Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing

A Comparative Evaluation of Tests Generated from Different UML Diagrams

Supaporn Kansomkeat
Department of Computer Science
Faculty of Science, Prince of Songkla University
Hat Yai, Songkhla, 90112, Thailand
supaporn.k@psu.ac.th

Jeff Offutt, Aynur Abdurazik, Andrea Baldini
Software Engineering
George Mason University
Fairfax, VA 22030, USA
{offutt, aabduraz,baldini}@gmu.edu

Abstract

This paper presents a single project experiment on the fault revealing capabilities of model-based test sets. The tests are generated from UML statecharts and UML sequence diagrams. This experiment found that the statechart test sets did better at revealing unit level faults than the sequence diagram test sets, and the sequence diagram test sets did better at revealing integration level faults than the statechart test sets. The statecharts also resulted in more test cases than the sequence diagrams. The results show that model-based testing can be used to systematically generate test data and indicates that different UML models can play different roles in testing.

1. Introduction

Traditional testing has often generated tests from program source code, usually by abstracting the program into control flow diagrams, data flow graphs, call graphs, or other high level representations. Techniques to derive tests from formal specifications have also been developed. A more general term is that of *model-based testing*, which generally creates tests from an abstract model of the software, including formal specifications and semi-formal design descriptions such as UML diagrams.

When deriving tests from model-based descriptions of the software, the test engineer must choose among a variety of models, and decide which model to use. These choices can only be made on the basis of empirical data. This paper presents data comparing the use of statecharts and sequence diagrams for unit and model software testing. Specifically, this study asked whether tests that are designed from one type of software model to (theoretically) target one type of faults will find the faults that are targeted, and furthermore, whether the same tests can find other types of faults. This experiment is based on a non-trivial model of a cell phone, and tests are derived from UML statecharts and sequence diagrams and evaluated with hand-seeded faults.

In this paper, *unit and module testing* (or just unit testing) is testing program units and modules (classes and packages) independently from the rest of the program [3]. *Integration testing* refers to testing interfaces between units and modules to assure that they have consistent assumptions and communicate correctly. This is in contrast with *system testing* where the objective is to test the entire integrated system as a whole. A *test criterion* is a collection of rules that lead to *test requirements*, or specific elements in a program or model that must be covered during testing. Test suites are measured by how many requirements they satisfy.

2. The Unified Modeling Language

The Unified Modeling Language (UML) is a collection of languages for specifying, visualizing, constructing, and documenting the artifacts of software systems [14]. In the UML, complex systems are designed and modeled through a collection of views of a model. The UML defines nine separate graphical diagrams to specify and design software.

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 naturally 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 phone call” as a flow of events that

978-0-7695-3263-9/08 $25.00 © 2008 IEEE
DOI 10.1109/SNPD.2008.48

result in a successful call. Alternatives for this interaction include other event flows that cause "make phone call" to fail, including "callee busy," "network unavailable," and "caller aborts the call before connection is made."

Statechart diagrams describe software behaviors with states and state transitions. They define the dynamic behavior of software in terms of how it responds to external stimuli. Statechart diagrams are especially useful for modeling reactive objects whose states are triggered by specific events. Statechart diagrams describe behavior of individual software components, and thus are naturally considered to be a good source for unit testing.

3. Description of the Experiment

To be consistent with previous experiments, we choose to use the experimental framework by Basili et al. [5]. Their suggested framework contains four phases: (1) definition, (2) planning, (3) operation, and (4) interpretation. Table 1 shows the definition phase of this experiment. The motivation of this experiment 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 experiment is designed from the perspective of a researcher, and is carried out as a case study (single project).

Table 1. Study Definition

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 capabilities
Perspective	Researcher
Domain	Project
Scope	Single project

3.1. Hypotheses

A number of papers in the literature have assumed that effective tests at several levels (unit/module, integration, and system) can be created by basing the tests on specification and design model artifacts that describe aspects of the software at those levels [1, 6, 11, 15, 17]. One of our goals was to evaluate that assumption. As a beginning, we compared the fault revealing ability of tests cases generated from artifacts at different levels. We were also interested in how many test cases are required for different specification and design artifacts. Thus we compared the number of tests and the number of faults revealed by those tests. We first empirically compared tests derived from UML statecharts, which are used to describe units and modules, and sequence diagrams, which are used to define module integration.

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.

3.2. Independent and Dependent Variables

Independent variables in this experiment were the types of UML diagrams, the testing levels used, the criteria used to create tests, and the levels of faults (unit and integration). Statecharts and sequence diagrams were 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 could 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 were inserted by hand.

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

3.3. Experimental Subjects

We modeled software for a typical cell phone and developed five types of experimental materials.

1. Specification and design documents of the cell phone handset system. This includes class diagrams 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 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 ran the test cases on each faulty version of the implementation and recorded the results.

Space prohibits including all the UML diagrams in this paper. The diagrams, implementation, and other experimental subject material are in a technical report [2]. The cell phone use case diagram has two actors, *User* and *Timer*. The User actor has four use cases, *Initialization*, *Make a Call*, *Answer a Call* and *TurnOffCellPhone*. The Timer actor has three use cases, *Answer a Call*, *Notify Incoming Call* and *Notify Text Message*. Behaviors of all functions are described with six sequence diagrams with a total of 37 alternatives. These diagrams range in size from seven states with two levels of abstraction to two states.

The class diagram has six classes, *UserInterface*, *HandsetController*, *NetworkInterface*, *Transmitter*, *AudioSample*, and *Receiver*. Five use state dependent design and so have statecharts (all but *AudioSample*).

3.4. Generating Test Cases

Test cases were generated by hand. To eliminate bias, the third author wrote the software and inserted the faults, and the fourth author generated the tests. The test criteria used and the process followed for each diagram are described below.

Sequence diagrams: In the UML, a *message* is a request for a service from one UML actor to another. These are 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. The following coverage criteria for generating tests from sequence diagrams were defined previously [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 is executed.*

Statecharts: UML statecharts are based on finite state machines using an extended Harel statechart 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. Objects' behaviors are modeled through transitions among states.

We use the *full predicate (FP)* criterion defined by Offutt et al. [15]. FP is similar to the masking form of MCDC [7, 8] and CACC [3, 4]. 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. Both 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).

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

At the complete sequence level, test engineers must use their experience and judgment to develop sequences of states that should be tested.

As stated above, the full predicate and complete sequence coverage criteria are used with modifications in generating tests from statecharts. 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 full predicate testing method. Also, instead of considering each transition at a time, a complete sequence of transitions is considered for test case generation. 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. Next, we assign other values as necessary for the statechart to have the chosen complete sequence.

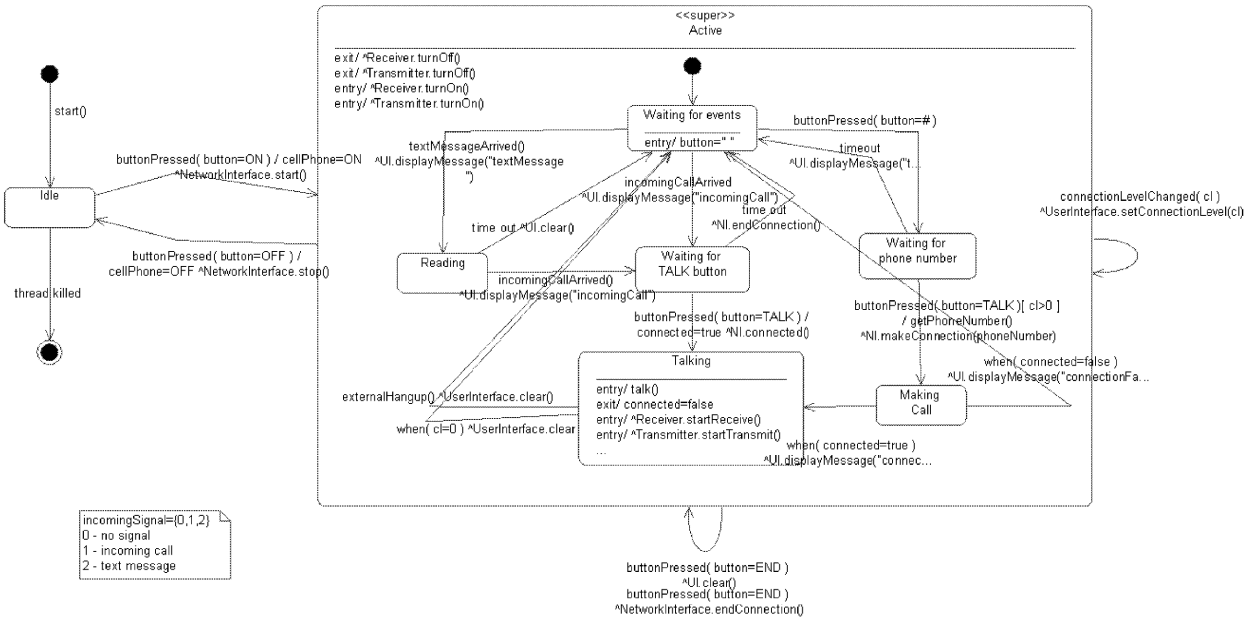

Figure 1. Handset Controller StateChart

A total of 81 test cases were generated from the statecharts and 43 from the sequence diagrams. As an example, consider the transition from "*waiting for phone number*" to "*making call*" in the state diagram for the handset controller, shown in Figure 1. The predicate on the transition is:

buttonPressed (*button* = *TALK*) [*cl* > 0] / *getPhoneNumber*() ∧ *NI.makeConnection* (*phoneNumber*)

The guard, *cl* > 0, refers to connection level, and takes an integer from 0 to 5. This guard indicates that the user can only make a call if the phone has connection. This guard results in two tests, one where *cl* > 0 and one where *cl* = 0.

Tests were created as sequences of method calls to the associated object. The mapping of values to method calls was done by hand. The tests for this transition are shown in Figure 2.

3.5. Program Faults

Unit level and integration level faults were inserted into the implementation by hand. We define a *unit level fault* as causing incorrect behavior of a unit when executed in isolation. This includes most of the traditional mutation operators, including variable reference faults, operator reference faults, associative shift faults, variable negation faults, and expression negation faults [9, 10]. We define an *integration fault* as causing two or more units to interact together incorrectly, even if they are correct when tested in isolation. This includes faults such as incorrect method calls, incorrect parameter passing, and incorrect synchronization. For this experiment, 30 unit level faults and 20 integration level faults were designed. We found one existing unit level fault in the implementation, and three integration faults turned out to be similar and all failed under the same conditions. These were combined, resulting in 31 unit level faults and 18 integration faults.

The faults were inserted and tested in the following manner: one faulty version of the program was created at a time, and then run against each test case in turn until either the fault was 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 carried out in five steps.

1. Analyze and specify the cell phone handset system using UML diagrams. This step resulted in class, statecharts, collaboration/sequence, and use case diagrams (third author).

2. Implement the system in Java (third author).
3. Generate test cases by hand from statecharts (81) and sequence diagrams (43) to satisfy the testing criteria (fourth author).
4. Design faults by hand for unit and integration testing level and insert them into the implementation. The second author designed the faults and the fourth author implemented them.
5. Run each set of test cases from each diagram type on the implementation, and record faults found by their types (third author).

4. Experimental Results and Analysis

The numbers 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 and percentage of faults found by the two groups of tests.

Table 2. Experimental Results

Fault Levels	Number Faults	Faults Found	
		Statechart Tests	Sequence Tests
Unit	31	**77%** (24)	65% (20)
Integration	18	56% (10)	**83%** (15)

We can see from the data in the table that the statechart tests revealed 12% more unit level faults than the sequence diagram tests, and the sequence diagram tests revealed 27% more integration level faults than the statechart tests. Also, there are 88% more statechart tests (81) than sequence diagram tests (43). Hence, both null hypotheses are rejected for these data.

An analysis of the faults revealed some insights into the techniques. The statechart tests found all 20 unit faults that the sequence diagram tests found, plus four more. Likewise, the sequence diagram tests found all of the 10 integration faults that the statechart tests found, plus five more. While it is tempting to conclude that sequence diagram tests are redundant for unit testing and vice versa, there is not enough data to make that general conclusion. The four unit faults missed by the sequence diagram tests were all related to functionality that was not used in the integrated cell phone system; so integration tests could not find them. Similarly, the five integration faults missed by the statechart tests were all related to functionality that could only be used when two classes were integrated together; so unit tests could not find them.

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 is needed to generalize the results.

Test 1

```
HandsetController.printStateInfo();
HandsetController.printStateInfo();
HandsetController.buttonPressed ("ON");
HandsetController.printStateInfo();
HandsetController.buttonPressed ("#");
HandsetController.printStateInfo();
HandsetController.connectionLevelChanged (0);
HandsetController.printStateInfo();
HandsetController.buttonPressed ("TALK");
HandsetController.printStateInfo();
HandsetController.buttonPressed ("OFF");
HandsetController.printStateInfo();
HandsetController.kill();
HandsetController.printStateInfo();
```

Test 2

```
HandsetController.printStateInfo();
HandsetController.printStateInfo();
HandsetController.buttonPressed ("ON");
HandsetController.printStateInfo();
HandsetController.buttonPressed ("#");
HandsetController.printStateInfo();
HandsetController.connectionLevelChanged (3);
HandsetController.printStateInfo();
HandsetController.buttonPressed ("TALK");
HandsetController.printStateInfo();
HandsetController.buttonPressed ("OFF");
HandsetController.printStateInfo();
HandsetController.kill();
HandsetController.printStateInfo();
```

Figure 2. The Tests for the transition from "waiting for phone number" to "making call"

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 [16] can play a supplementary role for this purpose.

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 problem in automating the testing process, and we probably need to incorporate some concurrent testing approaches [12, 13].

5. Conclusions and Future Work

This paper has presented a single project experiment on the fault revealing capabilities of test sets that are generated from UML statecharts and sequence diagrams. In this experiment, the statechart tests found more unit faults than the sequence diagram tests, and the sequence diagram tests found more integration faults than the statechart tests. There are

also almost twice as many statechart tests as sequence diagram tests.

Although the fact that we used only one project limits the general conclusions that can be drawn from this study, some preliminary conclusions can safely be made. First, it is reasonably effective to use UML diagrams as either a source for generating tests or as a way to evaluate tests generated elsewhere. Second, the data matches intuition, specifically that statecharts should be used for unit level testing and sequence diagrams should be used for integration level testing. This is evidence not only of the proper use of the UML diagrams for testing, but also that both unit and integration testing should be done.

The fact that more tests were generated to satisfy the statechart criterion (full predicates) than the sequence diagram criterion (message sequence paths) should not be surprising. Designers will usually include more detail and more decision points in statecharts than they will message sequence paths. This difference cannot be quantified, however, because the numbers can vary by developer and project.

6. References

[1] A. Abdurazik and J. 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)*, York, England, October 2000, pp. 383-395.

[2] A. Abdurazik, J. Offutt, and A. Baldini, "Experimental evaluation of UML-based testing on a cell phone", *Technical report ISE-TR-05-04*, Department of Information and Software Engineering, George Mason University, Fairfax VA, 2005. http://www.ise.gmu.edu/techrep/.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

[4] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions", In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, Denver, CO, IEEE Computer Society Press, November 2003, pp. 99-107.

[5] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering", *IEEE Transactions on Software Engineering*, SE-12(7), July 1986, pp. 733-743.

[6] P. Chevalley and P. Thevenod-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.

[7] J. J. Chilenski and S. P. Miller. "Applicability of modified condition/decision coverage to software testing", *Software Engineering Journal*, 9(5), September 1994, pp. 193-200.

[8] J. Chilenski and L. A. Richey, "Definition for a masking form of modified condition decision coverage (MCDC)", Technical report, Boeing, Seattle, WA, 1997. http://www.boeing.com/nosearch/mcdc/.

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

[10] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation", *IEEE Transactions on Software Engineering*, 17(9), September 1991, pp. 900-910.

[11] 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), August 1999, pp. 187-192.

[12] Y. Lei and R. Carver, "Reachability Testing of Concurrent Programs", *IEEE Transactions on Software Engineering*, 32(6), 2006, pp. 382-403.

[13] Y. Lei, R. Carver, R. Kacker, and D. Kung, "A Combinatorial Testing Strategy for Concurrent Programs", *Journal of Software Testing, Verification, and Reliability*, 17(4), 2007, pp. 207-225.

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

[15] J. Offutt and A. Abdurazik, "Generating tests from UML specifications", In *Proceedings of the Second International Conference on the Unified Modeling Language (UML '99)*, Fort Collins, CO, October 1999, pp. 416-429.

[16] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999. ISBN 0-201-37940-6.

[17] 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.