

EPDUT: An Eclipse Plugin for Testing UML Designs ^{*}

Trung Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert France¹
{trungdt,kawane,ghosh,france}@cs.colostate.edu
Anneliese A. Andrews²
aandrews@eecs.wsu.edu

¹ Computer Science Dept., Colorado State University, Fort Collins, CO 80523

² School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164

1 Introduction

Studies show that many software faults occur in the design phase. Detecting and removing these faults before the designs are implemented helps reduce software development cost and time to market. However, UML design models are typically evaluated using walkthroughs, inspections, and other informal types of design review techniques that are largely manual and consequently, tedious, error-prone and less effective.

We present an approach to dynamic testing of UML design models consisting of class diagrams, sequence diagrams, and activity diagrams. Models under test are converted into an executable form that utilizes an underlying test infrastructure. The models are exercised with generated test inputs. We have implemented the approach as an Eclipse plugin (EPDUT — Eclipse Plugin for Testing UML Designs).

2 Test Approach

We assume that the diagrams are syntactically well-formed when submitted for testing. This check can be done automatically by UML drawing tools. We also assume that the models under test describe sequential behavior only. We use the following types of actions in the activity diagrams: call operation actions, calculation actions, create and destroy object actions, create and destroy link actions, read and write link actions, and read and write variable actions.

Information from class and sequence diagrams is used to assess test adequacy. Information from class diagrams and activity diagrams is used to obtain the executable form of the design model.

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

Specification of the Design Under Test: The model of the design under test (*DUT*) is specified using the EPDUT graphical user interface. We describe actions in activity diagrams using *JAL*, a Java-like action language, which supports the action semantics described in the UML specification [1]. *JAL* has a subset of Java as its syntactical representation.

Before a test is executed, the system is in an initial configuration containing a set of objects that can create any valid configuration of the *DUT*. A prefix, which is a sequence of system events, is applied to the system in the initial configuration to move it to the desired configuration in which testing can be started. Testing is performed by applying to the system a sequence of system events. We restrict system events to be operation calls. An oracle is used to define the expected behavior of the system. It describes the conditions (expressed in the OCL) that need to be satisfied during and after each event.

Generation of Executable Form: The *DUT* is converted into an executable form *EDUT* using design information in structural (class diagram) and dynamic (activity diagrams) views of the design. UML classes, attributes, and operations are transformed into Java classes, state variables and method declarations. For each class *C* in the *DUT*, a collection class *SetOfC* is generated to take care of association-end multiplicities that are greater than 1. Association ends are transformed into Java state variables. Associations are transformed into Java attributes with collection class types.

A class named *TFactory* is generated from the class diagrams. This class has public methods to create and destroy instances of every class and association in the class diagrams.

Activity diagrams are transformed into Java method bodies using the following rules:

1. *Call* actions become Java method invocations.
2. Return actions become return statements.
3. *Create object* actions become Java object creation statements.
4. Java condition (`if ... then ... else ...`) and loop structures (`while ...`) are derived from activity condition and iteration structures respectively.
5. Object (or link) create and destroy actions are transformed into appropriate invocations of the methods in *TFactory*.

Generation of the Testable Form: Test scaffolding is added to the *EDUT* to obtain the *TDUT*. Scaffolding includes test drivers and code to detect test failures. Test drivers consist of Java code to (1) create the initial configuration, (2) apply test inputs to the system, and (3) execute tests. Failure detection involves execution of code that checks for certain failure conditions. The following are some of the conditions that are checked:

1. Uninitialized variables in conditions (such as transition guards in activity diagrams).
2. Uninitialized parameters passed in operation calls.
3. Non-existent target object of an operation call.

4. Pre-conditions before method execution evaluate to false.
5. Post-conditions after method execution evaluate to false.
6. Object configuration produced by the execution of a system event violates constraints imposed by a class diagram.

The first three checks are performed by code inserted in the *EDUT*. For the last three checks, we use the facilities provided by the USE tool [2]. USE is a tool that allows the validation of a configuration against the constraints described in a class diagram. This tool accepts UML class diagrams in its own textual format. Therefore, EPTUD transforms the *DUT* into USE format and provides it to USE.

Test Execution and Failure Reporting: Testing is performed by executing the *TDUT* using the generated test inputs. During test execution, the effects of system behaviors modeled by activity diagrams are observed in terms of changes in the configurations.

EPTUD provides the USE tool with pre- and post-conditions specified in the OCL and requests USE to validate them for every operation before and after its execution respectively. Also, after the execution of every system event in the test input, EPTUD signals USE to check the object configuration against the class diagram constraints.

During execution, the USE tool maintains its own representation of the object configuration. When testing begins, EPTUD signals USE to create its representation of the initial configuration. Because the tools perform a different set of failure checks, they maintain their own copies of the configuration. Whenever the configuration changes, USE is informed about the modification, so that both EPTUD and USE always maintain the same configuration. The changes in the configuration include adding or removing an object or a link, and modifying an attribute value.

The configuration of the *TDUT* is updated continuously during the test. If a configuration produced during the test violates any constraint described by the class diagrams, a failure is reported.

3 Conclusions and Future Work

We outlined a systematic approach to testing UML design models and described an Eclipse plugin that supports the approach. We are currently adding capabilities to visualize test execution through animated sequence diagrams and observe test coverage in the different views of the *DUT*. Future work also includes developing techniques for test input generation.

References

1. Object Management Group: The Unified Modeling Language UML 1.5. Technical Report formal/03-03-01, The Object Management Group (OMG) (2003)

2. 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