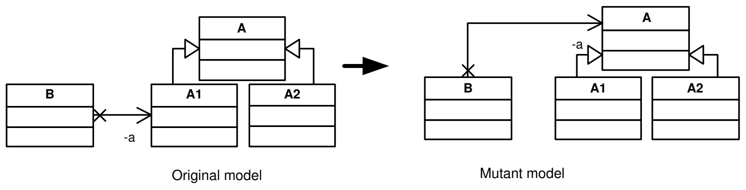

Fig. 2. Wrong Reference to Parent Class (WRP operator)

association to a parent class. The resulting model will have a weaker constraint than the original one. Hence, the mutant may be equivalent. Even though it is not detectable by testing, this operator is still interesting as it corresponds to an actual fault that can occur when a system is modeled.

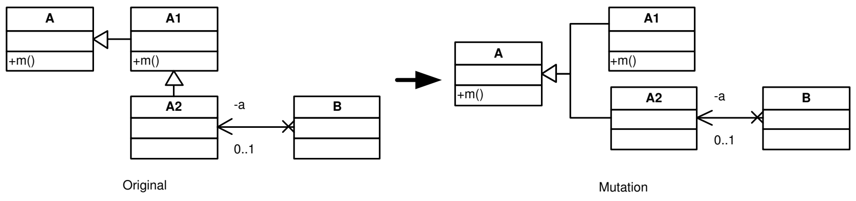

Fig. 3. Wrong Inheritance Tree (WIT operator)

Wrong Inheritance Tree (WIT): An example application of this operator is described in Figure 3. The operator builds a broad inheritance tree instead of a deep inheritance tree. This operator results in a category 2 fault (for example if B calls a method inherited from A1 that is only defined in A1) or in a behavioral fault (if B calls a method inherited from A that is redefined in A1).

Some of the mutation operators we defined in this paper have equivalent definitions at the program level. Ma et al. [3] define a set of mutation operators for classes, some of which are specific to Java while others apply to object-oriented languages in general. Three of these operators correspond to the operators that we have defined at the model level.

- The WRP operator is equivalent to the PMD operator that changes the declaration of a variable to its parent class type.
- The operator that changes the visibility of class variables is equivalent to the AMC operator that changes the access level for instance variables.
- The operator that deletes a redefinition is equivalent to the IOD operator that deletes an overriding a method in an OO program.

4 Conclusions and Future Work

We presented a taxonomy of faults for UML designs and described the mutation operators for UML class diagrams. Although the set of operators may not be complete, it is still significant as it is the first set of operators defined for UML models that we are aware of. We need to further explore actual faults created by both novice and experienced developers. Some of the mutation operators we presented result in mutants that are trivially detected (killed) by a UML editing tool, while some others create equivalent mutants. We need to further investigate these mutation operators for their utility in evaluating testing approaches.

The fault model is being used to evaluate the UML model testing approach [4] being developed at Colorado State University and the model transformation testing approach [5] being developed at IRISA.

References

1. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20** (1994) 476–493
2. Briand, L.C., Melo, W.L., Wüst, J.: Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans. Software Eng. **28** (2002) 706–720
3. Ma, Y.S., Kwon, Y.R., Offutt, J.: Inter-class mutation operators. In: ISSRE'02 (Int. Symposium on Software Reliability Engineering), Annapolis, MD, USA, IEEE Computer Society Press, Los Alamitos, CA, USA (2002) 352 – 363
4. Andrews, A., France, R., Ghosh, S., Craig, G.: Test Adequacy Criteria for UML Design Models. Journal of Software Testing, Verification and Reliability **13** (2003) 95–127
5. Fleurey, F., Steel, J., Baudry, B.: MDE and Validation: Testing Model Transformation. In: Proceedings of the SIVOES-Modeva workshop, SIVOES (Specification Implementation and Validation Of Embedded Systems)-MoDeVa (Model Design and Validation), Rennes (2004)