A Controlled Experimental Evaluation of Test Cases Generated from UML Diagrams

Aynur Abdurazik and Jeff Offutt
Dept of Info and Software Engr
George Mason University
Fairfax, VA 22030-4444
USA
{aynur,ofut}@ise.gmu.edu

Andrea Baldini
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24, I-10129, Torino
Italy
baldini@polito.it

Abstract

This paper presents a single project experiment on the fault revealing capabilities of test sets that are generated from UML statecharts and sequence diagrams. The results of this experiment show that the statechart test sets do better at revealing unit level faults than the sequence diagram test sets, and the sequence diagram test sets do better at revealing integration level faults than the statechart test sets. The experimental data also show that the statecharts result in more test cases than the sequence diagrams. The experiment showed that UML diagrams can be used in generating test data systematically, and different UML diagrams can play different roles in testing.

1. Introduction

There are different ways to develop software. Each methodology defines its own software life cycle concepts and uses different notations. All software development life cycles include five main activities: specification, design, implementation, testing, and maintenance. Testing activities are often considered to be the most expensive [4], and are crucial for establishing confidence in the software.

A program *unit* is a procedure, function, or method. A *module* is a collection of related units, for example, a C file, an Ada package, or a Java class. *Unit and module testing* (or just unit testing) is the testing of program units and modules independently from the rest of the software. *Integration testing* refers to testing interfaces between units and modules to assure that they have consistent assumptions and communicate correctly [4]. This is in contrast to *system testing* where the objective is test the entire integrated system as a whole. Because of the emphasis on testing interfaces, integration testing is usually a white box testing activity that requires the availability of source code. In contrast, system testing usually assumes the absence of source code, and is thus usually black box.

There are also many approaches to generate test cases. A *test* or *test case* is a general software artifact that includes test case input values, expected outputs for the test case, and any inputs that are necessary to put the software system into the state that is appropriate for the test input values. Test cases are usually derived from software artifacts such as specifications, design or the implementations. One significance of producing tests from specifications and design is that the tests can be created earlier in the development process, and be ready for execution before the program is finished. Additionally, when the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications and design, allowing the specifications and design to be improved before the program is written.

Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of test requirements that are satisfied. There are various ways to classify adequacy criteria. However, specification-based testing criteria have been focused on formal specifications and based on certain specific *parts* of the specifications. Large, complex software systems are specified in many parts that describe different aspects of the software. This includes such aspects as the overall functional description of the software, state-based behavior models of the software, and connections between software components. Researchers have used this to generate separate types of tests that are drawn from specific specification descriptions, and that are intended to target different types of faults.

However, the question of whether this piece-meal approach to testing will find the different types of faults has not been empirically validated. This paper carried out an experimentation on this testing assumption. Specifically, we ask the dual question of whether tests that are designed from

one type of specification description and target one type of faults will find the target faults, and whether the same tests can find other types of faults. In our study, two different specification/design sources are used to generate tests, and their fault revealing capabilities at two testing levels are compared. The specification/design language for this experimentation is the Unified Modeling Language, the parts used are the statecharts and sequence diagrams, and the testing levels are unit and integration testing levels.

2. The Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It is also used for business modeling and other non-software systems. The UML represents a collection of engineering practices that have been used to model large and complex systems [9].

In the UML, complex systems are designed and modeled through a small set of nearly independent views of a model. The UML defines nine graphical diagrams to specify and design software: use case diagrams, class diagrams, object diagrams, collaboration diagrams, sequence diagrams, statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. The following subsections give a detailed description of statecharts and sequence diagrams, the two UML diagrams used in this experiment.

2.1. Sequence Diagrams

Sequence diagrams capture time dependent (temporal) sequences of interactions between objects. Sequence diagrams can be transformed to equivalent collaboration diagrams. Message sequence descriptions are provided in sequence diagrams to elicit meanings of the messages passed between objects. Sequence diagrams describe interactions among software components, and thus are considered to be a good source for integration testing.

Sequence diagrams include flows of events during interactions, with primary flows and *alternative* flows. Alternative flows represent conditional branches in the processing. For example, we describe the normal flow of events for “make call” as a flow of events that happen to make a successful call. Alternatives for this interaction include other event flows that cause “make call” to fail, including “callee busy”, “network unavailable”, and “caller aborts the call before connection is made”.

2.2. Statechart Diagrams

Statechart diagrams show states and state transitions to describe the behavior of the system. Statechart diagrams define the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events. Statechart diagrams describe behavior of individual software components, and thus are considered to be a good source for unit testing.

3. Description of the Experiment

Basili et al. [3] recommended a framework for designing and analyzing experimental work performed in software engineering. The suggested framework of experimentation consists of four phases: (1) definition, (2) planning, (3) operation, and (4) interpretation. Table 1 shows the definition phase of this experimentation. The motivation of this experimentation is to understand the roles of different UML diagrams in test case generation. To achieve this goal, test cases that are generated from UML statecharts and sequence diagrams are used both at unit and integration level testing, and their fault revealing capabilities are compared. This experimentation is designed from the perspective of a researcher, and is carried out as a case study (single project).

3.1. Hypotheses

A number of papers in the literature have made the assumption that effective tests at several levels (unit/module, integration, and system) can be created by basing the tests on specification and design artifacts that describe aspects of the software at those levels. [11, 1, 16, 8, 5]. One of our long term goals is to use various UML specification and design descriptions to develop tests, and to evaluate the assumption. As a beginning, we are comparing the fault revealing ability of tests cases generated from artifacts at different levels. We are also interested in the numbers of the test cases that result from different specification and design artifacts. We would like to see if there is correlation between the number of test cases in a test set and the number of faults revealed by that test set. As a beginning, we experimentally compare tests derived from UML statecharts, which are used to describe units and modules, and sequence diagrams, which are used to define the integration of modules.

The null hypotheses for our experiment are:

- H_{01}. There is no difference between the number of faults revealed by statechart and sequence diagram test sets at unit and integration testing levels.
- H_{02}. There is no difference between the number of test cases generated from statecharts and sequence diagrams.

Motivation	Understand the roles of different UML diagrams in test case generation.
Object	Theory
Purposes	Characterize the test cases that were generated from different UML diagrams, and compare their fault revealing capability.
Perspective	Researcher
Domain	Project
Scope	Single project

Table 1. Study Definition

3.2. Independent and Dependent Variables

Independent variables in this experiment are the types of UML diagrams, the testing levels used, the criteria used to create tests, and the faults that are used at each testing level. Statecharts and sequence diagrams are used because they are intended to help developers describe software at different levels of abstraction and because criteria for generating tests have previously been defined that easily be applied to these diagrams. The criteria based on statecharts are designed to be applied during unit and module testing, and the criteria based on sequence diagrams are designed to be applied during integration testing. These criteria are defined in Section 3.4. The faults are inserted by hand.

The dependent variables of the experiment are the two sets of test cases that are generated and the number of faults found at each level using these test sets.

3.3. Experimental Subjects

For this experiment, we modeled the software for a standard cell phone. The experimental materials that needed for this experiment are the following:

1. Specification and design documents of the cell phone handset system. This includes a class diagram of six classes, five statechart diagrams, and six sequence diagrams with 37 alternatives.
2. The implementation of the above specification and design, including eight classes of about 600 lines of Java code.
3. Test cases that are generated from the statecharts and sequence diagrams. There were **81** tests for the statecharts and **43** from the sequence diagrams.
4. A collection of **49** unit and integration level faults, each of which was placed into a separate copy of the program.
5. Unix shell scripts that run the test cases on each faulty version of the implementation and records the result.

Complete UML diagrams and implementation will be provided in an accompanying technical report [2]. A high level class diagram is shown in Figure 1. The cell phone includes initialization of the system, making a call, answering a call, notification of an incoming call, and notification of an incoming text message. The cell phone system in this experiment has eight classes. Five of the classes use state dependent design, hence, we have five state charts. They are `UserInterface`, `HandsetController`, `NetworkInterface`, `Transmitter`, and `Receiver`. The functionalities of the cell phone are described with six sequence diagrams with a total of 37 alternatives: `Initialization`, `Making a call`, `Answering a call`, `Notification of incoming call`, `Notification of incoming text message`, and `Turning off the cell phone`.

3.4. Generating Test Cases

Test cases were generated by hand following previously-defined test generation techniques. To eliminate potential for bias, tests were not generated by the same person who wrote the software and inserted faults. The test criteria used and process followed are detailed for each diagram below.

Sequence diagrams: In the UML, a *message* is a request for a service from one UML actor to another, these is typically implemented as method calls. Each sequence diagram represents a complete trace of messages during the execution of a user-level operation. We form *message sequence paths* by using the messages and their sequence numbers. *Message sequence paths* can be traces of system level interactions or component (object) level interactions. We defined the following coverage criteria for generating tests from sequence diagrams was defined elsewhere [1].

Message sequence path coverage: *For each sequence diagram in the specification, there must be at least one test case T such that when the software is executed using T, the software that implements the message sequence path of the sequence diagram must be executed.*

The *message sequence path* coverage criterion is used to generate tests from the sequence diagrams. For each se-

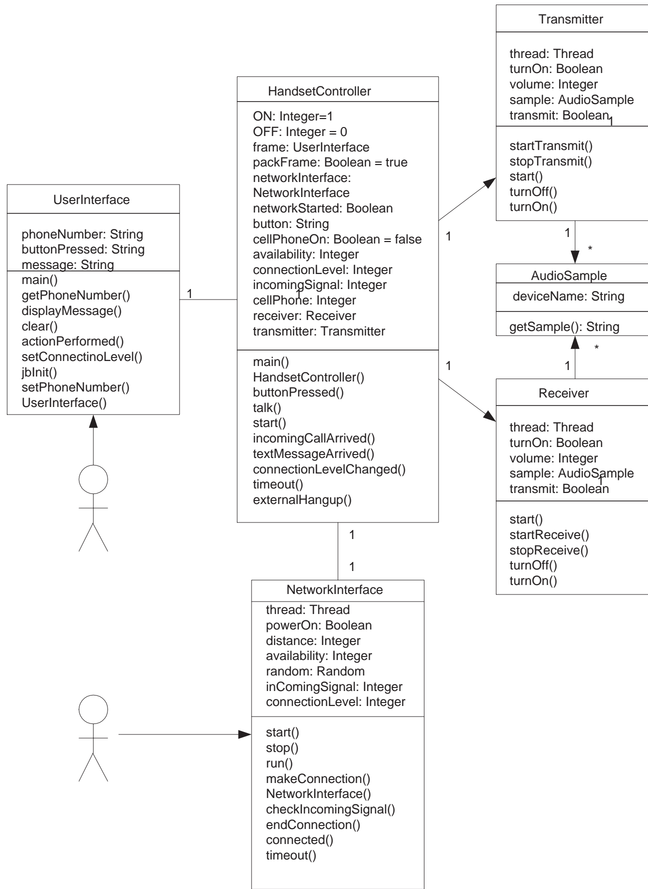

Figure 1. Cell Phone Class Diagram

quence diagram in the specification, a test case is generated for each normal and for each alternative message sequence.

Statecharts: UML statecharts are based on finite state machines using an extended Harel state chart notation, and are used to represent the behavior of an object. The *state* of an object is the combination of all values of attributes and objects the object contains. The dynamics of objects are modeled through transitions among states.

We use the *full predicate* test case generation criterion defined by Offutt et al. [12, 10, 11]. Statecharts represent *guards* and *actions* on transitions using predicates. The guards are conditions that must be true for the transition to be taken, and the actions represent what happens when the transition is taken. Full predicate coverage requires that for every transition, every predicate and every clause within the predicate has taken every outcome at least once, and every clause has been shown to independently affect its predicate.

Full Predicate Coverage: *For each predicate P on each transition, the test set T includes tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.*

In this definition, "directly correlated" means that c controls the value of P, that is, one of two situations occurs. Either c and P have the same value (c is true implies P is true and c is false implies P is false), or c and P have opposite values (c is true implies P is false and c is false implies P is true). This explicitly disallows cases such as c is true implies P is true and c is false implies P is true.

To satisfy the requirement that the *test clause* controls the value of the predicate, other clauses in the predicate must be either `True` or `False`. For example, if the predicate is $(X \wedge Y)$, and the test clause is X, then Y must be `True`. Likewise, if the predicate is $(X \vee Y)$, Y must be `False`.

The original full predicate coverage criterion was based on the notion of a predicate. The criterion considers transitions that are triggered by change events with or without other conditions that can be expressed in boolean expressions. However, UML statecharts have other types of events, *call events* and *signal events*. These events cannot be mapped directly into the existing full predicate testing method. To generate test cases, we first find out what event can trigger the starting transition of the statechart and under what conditions the event can be triggered. We then choose values to cause that event to occur and to satisfy the conditions.

Test case generation for the cell phone application yielded 81 test cases from statecharts, and 43 from sequence diagrams.

3.5. Program Faults

Unit level and integration level faults are inserted into the implementation by hand. We define unit level faults as faults that cause obvious and direct incorrect behavior of a unit. This includes most of the traditional mutation operators such as variable reference faults, operator reference faults, associative shift faults, variable negation faults, and expression negation faults [6, 7]. Integration faults are defined as faults that can cause two or more units to interact together incorrectly, even they are correct when tested in isolation. This includes faults such as incorrect method call, incorrect parameter passing, and incorrect synchronization. For this experiment, 30 unit level faults and 20 integration level faults were designed. We found one existing fault in the implementation, and three integration faults turned out to be similar and all failed under the same conditions. Thus, the experiment used 31 unit level faults and 18 integration faults.

The faults were inserted and tested in the following manner: one faulty version of the program is created at a time, and then ran against all the test cases one-by-one until either the fault is revealed or all test cases are executed. A fault is considered to be *revealed* if the output of the faulty

version of the program is different from that of the original program on the same input. That is, we used the original program as the "oracle". The faults were kept in separate versions of the program to make bookkeeping easier (when a failure occurred, it was clear which fault was found) and to avoid interactions between faults such as masking.

3.6. Experimental Procedure

The experiment was performed according to the following steps.

1. Analyze and specify the cell phone handset system using UML diagrams. The outcome of the specification and design include Class Diagrams, Statecharts, Collaboration/Sequence Diagrams, and Use Case diagrams.

2. Implement the system.

3. Manually generate test cases from statecharts (81) and sequence diagrams (43) to satisfy the testing criterion.

4. Manually generate faults for unit and integration testing level and insert them into the implementation.

5. Run each set of test cases from each diagram type on the implementation, and record faults found by their types.

The specification and implementation of the cell phone system was done by the first author, and test cases were generated by the third author. Faults were designed and inserted by the first two authors. Two software tools were used to specify and implement the subject system. The TogetherControl Center software (`http://www.togethersoft.com/`) was used to specify the cell phone system, and the JBuilder tool (`http://www.borland.com/jbuilder/`) was used to implement the system in Java. Both are leading tools that are widely used in industry.

4. Experimental Results and Analysis

The number of faults found during this experiment are given in Table 2. The rows represent the two types of faults, and the columns represent the number of faults found by the two types of test cases.

We can see from the data in the table that the statechart tests revealed more unit level faults than the sequence diagram tests, and the sequence diagram tests revealed more integration level faults than the statechart tests. Also, there are more statechart tests than sequence diagram tests. Hence, we can state that both null hypotheses are rejected.

The primary threat to the validity of the experimental data is external, this is only one application and one set of tests. Repetition of these results are needed in order to generalize the results.

There were also several lessons learned during this experiment. One thing that became apparent is that UML statecharts are not always sufficient for specifying low level details, particularly when great precision is required. The Object Constraint Language [15] can play a supplementary role for this purpose, and the integration of the OCL into the UML will provide better information for testing. Also, we need to incorporate the previously unconsidered event types into the full predicate criteria for UML statecharts.

Another problem encountered during this experiment was with concurrency. The expected execution trace that was developed from the sequence diagram sometimes turned out to be different from the actual execution trace because of concurrency interactions. This could be a potential problem in automating the testing process, and we probably need to incorporate some concurrent testing approaches [13, 14].

5. Conclusions and Future Work

This paper presented a single project experimentation on the fault revealing capabilities of test sets that are generated from UML statecharts and sequence diagrams. The results of this experiment shows that the statechart test sets have better capability of revealing unit level faults than the sequence diagram test sets, and the sequence diagram test sets have better capability of revealing integration level faults than the statechart test sets. The experimental data also shows that the statecharts resulted in more test cases than the sequence diagrams. The experiment showed that UML diagrams can be used to generating test data systematically, and different UML diagrams can play different roles in testing.

This experiment has limitations that difficult to avoid in this kind of study. We are not able to do statistical analysis because of the limited data from a single project. Also, we cannot state that this project uses the "sample representative" of the software population. This is a problem that plagues almost all software experimentation.

This is but one step in an ongoing research project. We have defined test criteria for UML statecharts and sequence diagrams. In the future, we plan to integrate UML specifications and OCL to generate stronger tests, and to develop new testing criteria to test concurrency aspects. Finally, large scale a multi-project experiment will be carried out to make a general conclusion on this topic.

	Number of Faults Inserted	Faults Found by Statechart TCs	Faults Found by Sequence Diagram TCs
Unit Faults	31	24 (77%)	20 (65%)
Integration Faults	18	10 (56%)	15 (83%)

Table 2. Experimental Results

References

[1] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the Third International Conference on the Unified Modeling Language (UML '00)*, pages 383–395, York, England, October 2000.

[2] Aynur Abdurazik, Jeff Offutt, and Andrea Baldini. Experimental evaluation of uml-based testing on a cell phone. Technical report ISE-TR-02-02, Department of Information and Software Engineering, George Mason University, Fairfax VA, 2002. http://www.ise.gmu.edu/techrep/.

[3] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[5] Philippe Chevalley and Pascale Thvenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *Proc. of IEEE 25th Annual International Computer Software and Applications Conference (COMPSAC2001)*, Chicago IL, October 2001.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991. http://ise.gmu.edu/faculty/ofut/rsrch/abstracts/cbt.html.

[8] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings – Software*, 146(4):187–192, August 1999.

[9] Object Management Group. *OMG UML Specification Version 1.3*, June 1999. Available at http://www.omg.org/uml/.

[10] A. J. Offutt. Generating test data from requirements/specifications: Phase III final report. Technical report ISE-TR-00-02, Department of Information and Software Engineering, George Mason University, Fairfax VA, January 2000. http://www.ise.gmu.edu/techrep/.

[11] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second International Conference on the Unified Modeling Language (UML '99)*, pages 416–429, Fort Collins, CO, October 1999.

[12] Jeff Offutt and Shaoying Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62, December 1999.

[13] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.

[14] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, 6(3):265–277, May 1980.

[15] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999. ISBN 0-201-37940-6.

[16] H. Yoon and B. Choi. Effective testing technique for the component customization in EJB. In *Proceedings of 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, Macau SAR, China, December 2001. To appear.