UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs

Trung Dinh Trong, Sudipto Ghosh, Robert B. France, Michael Hamilton, Brent Wilkins
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
{trungdt, ghosh, france, hamiltom, wilkins}@cs.colostate.edu

ABSTRACT

We describe the UML Animator and Tester (UMLAnT), which is an Eclipse plug-in for animating and testing UML models. UMLAnT can be used both by developers in the software industry and students who are learning concepts in object-oriented modeling. UMLAnT helps designers visualize the behavior specified in a UML model by displaying animated object diagrams and sequence diagrams. UMLAnT allows testers to specify test cases and notifies them about failures during test execution.

Keywords

Eclipse, EclipseUML, EMF, UML, activity diagram, class diagram, model execution, model testing, object diagram, plugin, sequence diagram

1. INTRODUCTION

A number of software developers use the Unified Modeling Language (UML) to describe designs at different levels of abstraction, from conceptual to detailed design [1]. UML designs typically contain multiple views that show different aspects of the system being modeled. For example, the class diagram view shows the structural aspects, and the sequence and activity diagram views show the behavioral aspects. Currently, designers have no way to observe the modeled behavior until the designs are implemented and executable code is available.

Understanding and validating large, complex designs that are split into multiple views is tedious and challenging. Developers typically read UML models to understand them and perform manual reviews and inspections to establish their correctness.

In this paper, we present an Eclipse plugin, UMLAnT, that simulates the behavior modeled in a UML design. It helps developers visualize the execution of a model using two types of displays – object diagrams and sequence diagrams. The user can step through operations and the views get updated as execution proceeds. The object diagram shows the creation and deletion of objects and links, and the modification of attribute values. The sequence diagram shows the messages sent between objects. Test cases for the model can be specified using the JUnit style, and UMLAnT notifies the tester whenever a test fails.

UMLAnT can benefit Computer Science students who are beginning to learn object-oriented modeling and the UML. They can use UMLAnT to get quick visual feedback on their models. Software developers in the industry can use UMLAnT as a tool for rigorous testing of their models before the models are implemented and design faults are passed into the code.

2. UML DESIGN MODELS

In our approach, we can visualize and test models consisting of (1) class diagrams that characterize a set of valid object configurations, (2) sequence diagrams that characterize the interactions between objects, and (3) activity diagrams that describe class operations. OCL is used to describe invariants and operation pre- and post-conditions. We assume that the design models describe sequential behavior only. We also assume that the diagrams are syntactically well-formed when submitted to UMLAnT. This check can be done automatically by UML editing tools.

Class diagrams give us information about object configurations and are used both to define test objectives based on the types of configurations that we would like to be tested and to generate the executable form of the design model.

Sequence diagrams are used to define test objectives based on the conditions and paths that we would like to test in different scenarios. Because sequence diagrams only tell us what messages (e.g., operation calls) may be sent from one object to another, but not what happens inside the operation call, we obtain such information from the activity diagrams corresponding to the operations.

In our activity diagrams, we use the following types of actions: 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. The UML specification describes its action semantics, but does not prescribe a surface language for specifying actions. Therefore, we describe the actions using our Java-like Action Language (JAL)[1], which supports the action semantics described in the UML specification [4]. We designed JAL such that its syntax is similar to Java; developers who wish to implement the action semantics do not need to learn any new syntax. JAL has constructs that deal with design elements, such as creating and deleting instances and links.

Class diagrams and the JAL representation of actions are needed by UMLAnT to test and animate the models. Sequence diagrams are optional and are only used for identifying test objectives.

3. USING UMLANT

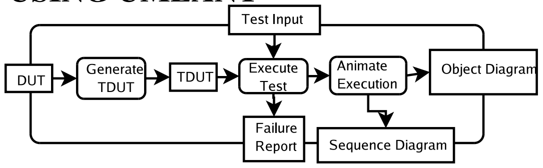


Figure 1: Overview of the Approach.

Figure 1 shows an activity diagram summarizing the process of using UMLAnT. The tester first provides the UML design model under test (DUT) to the UMLAnT. The EclipseUML plugin is used to draw class and sequence diagrams in Eclipse. UMLAnT provides the default ecore system editor that is used to specify operations in terms of actions using the JAL. A similar mechanism is used to specify OCL constraints.

The DUT is converted into an executable form using design information in structural (class diagram) 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, $TDUT$.

The tester begins test execution by providing UMLAnT with a test input. A test input is a tuple consisting of two components: a prefix P and a sequence of system events E. Before a test is performed, the system is in an initial configuration containing a basic set of objects (e.g., controller or factory instances) 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, E, of system events, $< e_i, i = 1..n >$, where e_i is a system event. We restrict system events to be operation calls.

UMLAnT extends the JUnit framework to support the testing of models. Test inputs are written in the form of JUnit test cases. Each method in a JUnit test class contains code to set up the initial configuration and a sequence of method calls that correspond to the sequence of operation calls. Currently the test cases are written manually. We are working on the automatic generation of test inputs in the form of JUnit test cases using ideas from symbolic execution and constraint solving.

[1]See `http://www.cs.colostate.edu/~trungdt/publication/NileshThesis.pdf`

Testing is performed by executing the $TDUT$ using the provided test inputs. During test execution, the effects of system behaviors modeled by activity diagrams are observed in terms of changes in the configurations. Test failures are reported by UMLAnT whenever the following situations occur:

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.

Testers use the failure data to locate the faults in the design. Examples of faults include incorrect multiplicity specifications on the association ends in a class diagram, or missing, faulty, and incorrectly ordered actions in an activity diagram.

During test execution, the trace of system behavior is recorded. At the end of the test, the tester can use UMLAnT to step through the trace. UMLAnT provides the object diagram view and the sequence diagram view to illustrate changes in the object configuration. The object diagram view shows how the object configuration changes during test execution. The sequence diagram views show the messages exchanged between the objects. Both views get updated as the tester steps through each action.

4. ECLIPSE PLUGIN

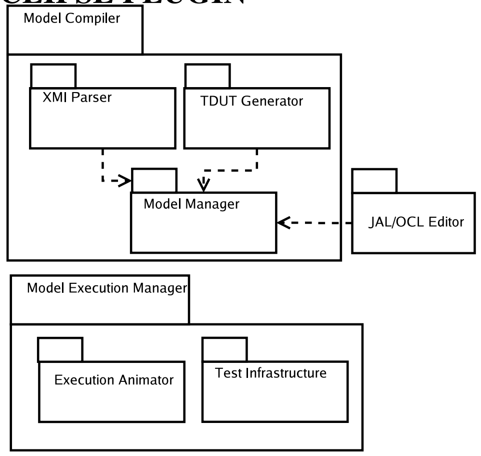
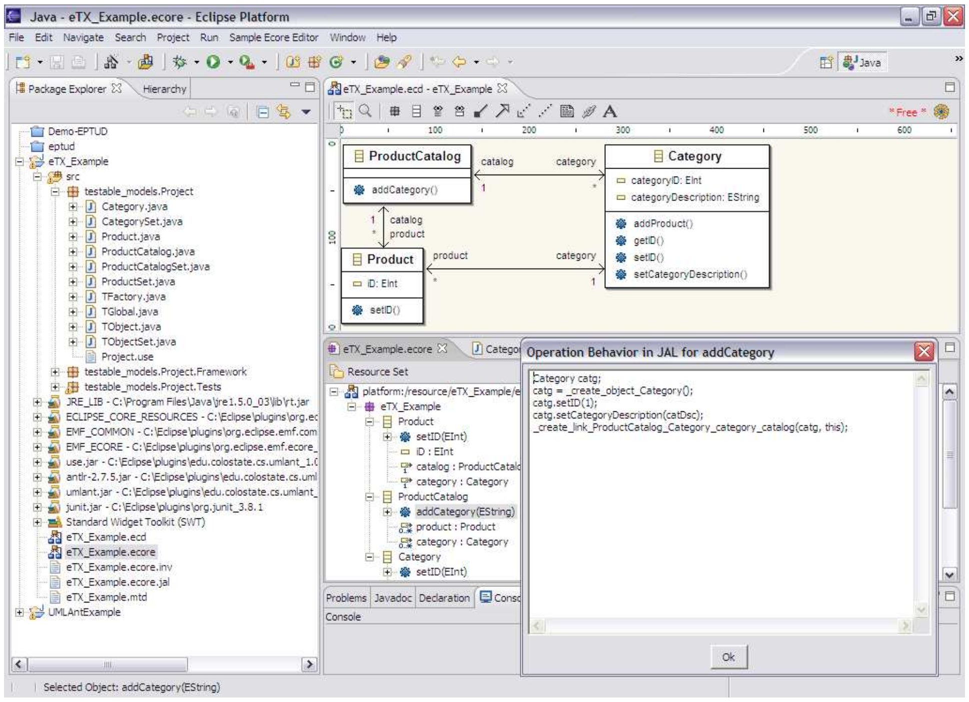


Figure 2: UMLAnT Architecture.

Figure 2 shows the architecture of UMLAnT. The subsystems are as follows:


Java - eTX_Example.ecore - Eclipse Platform
Package Explorer
Hierarchy
eTX_Example.ecd - eTX_Example
ProductCatalog
addCategory()
Category
categoryID: EInt
categoryDescription: EString
addProduct()
getID()
setID()
setCategoryDescription()
Product
ID: EInt
setID()
catalog
category
product
Operation Behavior in JAL for addCategory
Ok
Selected Object: addCategory(EString)

```
Category catg;
catg = _create_object_Category();
catg.setID(1);
catg.setCategoryDescription(catDsc);
_create_link_ProductCatalog_Category_category_catalog(catg, this);
```

Figure 3: UMLAnT Input Screen.

1. The *JAL/OCL Editor* is used to specify the model under test (*DUT*).
2. The *Model Compiler* generates the testable form (*TDUT*).
3. The *XMI Parser* parses the class diagrams saved in the XMI format.
4. The *Model Manager* maintains instances of the UML metamodel (i.e. the models under test).
5. The *Test Infrastructure* executes tests and reports failures.
6. The *Execution Animator* helps visualize the execution.

4.1 Model Specification

The Eclipse Modeling Framework (EMF) and Omondo EclipseUML plugins are used to draw and specify the *DUT*. An XMI Parser is used to parse the models and generate an instance of the UML metamodel inside the Model Manager. Figure 3 shows the model input screen of UMLAnT with an example operation specification presented by the editor in the bottom right of the figure.

4.2 Generation of the Testable Form

The UML design models are first transformed into executable Java programs that simulate the behavior of the model. 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 [2].

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*.

Test scaffolding is added to the executable Java program to generate the *TDUT*. Scaffolding includes test drivers and code to detect test failures. Test drivers contain 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 as described in Section 3.

Checking for uninitialized variables in conditions, uninitialized parameters in operation calls, and existence of target objects is performed by code inserted in the *TDUT*. Pre-conditions, post-conditions, and object configurations are checked using the facilities provided by the USE tool [3].

The USE tool enables validation of an object configuration against the constraints described in a class diagram. This tool accepts UML class diagrams in its own textual format. Therefore, UMLAnT transforms the *DUT* into USE format. During execution, the USE tool maintains its own representation of the object configuration. When testing begins, UMLAnT signals USE to create its representation of the initial configuration. As both 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 UMLAnT 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.

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

4.3 Test Execution and Failure Reporting

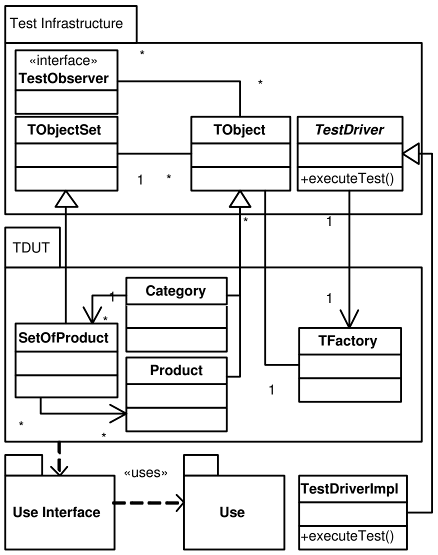


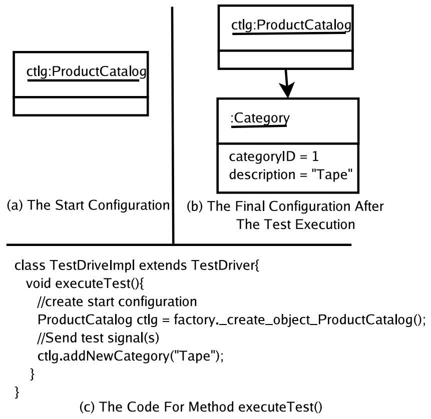
Figure 4: Classes in the Test Execution Phase.

Figure 4 shows the UMLAnT classes that are involved in the testing phase. The TestInfrastructure and TDUT packages are generated for every model. The TDUT package contains classes that are specific to the model, in this case, a ProductCatalog subsystem containing classes such as *Product* and *Category*. Only some classes in the *TDUT* are shown for lack of space.

TObject is a superclass of all the classes in the *TDUT*, which correspond to the classes in the *DUT*. *TObjectSet* is a superclass of the collection classes. *TestObserver* is an interface that allows the *Test Infrastructure* package to report failures.

TestDriver is an abstract class representing the test cases. It extends the JUnit framework to support a test environment for testing UML designs by providing a base class for the model test drivers, a graphical user interface for displaying progress and results of test execution, and an assertion function, **`assertConformance()`**, that delegates the validation of object configurations to the USE tool. The *TestDriver* class has an abstract method named **`executeTest()`**.

For each test case, the tester needs to create a class, *TestDriverImpl*, that is a subclass of *TestDriver*, and override the method **`executeTest()`**. The method body has two parts: a prefix to create the start configuration and a sequence of system operation calls. The prefix contains a series of **`TFactory`** method invocations to instantiate objects and links between them. In some cases the prefix may contain a few method invocations to set the object attributes. A system operation call is an invocation of a public operation of an object. The sequence of system operation calls in a test driver represents the sequence of system events in the corresponding test case.



```
class TestDriveImpl extends TestDriver{
  void executeTest(){
    //create start configuration
    ProductCatalog ctlg = factory._create_object_ProductCatalog();
    //Send test signal(s)
    ctlg.addNewCategory("Tape");
  }
}
```

(c) The Code For Method executeTest()

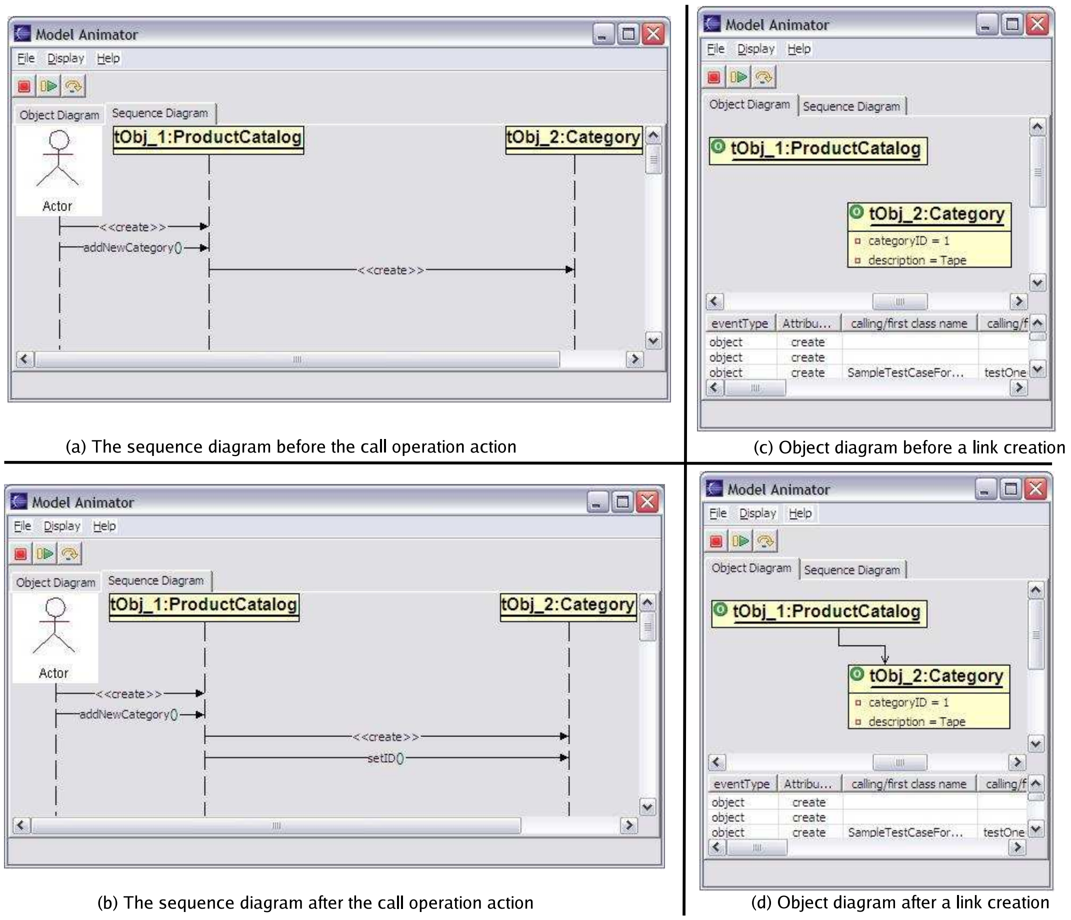
Figure 5: Object Configurations Resulting From `executeTest()`.

Figure 5 shows an example of the object configurations that result from the execution of a test method. Figure 5(c) shows the method **`executeTest()`** that a tester provides as a test input. The prefix part of the method creates the start configuration shown in Figure 5(a). The start configuration contains an instance of class *ProductCatalog*. The sequence of system operation calls contains the call **`ctlg.addNewCategory("Tape")`**. Figure 5(b) shows the final configuration after the test is complete.

When a test case denoted by the class *TestDriverImpl*, UMLAnT invokes the method **`executeTest()`**. The failure detected by USE or UMLAnT is reported using the interface **`TestObserver`**.

4.4 Model Animation

During test execution, the result of every action performed by each object is recorded in a log file. The test animator reads the log file and updates the sequence and object diagram views. Whenever the action involves changing an attribute value, and creation or deletion of a class or association instance, the object diagram view is updated. Whenever the action involves sending a message, and creation or deletion of an object, the sequence diagram view is



eventType	Attribu...	calling/first class name	calling/f
object	create		
object	create		
object	create	SampleTestCaseFor...	testOne

(a) The sequence diagram before the call operation action

(c) Object diagram before a link creation

(b) The sequence diagram after the call operation action

(d) Object diagram after a link creation

Figure 6: UMLAnt Animation Screen.

changed. Figure 6 shows an example of sequence diagrams (parts (a) and (b)) and object diagrams (parts (c) and (d)) created by UMLAnT during the animation of test execution. Figure 6(d) shows the creation of the link between the instances, `tObj_1:ProductCatalog` and `tObj_2:Category` in Figure 6(c).

5. CONCLUSIONS AND FUTURE WORK

We presented an Eclipse plugin, UMLAnT, that can animate and test the behavior of UML models. UMLAnT is integrated with some widely used software development technologies, tools and languages, such as Eclipse, EMF, JUnit, UML, and Java, thereby enhancing its applicability.

We are currently integrating a test input generation mechanism that uses symbolic execution and constraint solving techniques. We plan to add displays for test coverage measurement in each type of view. We are also working on validating the sequence diagrams obtained during execution against the sequence diagrams specified in the model.

Currently, the animations are performed after test execution is complete. We will integrate the plugin with the debug mechanism inside Eclipse so that we can perform the animations while the tests execute.

Acknowledgments

This material is based in part on work supported by the U.S. National Science Foundation under grant CCR-0203285 and IBM under Eclipse Innovation Grants.

6. REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[2] T. T. Dinh-Trong. Rules For Generating Code From UML Collaboration Diagrams and Activity Diagrams. Master's thesis, `http://www.cs.colostate.edu/~trungdt/code_generation/code_generation.htm`, Colorado State University, Fort Collins, Colorado, 2003.

[3] M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL models by automatic snapshot generation. In *Proceedings of the 6th Int. Conf. Unified Modeling Language (UML'2003)*, pages 265–279. Springer, Berlin, LNCS 2863, 2003.

[4] Object Management Group. The Unified Modeling Language 1.5. Technical Report formal/03-03-01, 2003.