

A Tool-Supported Approach to Testing UML Design Models

Trung Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert France
Colorado State University
Department of Computer Science
Fort Collins, Colorado 80523
{trungdt, kawane, ghosh, france}@cs.colostate.edu

Anneliese A. Andrews
Washington State University
School of Electrical Engineering and Computer Science
Pullman, Washington, 99164
aandrews@eecs.wsu.edu

Abstract

Dynamic testing of UML design models can reveal flaws that are more costly to fix after the design is implemented. This paper presents a dynamic approach to testing designs described by UML class diagrams, interaction diagrams, and activity diagrams. A UML design model under test is transformed into an executable form. Test infrastructure is added to the executable form to carry out the test. During testing, object configurations are created, modified and observed. In this paper, we identify the structural and behavioral characteristics that need to be observed during testing. We present a preliminary evaluation that identifies a set of design faults that were detected by the approach.

Keywords: *software testing, test adequacy criteria, UML, class diagram, model execution, test execution, code generation, interaction diagrams*

1. Introduction

A number of studies suggest that many software system faults occur in the design phase [18]. If a design fault is detected after the code has been written, both the design and the code need to be changed. Further, a small change in the design can sometimes lead to a significant change in the code. Finding and removing design faults before designs are implemented can help reduce software development cost and time to market.

The Unified Modeling Language (UML) [15] 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]. UML design models are typically evaluated using walkthroughs, inspections, and other informal types of design review techniques that are largely manual. These techniques are not effective when applied to UML design models of large or complex systems. 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, in which case reviewers need to account for these changes. The effect of changes are more difficult to assess in the presence of multiple views described by multiple diagrams, such as class diagrams, interaction diagrams, statecharts and activity diagrams.

We present a dynamic testing approach in which executable forms of UML design models under test are exercised with generated test inputs. The executable forms of the design models are generated from class diagrams and activity diagrams. 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.

We identify the functionality that executable forms of UML design models need to support. We also identify the structural and behavioral design characteristics that need to be observed during testing. We have developed a test infrastructure that is used to perform test execution and observe test results.

We describe the architecture of a tool that supports the testing approach. The tool (1) transforms a UML design model under test into a testable, executable form, (2) exercises the testable form with test inputs, and (3) reports failures.

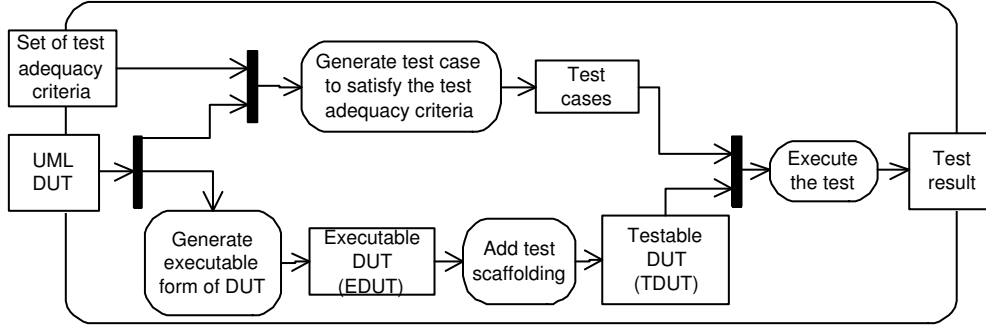


Figure 1. Overview of the testing process

This paper addresses the following research questions:

1. How can UML models be executed?
2. How can UML models be tested?
3. How can test results be observed?
4. How can failures be detected?

The rest of the paper is organized as follows. We describe our approach in Section 2. We present the architecture of a tool that supports the approach in Section 3. In Section 4 we summarize related work on UML based test input generation, UML design testing and UML design execution. We discuss our conclusions and outline directions for future research in Section 5.

2. Approach

The approach is applicable to design models that consist of class diagrams, interaction diagrams and activity diagrams. A class diagram characterizes a set of valid object configurations. OCL can be used with class diagrams to describe design invariants as well as operation pre- and post-conditions. An interaction diagram characterizes an interaction that takes place between objects. In our approach, activity diagrams are used to define class operations. Each activity diagram characterizes a sequence of actions that occur to accomplish an operation. Our approach accepts 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. We have developed a Java-like action language, *JAL*, which supports the action semantics described in the UML specification [15]. We use *JAL* to describe the actions in the activity diagram.

We assume that the models describe sequential behavior only. We also assume that the diagrams are syntactically well-formed. This check can be done automatically by UML drawing tools (e.g., Together [4] and Rational Rose [12]).

2.1 Testing Process

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 [2].

A set of test cases that satisfies the criteria is generated. 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. The actual testing is done by applying the sequence of events $E = \langle e_i : i = 1 \dots n \rangle$ to the system. We restrict system events to be operation calls. The oracle, *O*, is used to define the expected behavior of the system. An oracle is a sequence of tuples (o_i, e_i) , where o_i is the OCL constraint to be satisfied by the runtime configuration after the system event, e_i , is executed.

The testing system transforms the *DUT* into an executable form, *EDUT*, which is a program that simulates the behaviors modeled in the *DUT*. The *EDUT* utilizes information from structural (class diagrams) and dynamic (activity diagrams) descriptions of the design to carry out system operations defined in the design. Test scaffolding is added to the executable form to automate test execution and failure detection. We call the combination of the executable form of the design and the test scaffolding, the testable form, *TDUT*. The testable form of a *DUT* has the functionality of its corresponding *EDUT* along with the capability to execute test cases with provided inputs and report test failures.

Testing is performed by executing the *TDUT* with the generated test cases. During test execution, the effects of system behaviors modeled by activity diagrams are observed in terms of changes in the configurations. The con-

figuration of the *TDUT* is updated continuously during the test. A test failure is reported whenever a configuration produced during the test violates a constraint described by the class diagrams and their associated OCL invariants, or an oracle condition is violated.

Generating Executable UML Designs: Information from class diagrams and activity diagrams is used to obtain an executable form of design models. The *EDUT* simulates modeled behavior by maintaining and updating the runtime configuration of the *DUT*. The *EDUT* contains two parts: a static structure representing the runtime configuration of the *DUT*, and a simulation engine.

The *EDUT* static structure is generated from class diagrams, and can create and maintain runtime configurations of the *DUT*. A configuration contains objects, their attribute values, and the links between them. The *EDUT* simulation engine is generated from activity diagrams. This engine decodes system events, triggering sequences of actions according to the information modeled in the activity diagrams, and sends a sequence of signals to the *EDUT* static structure to update the configuration. The update involves adding and removing objects and links, as well as modifying attribute values.

Generating Testable UML Designs: Test scaffolding is added to the *EDUT* to create the *TDUT*. Test scaffolding executes the test and detects failures. Executing test includes creating the initial configuration and applying test inputs to the *TDUT*. Failure detection involves execution of code that checks for failure conditions. The following are some of the checks that are performed:

1. Are the variables in conditions (such as transition guards in activity diagrams) initialized?
2. Are the parameters passed in operation calls initialized?
3. Does the target object of an operation call exist?
4. Are the pre-conditions evaluated to true before method execution?
5. Are the post-conditions evaluated to true after method execution?
6. Does the configuration produced by the execution of system events violate constraints imposed by class diagrams? The set of constraints includes the association end multiplicity constraints and any other constraints expressed in OCL. We assume that these constraints must hold after the execution of every system event.
7. Do the oracle constraints evaluate to true?

TDUT reports a failure if any of the above checks return a negative answer.

Executing Tests and Detecting Failures: Test execution is performed by applying the prefix followed by the sequence of system events specified in a test case to the generated program. During the execution, the *TDUT* maintains the runtime configuration of the design under test. When testing begins, an initial configuration is created. As the prefix and sequence of events are applied, the runtime configuration is updated to reflect changes in the system state. The changes include creation and destruction of objects and links, as well as the modification of object attribute values.

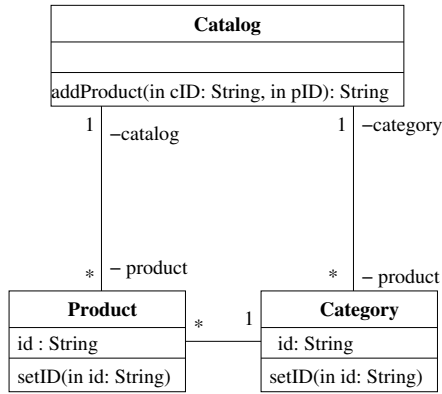
During test execution, the *TDUT* detects failures by checking the conditions described above. The causes for test failures are as follows:

1. Failing checks 1 – 3 indicates that there is a fault in the activity diagram.
2. Failing checks 4 or 5 indicates that the activity diagrams, the pre-conditions, and/or the post-conditions are faulty.
3. Failing check number 6 indicates that the class diagrams and/or the activity diagrams are faulty.
4. Assuming that the oracle is correct, failing check number 7 indicates that the activity diagrams are faulty.

Generating Test Inputs: Information from class diagrams and interaction diagrams is used to generate test input sets to assess the adequacy of testing. A test adequacy criterion defines the sets of the model entities that must be covered during the test. We use two sets of UML design test adequacy criteria that are based on coverage of elements of UML class diagrams and of interaction diagrams [2]. Three criteria were defined based on the coverage of association-end multiplicities, generalization-specialization relationships, and class attribute partitions in class diagrams. Four criteria were defined based on the coverage of conditions, predicates, messages on links, and message paths in collaboration diagrams.

2.2 Evaluation of the Approach

A couple of case studies were carried out to demonstrate the effectiveness of the test approach. The two systems used in the studies were: (1) a web-based course management system (WebC) and (2) a poker gaming system (Poker). The WebC system model consists of a set of eight use cases, corresponding collaboration diagrams, activity diagrams and one class diagram containing eight classes. The Poker system model consists of a set of nine use cases, their corresponding collaboration diagrams, activity diagrams and a class diagram containing six classes. The WebC system defines constraints on the values of some of the class attributes using the Object Constraint Language (OCL). The Poker system also defines pre- and post-conditions for system operations using the OCL. Each system has one instance of the



(a) Class diagram

```

[1] Category ctg;
[2] ctg = this.findCategory(categoryID);
[3] if ( ctg != null ) {
[4]   Product prd;
[5]   prd = _create_object_product();
[6]   prd.setID(productID);
[7]   _create_link_product_Category_pdcctg(ctg, prd);
[8]   return true;
[9] }
[10] else {
[11]   return false;
[12] }
  
```

(b) addProduct JAL specification

Figure 2. Product catalog system — Partial DUT

generalization-specialization relationship. Several collaboration diagrams contain conditional constructs, and thus, there is more than one possible execution path in the diagrams. In WebC, the system operations are independent of each other, whereas in Poker, the system operations must occur in a defined sequence.

A set of faults that commonly occur in UML models was compiled by identifying common faults that designers normally introduce in the constructs of a collaboration diagram. Each modeler seeded faults, one by one, in each collaboration diagram, thereby generating a number of faulty models. These models were given to testers. First, the testers randomly generated test cases using an automated test case generator. The generator uses information regarding parameter constraints on parameters of system operations and the sequence in which system operations can occur. The test cases that increased coverage with respect to one of the test criteria described in [2] were added to the test set. Since the randomly generated test cases did not result in complete coverage, a few test cases had to be added manually by the test engineer.

Table 1. Test Results for WebC and Poker Systems

System	Number of Test Cases	Number of Faults Seeded	Number of Faults Detected
WebC	13	10	8
Poker	14	9	5

Table 1 summarizes our results. The following types of faults were detected:

1. Using a variable before initialization.
2. Inconsistency between class and activity diagrams
3. Incorrect sequence of actions
4. Missing actions

3. Tool support

To automate the UML design testing approach, we have developed a prototype tool called “Eclipse Plugin for Testing UML Designs” (*EPDUT*). *EPDUT* operates in two phases: the pre-test and the testing phases. The pre-test phase involves transforming the *DUT* into the *TDUT*. Testers input UML design models under test (*DUTs*) that consist of class diagrams and activity diagrams into the tool. *EPDUT* transforms *DUTs* into *EDUTs* (executable *DUTs*), and then adds test scaffolding code to form *TDUTs* (testable *DUTs*). In the testing phase, testers enter test inputs into the *EPDUT*, which executes the *TDUT* with these inputs and reports failures.

We use a model of a product catalog system shown in Figure 2 to explain the details of the tool. Figure 2(a) shows a partial class diagram for the system, which contains three classes: *Catalog*, *Product*, and *Category*. Figure 2(b) shows the JAL specification of the *addProduct* operation to add a *Product* and corresponding *Category* objects to a *Catalog* object. Lines 1 and 4 are variable declarations. Line 2 shows an example of call action, and lines 3 and 10 contain conditional statements. Lines 5 and 7 show examples of special constructs *_create_object_CLASS-NAME* and *_create_link_ASSOCIATION-NAME_CLASS-NAME*. The first construct is used to create objects of a particular class and the latter is used to create links between two objects.

3.1 Generating testable design models

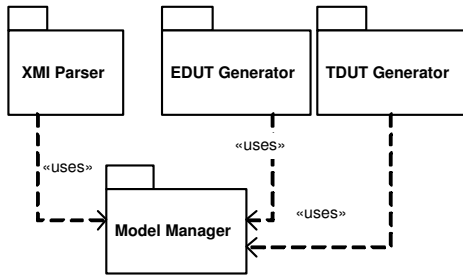


Figure 3. Pre-test subsystem

Figure 3 shows the subsystems of the tool that generates the testable form. A UML parser is used to parse the *DUT* and to generate an instance of the UML metamodel inside the Model Manager. The *EDUT* Generator transforms the *DUT* into an executable Java program (*EDUT*) that simulates the behavior of the model. The *TDUT* Generator adds test scaffolding to the *EDUT* to generate the *TDUT*.

3.1.1 Generating the *EDUT*

The *EDUT* Generator combines information from class diagrams and activity diagrams to generate Java programs. UML class, attribute, and operation notations are transformed into Java class, attribute, and method declarations, respectively. 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 [7].

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.

EPDUT allows testers to describe activity diagrams using JAL. 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 conditions (`if ... then ... else ...`) and loop structures (`while ...`) are derived from activity conditions and iteration structures respectively.

5. Object (or link) create and destroy actions are transformed into appropriate invocations of the methods in *TFactory*.

3.1.2 Generating the *TDUT*

Test scaffolding is added to the *EDUT* to perform failure checks. The first three checks mentioned in Section 2 are performed by code inserted in the *EDUT*. Figure 4 shows an example of test scaffolding added to check for the existence of the target object of a method invocation. In this example, the *EDUT* Generator transforms line 6 of the JAL specification (Figure 2) into line 1 of Figure 4(a). Figure 4(b) shows three lines of test scaffolding code (lines 1.1, 1.2, and 1.3) that are added by the *TDUT* Generator to check for the existence of the target object.

```

...
[1] product.setID(`00001`);
...

```

(a) *EDUT*

```

...
//BEGIN Test Scaffolding
[1.1] if(product == null)
[1.2]     reportError(NULL_ERROR);
[1.3] else
//END Test Scaffolding
[1.4]     product.setID(`00001`);
...

```

(b) *TDUT*

Figure 4. Test scaffolding example

For the checks 4-6, we use the facilities provided by the USE tool [10]. USE is an open source tool that validates whether a configuration conforms to the constraints described in a class diagram. To facilitate use of the tool, the *TDUT* Generator adds test scaffolding to *EDUT* to perform the following functions:

1. Inform USE about any changes in the state of the *EDUT* so that both *EPDUT* and USE always maintain the same configuration.
2. When there is a method invocation action, invoke USE to check the pre-condition.
3. When there is a return action, invoke USE to check the post-condition.

Figure 5 shows a *TDUT* for the `addProduct` operation

```

[1] public boolean addProduct( int categoryID, int productID ){
[2]     try{
[3]         use.optEnter("openter " + getUniqueName() + " addProduct(" +
[4]             String.valueOf(categoryID)+ "," + String.valueOf(productID) + ")" );
[5]     }
[6]     catch(Exception e){
[7]         System.out.println(e.getMessage());
[8]     }
[9]     boolean _ret = _addProduct(categoryID,productID);
[10]    try{
[11]        use.optExit( String.valueOf( _ret ) );
[12]    }
[13]    catch(Exception e){
[14]        System.out.println(e.getMessage());
[15]    }
[16]    return _ret;
[17] }
[18] private boolean _addProduct( int categoryID, int productID ){
[19]     Category ctg ;
[20]     ctg = this.findCategory(categoryID) ;
[21]     if (ctg!=null){
[22]         Product prd ;
[23]         if( prd == null ) {
[24]             reportError(``Message is sent to a null object``);
[25]         }
[26]         else {
[27]             prd.setID(productID) ;
[28]         }
[29]         prd = _factory()._create_object_product();
[30]         this._factory()._create_link_Product_Category_pdcatg(ctg,prd) ;
[31]         return true;
[32]     }
[33]     else {
[34]         return false;
[35]     }
[36] }

```

Figure 5. addProduct TDUT

specified in Figure 2(a). The *TDUT* generator generates code for two methods: `addProduct` and `_addProduct`. The Java code generated from the JAL specification in Figure 2 becomes a method body of the operation `_addProduct` in the *TDUT*. The method `_addProduct`, is called from `addPPProduct`. Lines 2-8 and 10-15 are inserted to allow the checking of pre- and post-conditions of the `addProduct` operation. In the example, the JAL construct `_create_link_` is transformed into a `TFactory` method call, `_create_link_ASSOCIATION-NAME()` (line 30), where `ASSOCIATION-NAME` is the name of the association between two corresponding classes.

3.2 Test execution

Figure 6 shows the EPTUD classes that are involved in the testing phase. Only some classes in the *TDUT* are shown for lack of space.

The *USE Interface* package represents a Java interface between the USE tool and *TDUT*. The *Test Infrastructure* package is used to coordinate test execution and to report test failures to the testers.

In the *TDUT*, the *TObject* is a superclass of all the *DUT* classes. The *TObjectSet* is a superclass of the collection classes. The *TestObserver* is an interface that allows the *Test*

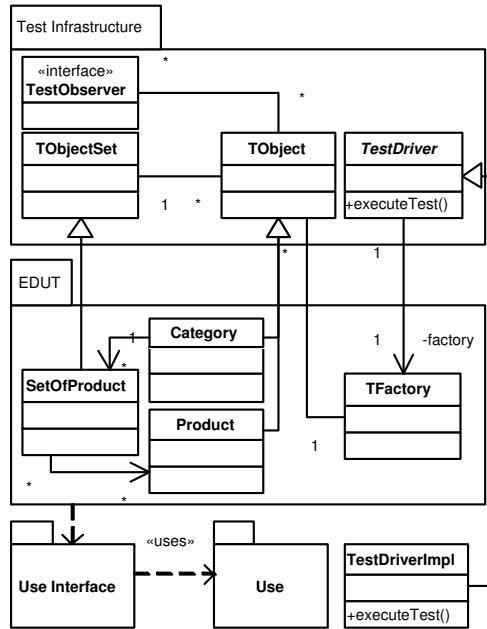


Figure 6. *T DUT* Package of test execution

Infrastructure package to report failures. The *TestDriver* is an abstract class representing the test cases. The *TestDriver* has an abstract method named `executeTest()`, which has an empty method body.

For each test case, the tester needs to create a class, *TestDriverImpl*, which 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.

Figure 7 shows an example of the method `executeTest()`. Figure 7(a) shows the start configuration. This configuration contains an instance of class *Catalog* and an instance of *Category* connected by an instance of the association between the two classes. Figure 7(b) shows the method `executeTest()` that a tester provides to *EPDUT* as a test input. The prefix part of the method creates the start configuration as shown in Figure 7(a). The sequence of system operation calls contains the method call `ctlg.addProduct('`CPT01'`, '`00001'')`.

When the tester executes a test case denoted by the class *TestDriverImpl*, *EPDUT* invokes the method

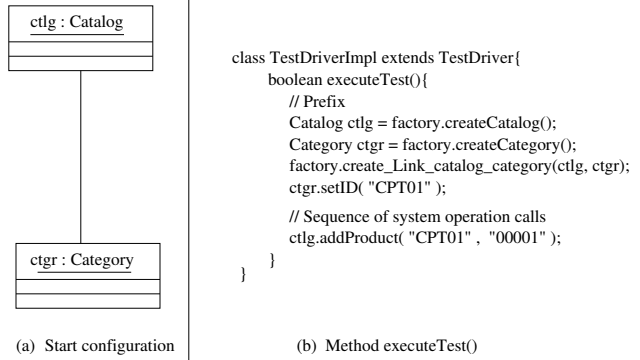


Figure 7. An example of a test case

`executeTest()`. Since USE accepts UML class diagrams in its own format, *EPDUT* transforms the *DUT* into USE format, and provides it to USE. When testing begins, *EPDUT* 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 USE and *EPDUT* always maintain the same configuration.

EPDUT 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, *EPDUT* signals USE to check the object configuration against the class diagram constraints. Any failure detected by USE or *EPDUT* is reported using the interface `TestObserver`.

Figure 8 shows a screen-shot for a sample test execution. The output window on the lower right shows a *Failure* test result for the test case shown in Figure 7. The failure occurs because the product object *prd* is not associated with any *Catalog* object. The class diagram specifies that a *Product* object must be associated with exactly one *Catalog* object. The corresponding fault can be fixed by adding the following JAL statement between line 7 and 8 of Figure 2(b):

```
_create_link_product_Catalog_pdctl(this,prd);
```

The above statement creates a link between the *Product* object, *prd* and the *Catalog* object *this*.

4. 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.

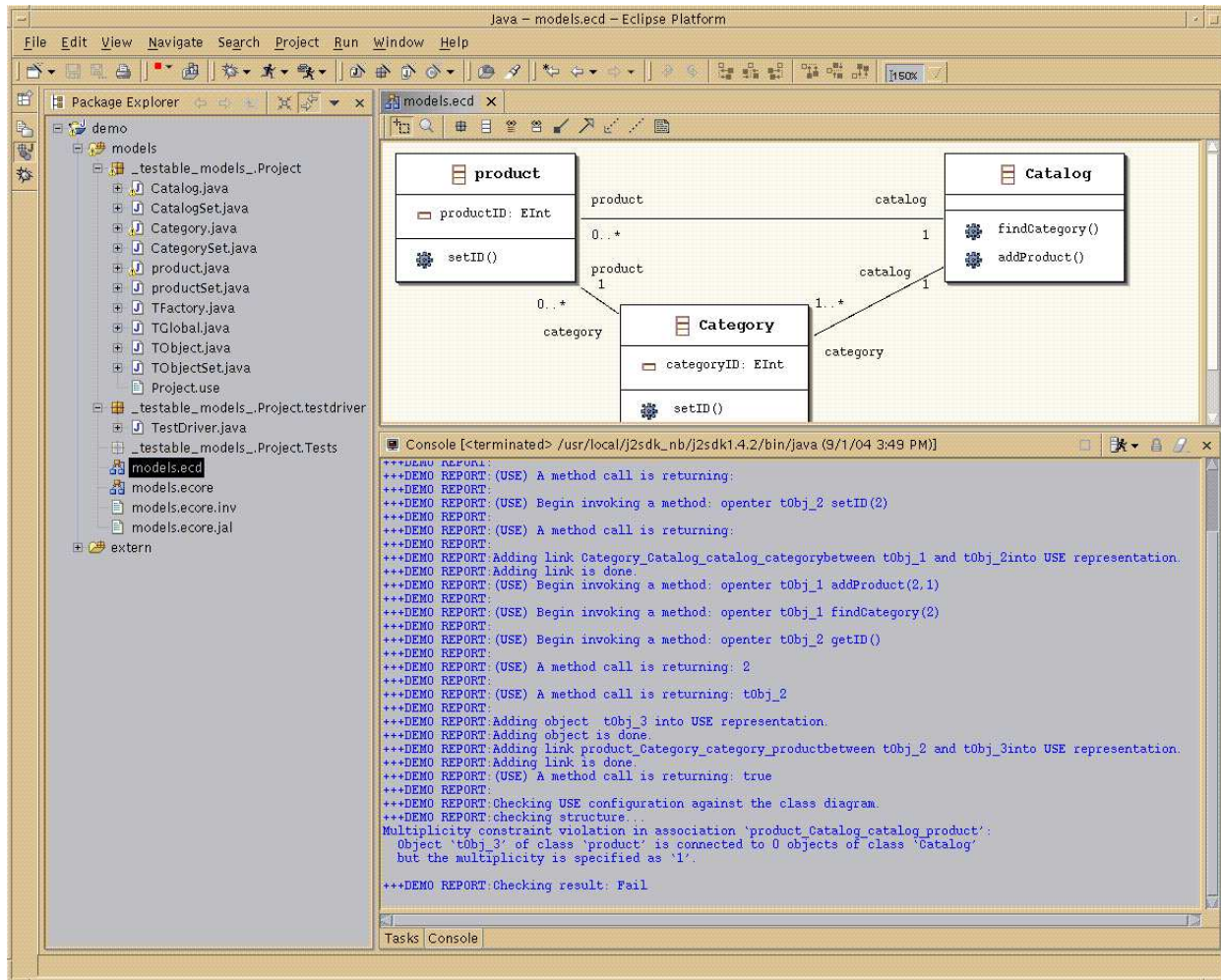


Figure 8. Test execution screenshot

4.1 UML Design Testing Approaches

Andrews et al. [2] 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. [9] demonstrate in a case study that test inputs tend to cover multiple coverage elements.

Piiskalns et al. [17] 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. [10] 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 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. We are planning to use this tool to validate constraints specified by users in an oracle.

4.2 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 [13] use domain-specific model compilers to pro-

duce executable UML models. Riehle et al. [19] 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. There are currently no publicly available virtual machines that cover the UML diagrams we are targeting in our work, and thus we had to investigate other approaches to executing design models.

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 [11] describe code generation from UML models consisting of class diagrams and statecharts. Engels et al. [8] present a set of rules to generate code from class diagrams and collaboration diagrams. Industrial tools such as Together [4] can generate code skeletons from both class diagrams and collaboration diagrams. The FUJABA tool [14] 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 [7] proposes an extensive set of rules to generate skeletal code from models consisting of class diagrams, collaboration diagrams and activity diagrams. To use any of the above approaches in our work would require extending the code generation mechanisms to support generation of the test infrastructure. We chose to extend the approach used by Dinh-Trong because we had access to code generation mechanisms.

4.3 UML-based Test Input Generation Techniques

Offutt and Abdurazik [16] define four levels of test coverage for UML state-charts: 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. [5] 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 [1] 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 once.

Scheetz et al. [20] 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 [6] 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.

All of the above approaches utilize the information in design models to test code. Our approach, on the other hand, focuses on testing the design model. However, these approaches can be utilized in our approach to derive test inputs for design models.

5. Conclusions

We presented a dynamic approach for testing UML designs. We transform UML designs under test into executable forms, and then add test scaffolding that performs test execution and observation. We described a list of conditions that are checked during testing and a set of failure types that are detected by our approach. We focus on finding failures that are likely to be related to design faults, such as the production of invalid configurations during test execution. Such faults are difficult to detect during reviews and walkthroughs. We applied the approach to two sets of faulty design models. The case studies demonstrated that the approach can find a number of faults in design models.

We have developed a tool as an Eclipse plugin to automate the approach. We are currently in the process of enhancing the tool to include checking oracle constraints. We plan to make the tool more interactive so that testers can enter activity diagrams to the tool in a graphical format. We also plan to add design debugging capabilities to the tool. Testers will be able to stop executing the model under test after a chosen action and review the current configuration of the *DUT*.

Currently, we only use class and interaction diagrams generate test input sets. We plan to extend the approach to consider information from activity diagrams and use case diagrams to generate test input. We are working on developing a technique to verify the consistency between activity diagrams and sequence diagrams. We are also working on empirical studies to assess the fault detection effectiveness of the approach.

References

- [1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *3rd International Conference on the UML*, pages 383–395, Oct. 2000.
- [2] A. Andrews, R. France, S. Ghosh, and G. Craig. Test Adequacy Criteria for UML Design Models. *Journal of Software*

- Testing, Verification and Reliability*, 13(2):95–127, April–June 2003.
- [3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [4] Borland Software Corporation. Together 6.0. <http://borland.com/together/>, 2003.
- [5] L. Briand, J. Cui, and Y. Labichi. Towards automated support for deriving test data from UML statecharts. In *6th International Conference on the UML*, pages 265–279, Oct. 2003.
- [6] L. Briand and Y. Labiche. A UML-based approach to system testing. In *4th International Conference on the UML*, pages 194–208, Oct. 2001.
- [7] T. Dinh-Trong. Rules For Generating Code From UML Collaboration Diagrams and Activity Diagrams. Master’s thesis, Colorado State University, Fort Collins, Colorado, 2003.
- [8] G. Engels, R. Hucking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformations to Java. In *2nd International Conference on the UML*, October 1999.
- [9] S. Ghosh, R. B. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 332–343, Denver, CO, 2003.
- [10] M. Gogolla, J. Böbling, 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.
- [11] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.
- [12] IBM Rational Rose URL <http://www.rational.com/>.
- [13] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley Professional, 2002.
- [14] U. Nickel, J. Niere, R. Wadsack, and A. Zundorf. Roundtrip Engineering with FUJABA. In *Proceedings of the 2th Workshop on Software-Engineering*, Bad Honnef, Germany, August 2000.
- [15] Object Management Group. The Unified Modeling Language UML 1.5. Technical Report formal/03-03-01, The Object Management Group (OMG), 2003.
- [16] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *2nd International Conference on the UML*, pages 416–429, Oct. 1999.
- [17] O. Pilskalns, A. Andrews, S. Ghosh, and R. B. France. Rigorous testing by merging structural and behavioral uml representations. In *Proceedings of the 6th International Conference on the Unified Modeling Language*, pages 234–248, San Francisco, CA, 2003.
- [18] R. Pressman. *Software Engineering - A Practitioner’s Approach*. McGraw-Hill, New York, NY, 7th edition, 2001.
- [19] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a UML virtual machine. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ’01)*, pages 327–341. ACM Press, 2001.
- [20] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an OO model with an AI planning system. In *ISSRE’99*, pages 250–259, 1999.