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, it becomes even more important to rigorously evalute the design quality of models. Evaluation techniques typically use design metrics or verification and validation approaches that target certain specific types of faults in the models. Fault models and taxonomies may be used to develop design techniques that reduce the occurence 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, such as (1) by using 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 definitely be some overlap in the fault models for designs and code. However, there is

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

a completely 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. 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 would also like 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 in UML designs is as follows:

1. **Design metrics related:** Examples of design metrics include cohesion, coupling, DIT, LCOM, and WMC [**?**]. Having undesirable values for these metrics does not necessarily imply that the system has faults; the design is probably still correct and the system will function correctly. 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 [**?**]. In this paper, we do not investigate this category any further.

2. **Faults that are detectable without execution:** Similar to the notion of compile-time faults in programs, such faults are usually based on the UML syntax defined by the UML metamodel, and are reported (even prevented) by the UML drawing tool itself. 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 are related to the consistency of information present in different views. For example, using an operation in a sequence diagram that is not defined in the class diagram. The degree to which such faults are detected by a tool depends on its sophistication.

3. **Faults related to behavior:** Similar to the notion of run-time faults in programs, these faults are more subtle and relate to the incorrect specification of

behavior in the sequence (or activity diagrams), the structure specified in a class diagram, or pre- and post-conditions and invariants in a class diagram. Such faults may be the result of incorrect specifications, or incorrect understanding of the problem and the solution. These faults can also result from unintended changes during design evolution.

3 Mutation Faults

We list below the mutation operators that we have identified for UML class diagrams. Some of these operators may result in a design model that is determined to be faulty without requiring execution (compile-time error). Others result in more subtle behavioral faults. We point out all the cases below.

Mutating classes: The attributes *isAbstract* and *Visibility* can be mutated. Making a class abstract when it is not results in an equivalent mutant. Making an abstract class concrete results in ???

Mutating class variables: The visibility 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.

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 behavior (e.g., in terms of malicious or unauthorized access).

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 to read only. In case there is behavior that required 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.

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

SG: Discuss the effect here.

Wrong Reference to Parent Class (WRP): SG: Describe the operator and discuss the effect here.

Fig. 1. Swapping Compatible Role Names.

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

Wrong Inheritance Tree (WIT): SG: Describe the operator and discuss the effect here.

4 Conclusions and Future Work

We presented a taxonomy of faults for UML designs and described the mutation operators for UML class diagrams. We do not claim that the set of operators is complete. 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 drawing 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 [1] being developed at Colorado State University and the model transformation testing approach [**?**] being developed at IRISA.

References

1. 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