A Systematic Approach to Testing UML Design Models *

Trung Dinh-Trong
trungdt@cs.colostate.edu

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523

Abstract. Finding and removing faults in the design phase can reduce software development cost and time to market. Designs are typically evaluated using walkthroughs, inspections, and other forms of reviews that lack the rigor of systematic testing techniques. This research proposes a systematic approach to testing design models described by UML class diagrams, sequence diagrams, and activity diagrams. An executable form of a UML design model under test is exercised using generated input. The expected behavior of a design under test is compared with the actual behavior that is observed during testing. Failures are reported if the observed behavior differs from the expected behavior.

Keywords: software testing, test adequacy criteria, test input generation, UML, model execution, test execution, class diagram, interaction diagram, activity diagram

1 Introduction

A number of studies show that the majority of software system faults occur in the design phase [1]. Development time is wasted when faulty designs are implemented. If a design fault is detected after the code has been written, both the design and the code need to be changed. Furthermore, a small change in the design sometimes leads to a significant change in the code. Hence, finding and removing faults before designs are implemented helps reduce software development cost and time to market.

The Unified Modeling Language (UML) [2] is an OMG standard language for modeling object oriented systems. Software developers can use the UML to describe designs at different levels of abstraction, from conceptual to detailed design [3]. Because of the position of the UML as an industry standard, tackling the problem of evaluating UML designs can lead to techniques that are widely applicable. However, UML designs are typically evaluated using walkthroughs, inspections, and other types of design review techniques that are largely manual. These techniques are not effective when applied to complex UML designs.

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

Reviewers need to manually track and relate a large number of concepts across various diagrams, and the manual tasks can quickly become tedious and fault-prone. Also, designs tend to change during the development process. Given that complex UML designs are expressed using multiple views through multiple diagrams, such as class diagrams, interaction diagrams, statecharts and activity diagrams, the effects of changes are sometimes difficult to assess during reviews.

We propose a dynamic testing approach in which executable forms of UML design models consisting of class diagrams, interaction diagrams, and activity diagrams are exercised with test inputs. The expected behavior of a design under test is compared with the actual behavior that is observed during testing. Failures are reported if the observed behavior differs from the expected behavior. The proposed approach includes a set of design test adequacy criteria that are likely to lead to test suites that have high fault detection rates. The approach also includes a technique to systematically generate test suites satisfying these criteria. The research questions we plan to investigate are:

1. What are design test adequacy criteria that are likely to lead to test suites that have high fault detection rates?
2. How can test cases satisfying the criteria be systematically generated?
3. How can UML models be executed?
4. How can test executions and test results be observed? How can failures be detected?
5. What types of faults can be detected by the approach?

2 Related Work

We first describe the current status of research on testing UML design models. Then we present existing work on UML design execution. Finally, we summarize UML-based test input generation approaches.

2.1 UML Design Testing Approaches

Andrews et al. [4] define two sets of UML design test adequacy criteria that are based on coverage of elements in UML class diagrams and collaboration diagrams. These criteria can be used to assess the adequacy of a test suite, and also to guide the creation of new test inputs. Ghosh et al. [5] demonstrate in a case study that test inputs tend to cover multiple coverage elements.

Pilskalns et al. [6] propose a graph-based approach to combine the information from class diagrams and sequence diagrams. In this approach, each sequence diagram is transformed into an Object-Method Directed Acyclic Graph (OMDAG). The OMDAG can be used to derive test execution paths and their corresponding conditions, which are recorded in a table called the Object-Method Execution Table (OMET).

Gogolla et al. [7] present a tool named USE to validate UML class diagrams and OCL models using snapshots. A snapshot is an object diagram that represents system states at any time with objects, attribute values and links. Test

cases are used to demonstrate that snapshots can be constructed to obey constraints in the model. Invariants can be dynamically loaded and checked against the snapshots. We will incorporate the USE tool to (1) check if the runtime state of a system under test conforms to the specification, and (2) validate operation pre- and post-conditions.

2.2 UML Design Execution Techniques

Executing UML designs with test inputs requires an operational semantics for the UML. A number of techniques have been proposed to execute UML models. Mellor and Balcer [8] use domain-specific model compilers to produce executable UML models. Riehle et al. [9] propose a UML virtual machine that has the UML as its instruction set and memory management facilities of an existing Java Virtual Machine as the memory model. The advantage of using a virtual machine is that UML models can be executed without being transformed into another form. However, these approaches do not include test-specific details (e.g., test drivers and failure detectors).

Another technique for executing UML designs is to execute code that is generated from the model. Assuming that the code and model both contain the same information, executing the code is the same as executing the model. Harel and Gery [10] describe code generation from UML models consisting of class diagrams and statecharts. Engels et al. [11] present a set of rules to generate code from class diagrams and collaboration diagrams. Industrial tools such as Together [12] can generate code skeletons from both class diagrams and collaboration diagrams. The FUJABA tool [13] represents designs using class diagrams and a notation called *Story Diagram* which combines statecharts and collaboration diagrams. FUJABA translates story diagrams to skeletal Java code. Dinh-Trong [14] propose an extensive set of rules to generate skeletal code from models consisting of class diagrams, collaboration diagrams and activity diagrams. None of the above approaches generate test infrastructure code that supports systematic testing of models.

2.3 UML-based Test Input Generation Techniques

Offutt and Abdurazik [15] define four levels of test coverage for UML statecharts: transition coverage, full predicate coverage, transition-pair coverage and complete sequence. The approach supports only simple states, enable transitions and change events. Briand et al. [16] enhance the above approach to support call and signal events, as well as five types of actions: call, send, assignment, create, and destroy.

Abdurazik and Offutt [17] describe a set of test requirements based on collaboration diagrams for both static and dynamic evaluations.The authors define a test criterion which requires that all messages in collaboration diagrams must be sent at least one.

Scheetz et al. [18] develop an approach to generate system test inputs from UML Class Diagrams. They first identify test objectives for class diagrams. An

AI planner is used to convert the test objectives into test input sets, which are described as a sequence of system events.

Briand and Labiche [19] propose the TOTEM (Testing Object-Oriented Systems) system test methodology. Test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and OCL expressions across these artifacts. The technique to derive test inputs from the requirements are left for future work.

Though the above approaches are described to generate inputs to test code, we will adapt them to test design models. Most of the techniques, except [15] and [18], only provide test requirements without describing how to derive test inputs from the requirements. This problem will be addressed in our approach.

3 Overall Approach

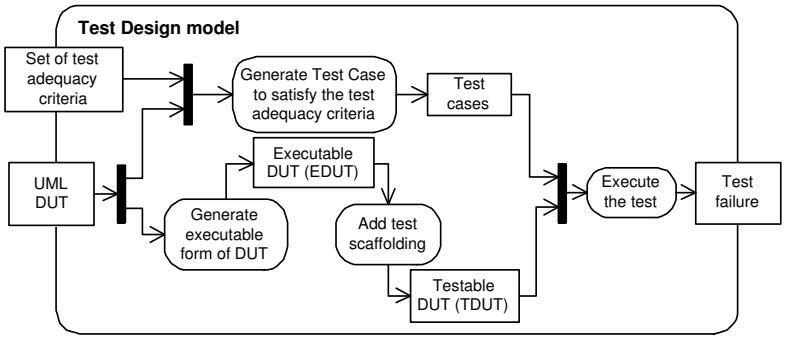

Fig. 1. Overview of the approach

The testing approach is applicable to models that consist of class, sequence and activity diagrams. A class diagram characterizes a set of valid object configurations. We require that operation pre- and post-conditions be represented using OCL. An interaction diagram characterizes an interaction that takes place between objects. Activity diagrams define system behaviors that occur inside each object (e.g., changes in attribute values). Such behavioral information cannot be captured in sequence diagrams.

The activity diagram in Fig. 1 summarizes the overall testing process. Testing begins when a tester selects a set of test adequacy criteria and provides the UML design model under test (DUT) to the testing system. The testing system generates a set of test cases for the DUT that satisfies the criteria. A test case is a tuple consisting of two components: a sequence of system event S, and a prefix P. Before a test is performed, the system under test is in an initial configuration C containing a set of objects that can create any valid configuration of the DUT.

The prefix, P, is a sequence of system events, which is applied to the system in the initial configuration to move it to the desired configuration in which testing can be started. The actual testing is done by applying the sequence of events, S, to the system under test.

The executable form of the design utilizes information in structural (class diagram) and dynamic (interaction diagrams and activity diagrams) aspects of design to carry out system operations defined in the design. The testing system also adds test scaffolding to the executable form to automate test execution as well as runtime failure detection. We call the combination of the executable form of the design and the test scaffolding the testable form $TDUT$.

Testing is performed by executing $TDUT$ with the generated test cases. During the test execution, the effects of system behaviors modeled by interaction and activity diagrams are observed in terms of changes in the configurations. The configuration of the DUT is updated continuously during the test. If a configuration produced during the test violates any constraint described by the class diagrams, a failure is detected.

3.1 Generating Executable UML Design ($EDUT$)

Executing UML designs requires an operational semantics for the UML. In our approach we use Java, a formal, executable language, to formally define the execution semantics of the UML. We transform UML design models (DUT) into executable Java programs ($EDUT$) that simulate the model. UML classes, attributes, and operations are transformed into Java classes, state variables and method declarations. UML association ends are also transformed into Java state variables.

Sequence diagrams are transformed into partial method bodies. Messages with *call* actions are transformed into Java method invocations. Messages with *create* actions are transformed into Java object creation statements. Java condition and loop structures are derived from message condition and iteration notations, respectively.

We describe actions in activity diagrams using a Java-like action language. This language has the semantics as described in the UML specification [20], and has a subset of Java as its syntactical representation. Code generated from activity diagrams is combined with code generated from sequence diagrams to form complete Java method bodies.

3.2 Generating Testable UML Design ($TDUT$)

As mentioned above, test scaffolding is added to the $EDUT$ to create the $TDUT$. Test scaffolding includes test drivers and code to detect failures. Test drivers consist of Java code to (1) create the initial configurations, (2) apply test inputs to the system, and (3) execute tests. Code to detect failures checks the following:

1. Are the variables in conditions (such as transition guards in activity diagrams or message conditions in sequence diagrams) initialized?

2. Are the parameters passed in method calls initialized?
3. Does the target object of a message exists?
4. Are the pre-conditions true before method execution?
5. Are the post-conditions true after method execution?
6. Is the configuration produced by the execution of an operation in the set characterized by a class diagram?

If any of the above check fails, a failure is reported. Our approach systematically adds code to the $EDUT$ to perform the first three checks. The last three checks are performed by delegating them to the USE tool [7].

3.3 Executing the test and detecting failures

The test execution is performed by applying test inputs to the generated program ($TDUT$). The testing system maintains the runtime configuration. When testing begins, the initial configuration is created. As the prefix and the sequence of events are applied, the current configuration is updated. For examples, objects and links are created or destroyed, and values of object attributes are modified.

During execution, the USE tool maintains its own representation of the object configuration. When testing starts, the testing system signals USE to create its representation of the initial configuration. Whenever the configuration changes, USE is informed about the modification, so that both USE and the testing system always maintain the same configuration. The testing system provides the USE tool with the pre- and post-conditions specified in the OCL. The testing system requests the USE tool to validate the pre-condition and post-condition of every operation before and after its execution respectively. Also, after the execution of any system event from the set S or P, the testing system signals USE to check the object configuration against the class diagram constraints.

3.4 Generating Test Inputs

Our approach includes the a set of criteria to measure the thoroughness of testing, as well as a technique to generate test input sets that satisfy the criteria. We will use Andrews et al.'s [4] criteria for class diagrams and sequence diagrams based testing. Given that both statecharts and activity diagrams are expressed as flow graphs, we will also adapt existing statecharts based test requirements ([15] and [16]) to define test adequacy criteria based on activity diagrams. For instance, transition coverage level in state-chart testing [15] suggests that we can define an all-edge coverage criterion for activity diagrams, which requires every activity edge to be exercised during testing.

We will generate test input from class diagrams using Scheetz et al.'s approach [18]. We will also adapt the OMDAG/OMET approach [6] to identify execution paths on activity diagrams and sequence diagrams, as well as their corresponding conditions as guidelines to generate test input. A technique to derive test inputs from these guidelines will also be developed.

4 Initial Results and Reseach Plan

To date we have developed an algorithm to execute and observe UML design testing. We have validated the fault detection ability of the algorithm using two case studies. The algorithm was executed by hand. The case studies used two models developed by students. The first model had 8 use cases, their corresponding interaction diagrams, and a class diagram with 6 classes. The second model had 9 use cases, their corresponding interaction diagrams, and a class diagram with 8 classes. We seeded faults into these models. Using our algorithm we were able to detect 8 out of 10 faults in the first model, and 5 out of 9 in the second.

We have built a tool that can read UML models in XMI format and produce Java programs (EDUT) that simulate the models. We are in the process of developing a tool that can transform the EDUT into a TDUT that incorporates the USE tool. We will use this tool to find the fault types that can be detected by our approach.

Our next task is to define test adequacy criteria based on activity diagrams and develop a technique to generate design test inputs. We will also conduct experiments to validate the effectiveness of our approach in detecting design faults.

5 Conclusions

We will deliver a dynamic approach to design testing. The methods and tools produced by our research could have a large impact on how UML designs are validated. We expect that our testing approach will help reduce the software development cost and time to market. Developers using our technique will detect more faults in less time than traditional review techniques. The approach will enable testing for consistency across multiple design views; this is especially important when views evolve during the course of software development. Our technique for test input generation from design artifacts may also be used as the basis for generating inputs for testing implementations.

Our research will deliver a set of formal UML execution semantics described in Java, which can be used to build tools to animate the behavior specified in UML models. Such tools can help students learn concepts in modeling and use of the UML by enabling them to visualize model behavior.

References

1. Pressman, R.: Software Engineering - A Practitioner's Approach. 7th edn. McGraw-Hill, New York, NY (2001)
2. Object Management Group: The Unified Modeling Language UML 1.5. Technical Report formal/03-03-01, The Object Management Group (OMG) (2003)
3. Booch, G., Rumbaugh, J., Hacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1999)

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. Ghosh, S., France, R.B., Braganza, C., Kawane, N., Andrews, A., Pilskalns, O.: Test adequacy assessment for UML design model testing. In: Proceedings of the International Symposium on Software Reliability Engineering, Denver, CO (2003) 332–343
6. Pilskalns, O., Andrews, A., Ghosh, S., France, R.B.: Rigorous testing by merging structural and behavioral uml representations. In: Proceedings of the 6th International Conference on the Unified Modeling Language, San Francisco, CA (2003) 234–248
7. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL models by automatic snapshot generation. In: Proceedings of the 6th Int. Conf. Unified Modeling Language (UML'2003), Springer, Berlin, LNCS 2863 (2003) 265–279
8. Mellor, S., Balcer, M.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley Professional (2002)
9. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), ACM Press (2001) 327–341
10. Harel, D., Gery, E.: Executable object modeling with statecharts. IEEE Computer **30** (1997) 31–42
11. Engels, G., Hucking, R., Sauer, S., Wagner, A.: UML collaboration diagrams and their transformations to Java. In: 2nd International Conference on the UML. (1999)
12. Borland Software Corporation: Together 6.0. `http://borland.com/together/` (2003)
13. Nickel, U., J.Niere, Wadsack, R., Zundorf, A.: Roundtrip Engineering with FUJABA. In: Proceedings of the 2th Workshop on Software-Engineering, Bad Honnef, Germany (2000)
14. Dinh-Trong, T.: Rules For Generating Code From UML Collaboration Diagrams and Activity Diagrams. Master's thesis, Colorado State University, Fort Collins, Colorado (2003)
15. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: 2nd International Conference on the UML. (1999) 416–429
16. Briand, L., Cui, J., Labichi, Y.: Towards automated support for deriving test data from UML statecharts. In: 6th International Conference on the UML. (2003) 265–279
17. Abdurazik, A., Offutt, J.: Using UML collaboration diagrams for static checking and test generation. In: 3rd International Conference on the UML. (2000) 383–395
18. Scheetz, M., von Mayrhauser, A., France, R., Dahlman, E., Howe, A.E.: Generating test cases from an OO model with an AI planning system. In: ISSRE'99. (1999) 250–259
19. Briand, L., Labiche, Y.: A UML-based approach to system testing. In: 4th International Conference on the UML. (2001) 194–208
20. Object Management Group: Unified Modeling Language: Superstructure. Technical Report ptc/03-07-06, The Object Management Group (OMG) (2003)