EPTUD: 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@eeecs.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

For model driven development approaches to succeed, there is a need for developing techniques for validating models. Studies show that many software faults occur in the design phase. Hence, it is essential to find and remove faults in design models. Currently, 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 testing 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 (EPTUD — Eclipse Plugin for Testing UML Designs).

2 Test Approach

We assume that the diagrams are syntactically well-formed. 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.

The activity diagram in Figure 1 summarizes the overall testing process. Testing begins when a tester provides the UML design model under test, DUT, to the testing system and selects a set of test adequacy criteria [1].

* This research was supported in part by National Science Foundation Award #CCR-0203285 and an Eclipse Innovation Grant from IBM.
A test case is a tuple consisting of three components: a prefix, $P$, a sequence of system events, $E$, and an oracle, $O$. Before a test is performed, the system is in an initial configuration 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. Testing is performed by applying to the system a sequence of system events, $E = \langle e_i : i = 1 \ldots n \rangle$, where $e_i$ is a system event. We restrict system events to be operation calls. The oracle, $O$, is used to define the expected behavior of the system. In our approach, an oracle is a sequence of tuples $(o_i, e_i)$, where $o_i$ is a condition (expressed in OCL) that the runtime configuration of the $DUT$ must satisfy after the system event, $e_i$, is executed.

The $DUT$ is converted into an executable form, $EDUT$, using design information in structural (class diagrams) and dynamic (activity diagrams) views of the design to simulate the behavior of the model. Test scaffolding is added to the executable form to automate test execution and enable runtime failure detection. The combination of the executable form of the design and the test scaffolding is called the testable form, $TDUT$.

Testing is performed by executing the $TDUT$ with 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. The configuration of the $TDUT$ is updated continuously during the test. Also, after the execution of each system event, $e_i$, the corresponding oracle condition, $o_i$, is checked. If a configuration produced during the test violates any constraint described by the class diagrams, or if any condition, $o_i$, evaluates to false, a failure is reported.

3 Eclipse Plugin

We have developed a prototype implementation of our approach in the form of an Eclipse plugin.

**Specification of the Model Under Test**: The Omondo EclipseUML plugin is used to draw class and sequence diagrams. Operation behaviors are described as actions using the Java-like Action Language (JAL) [2] developed by our research
group. Developers use the ecore system editor to specify operations and OCL constraints.

**Generation of the Executable and Testable Forms:** 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. An instance of \( SetOfC \) maintains a collection of instances of \( C \). The \( SetOfC \) class is needed to take care of association-end multiplicities that are greater than 1. The \( SetOfC \) class has methods to add (or remove) an instance of \( C \) to (or from) the collection. Association ends are transformed into Java attributes with collection class types. For more details on transforming UML class diagrams into Java, please refer to Dinh-Trong [3].

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 \( TFactory \).

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 conditions 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 [4]. USE is an open source tool that validates whether a configuration conforms to the constraints described in a class diagram. USE accepts UML class diagrams in its own format. Therefore, EPTUD transforms the \( DUT \) into USE format.
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 USE 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.

Because the tools perform a different set of failure checks, both maintain their own copies of the configuration during test execution. When testing begins, EPTUD signals USE to create its representation of the initial configuration. Whenever the configuration changes, USE is informed about the modification. 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.

4 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