# A Tool-Supported Approach to Testing UML Design Models

Trung Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert France
Computer Science Department
Colorado State University
Fort Collins, CO 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

*For Model Driven Development approaches to succeed, there is a need for model validation techniques. This paper presents an 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 tests. 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 describe a prototype tool that (1) transforms UML design models into executable forms with test infrastructure, (2) executes tests, and (3) reports failures.*

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

## 1. Introduction

Model Driven Development (MDD) approaches tackle the complexity of developing large software systems by raising the level of abstraction at which developers build software. In MDD, developers focus on creating and evolving design models. MDD designs usually evolve in several steps, from higher level (e.g., Platform Independent Models (PIM)) to lower level models (e.g., Platform Specific Models (PSM)) [20]. Eventually, substantial portions of implementations are automatically generated from the designs. If a design model contains faults, those faults are passed to its refinements, including code. Finding and removing faults in an abstract design model can be less costly and require less effort than finding and removing the same faults in the refined models. Hence, for model driven development approaches to succeed, practical techniques for validating models need to be developed.

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.

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

In this paper, we describe how a UML design model can be executed and identify the structural and behavioral design characteristics that need to be observed during testing. We also describe the architecture of a prototype 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.

The rest of the paper is organized as follows. We describe our approach in Section 2. We present the architecture of a prototype tool that supports the approach in Section 3. In Section 4, we present the results of two case studies that were part of an initial evaluation of the proposed approach. In Section 5 we summarize related work on testing UML designs and executing UML models. We discuss our conclusions and outline directions for future research in Section 6.

## 2. Approach

In our approach, we assume that the models describe deterministic sequential behavior only. Thus, the state of the system can always be determined after the execution of an action. 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]).

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 interactions that take 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. The UML specification [15] does not define a language for describing actions in activity diagrams. We have developed a Java-like action language, $JAL$, which supports the action semantics of UML. In our approach, $JAL$ is used to describe the sequence of actions performed by a class instance during the execution of an operation call. The following types of actions can be included in the activity diagrams used in our approach: 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. Developers can use $JAL$ to express an activity diagram in a textual format.

### 2.1. JAL: The Java-like Action Language

The $JAL$ was designed so that (1) it is easy to learn by someone already familiar with Java, and (2) models expressed in the language are easy to transform into executable forms. The syntax of $JAL$ is based on Java, a widely used programming language.

$JAL$ call operation, computation, and return actions have the same syntax as Java method invocation, computation expression and return statement. $JAL$ also supports the following primitive actions:

- Create object action:
  *<objectHandle> = _create_object_<ClassName>()*
- Destroy object action:
  *_delete_object(<objectHandle>)*
- Create link action:
  *_create_link_<AssociationName>(<objectHandle1>, <objectHandle2>)*
- Delete link action:
  *_delete_link_<AssociationName>(<objectHandle1>, <objectHandle2>)*
- Get the number of objects at the other association end:
  *<AssociationEndName>.getTotal()*
- Access an object through an association end:
  *<AssociationEndName>.getAt(<index>)*
- Read attribute action:
  *_get_<AttributeName>()*
- Write attribute action:
  *_set_<AttributeName>(<value>)*

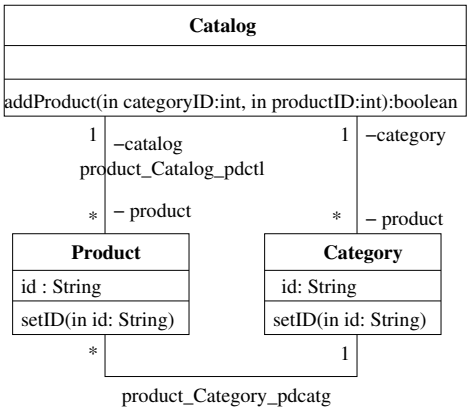
Following the UML standard's recommendation for action languages, $JAL$ supports control mechanisms besides primitive actions. Control mechanisms in $JAL$ are loop and condition structures, which have the same syntax as the 'if' and the 'while' structures in Java.

A $JAL$ statement can be either a simple statement (e.g., create an object), or a sequential logic structure (loop and condition). Each statement ends with a semicolon. $JAL$ supports the following primitive types: integer, real, boolean, and String. A variable can have a primitive type, or be an object handle. Variables need to be declared before use.

Figure 1 shows a partial UML design model with a $JAL$ specification. Figure 1(a) shows a partial class diagram containing three classes. The $Catalog$ class contains an `addProduct` operation that adds a $Product$ to a $Catalog$. Figure 1(b) shows the $JAL$ specification of the operation `Catalog::addProduct(int, int)`. Lines 1 and 4 are variable declarations. Line 2 shows an example of a call action and lines 3 and 10 contain condition structures. Line 5 is an example of a create object action, where an instance of the $Product$ is created. Line 7 is an example of a create link action, where a link between instances `ctg` and `prd` is created as an instance of the association between $Category$ and $Product$ classes.

### 2.2. Testing Process

The activity diagram in Figure 2 summarizes the 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) Partial 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_pdcatg(ctg, prd);
[8]      return true;
[9]   }
[10] else {
[11]     return false;
[12]  }
```

(b) JAL specification for the addProduct operation

**Figure 1. Product catalog system — Partial DUT**

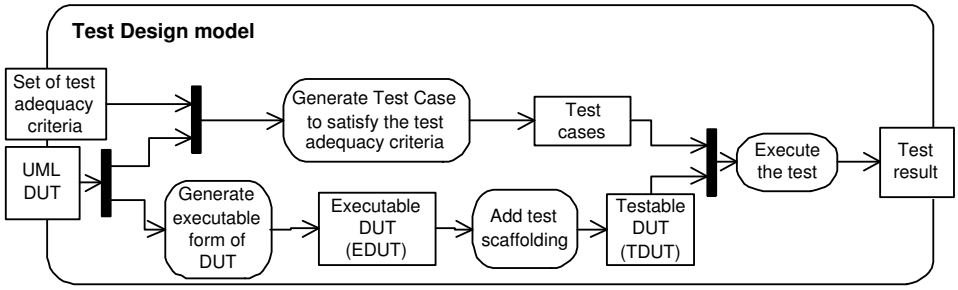



**Figure 2. Overview of the testing process**

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 are applied to the system in the initial configuration to move it to the configuration in which testing starts. Testing is performed by applying the sequence of events, $E =< e_i : i = 1 \ldots n >$, to the system. Each system event is a call to an operation. The oracle, $O$, defines 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.

### 2.2.1 Generating Executable UML Designs

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 simulate modeled behavior. The $EDUT$ contains two parts: a static structure representing the runtime configuration of the $DUT$, and a simulation engine. The static structure generated from class diagrams can create and maintain runtime configurations of the $DUT$. A configuration contains objects, their attribute values, and the links between them. The simulation engine is generated from activity diagrams (JAL specifications). This engine decodes system events, triggering sequences of actions according to the information 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.

### 2.2.2 Generating Testable UML Designs

Test scaffolding is added to the $EDUT$ to automate test execution and failure detection. The result is referred to as the testable form of the design, $TDUT$. Test scaffolding exe-

cutes the test and detects failures. Executing tests 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. Does the pre-condition evaluate to true before operation execution?
5. Does the post-condition evaluate to true after operation execution?
6. Does the configuration produced by the execution of system events violate constraints expressed in class diagrams? The set of constraints includes the association end multiplicity constraints and any other constraints expressed in OCL. These constraints must hold after the execution of every system event.
7. Do the oracle constraints evaluate to true?

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

#### 2.2.3 Executing Tests and Detecting Failures

Testing is performed by executing the $TDUT$ with provided test cases. This involves applying the prefix followed by the sequence of system events specified in a test case to the $TDUT$. During test execution, the effects of system behaviors modeled by activity diagrams are recorded and observed in terms of changes in the system state, where a system state is represented as an object configuration. 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. Some possible causes for test failures are given below:

1. Failing checks 1 – 3 indicate that there may be 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 may be faulty.
3. Failing check number 6 indicates that the class diagrams or the activity diagrams may be faulty.

#### 2.2.4 Assessing Test Adequacy

Information from class diagrams and interaction diagrams is used to assess the adequacy of test input sets. 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]. Ghosh et al. [9] present a case study where the criteria are used to define test objectives.

Three criteria were defined based on the coverage of association-end multiplicities, generalization-specialization relationships, and class attribute partitions in class diagrams. Test cases satisfying this set of criteria identify configurations that test various class diagram constraints. For example, the association end multiplicity criterion specifies that a test case must cause configurations to be created that contain representative association end multiplicity-pair values.

Four criteria were defined based on the coverage of conditions, predicates, messages on links, and message paths in collaboration diagrams. Test cases satisfying this set of criteria cause the models to be executed to test various interaction diagram elements. For example, a test set that satisfies the full predicate criterion ensures that every predicate in each condition inside the interaction diagram evaluates to true and false.

## 3. Tool Support

To automate the generation of the *TDUT* and support test execution, we have developed a prototype tool called the "Eclipse Plugin for Testing UML Designs" (*EPTUD*). *EPTUD* operates in two phases: the pre-test and the testing phases. The pre-test phase involves transforming the *DUT* into the *TDUT*. Testers provide a UML *DUT* that consists of class diagrams and activity diagrams as input to the tool. The tool transforms the *DUT* into a *TDUT*. In the testing phase, test inputs are entered by testers and used to execute the *TDUT*.

### 3.1. Generating Testable Design Models

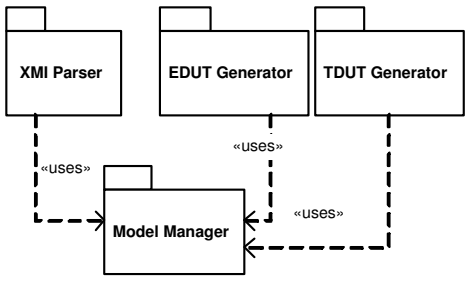



**Figure 3. Pre-test subsystem structure**

```
[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] } // EDUT code for addProduct() follows
[18] private boolean _addProduct(  int categoryID, int productID ){
[19]      Category ctg ;
[20]      ctg = this.findCategory(categoryID) ;
[21]      if (ctg!=null){
[22]           prd = _factory()._create_object_product();
[23]           if(prd == null)
[24]                reportError(``Message sent to null object'');
[25]           else
[26]                 prd.setID(productID);
[27]           this._factory()._create_link_Product_Category_pdcatg(ctg,prd) ;
[28]           return true;
[29]      }
[30]      else {
[31]           return false;
[32]      }
[33] }
```

**Figure 4. Partial TDUT generated for the `addProduct` operation**

Figure 3 shows the subsystem structure of the tool that generates the $TDUT$. An XMI parser is used to parse the *DUT* and generate an instance of the UML metamodel inside the *Model Manager*. The *EDUT Generator* transforms the *DUT* into an executable Java program 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 (remove) instances of *C* to (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 $DUT$ class diagram. This class has public methods to create and destroy instances of every class and association in the class diagrams.

The tool allows testers to describe activity diagrams using *JAL*. Activity diagrams are transformed into Java method bodies. For example, the $JAL$ specification in Figure 1(b) is transformed into lines 18-33 for the operation *_addProduct* in Figure 4 using the following rules:

1. *Call* actions become Java method invocations. For example, line 2 in Figure 1(b) becomes line 20 in Figure 4.
2. Return actions become return statements. For exam-

ple, line 8 in Figure 1(b) becomes line 28 in Figure 4.
3. Java conditions (`if ... then ... else ...`) and loop structures (`while ...`) are derived from activity conditions and iteration structures respectively. For example, lines 3 and 10 in Figure 1(b) become lines 21 and 30 in Figure 4.
4. Create and destroy actions for objects and links are transformed into appropriate invocations of the methods in `TFactory`. For example, lines 5 and 7 in Figure 1(b) become lines 22 and 27 in Figure 4.

#### 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*. Lines 23-25 in Figure 4 show an example of test scaffolding added to check for the existence of the target object, `prd`, of the `setID` operation call. Line 26 is a mehod invocation generated by the *EDUT generator*.

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

1. Inform USE about any changes in the state maintained by the *EPTUD* tool.
2. When there is an operation call action, invoke USE to check the pre-condition.
3. When there is a return action, invoke USE to check the post-condition.

Figure 4 shows the *TDUT* for the `addProduct` operation specified in Figure 1(a), produced by the *TDUT Generator*. The method, `_addProduct()`, generated by the *EDUT Generator*, is called from `addPProduct`. Lines 2-8 and 10-15 are inserted to check the pre- and post-conditions of the `addProduct` operation.

### 3.2. Test Execution

Figure 5 shows the packages that are involved in the testing phase. 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 test infrastructure, 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 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

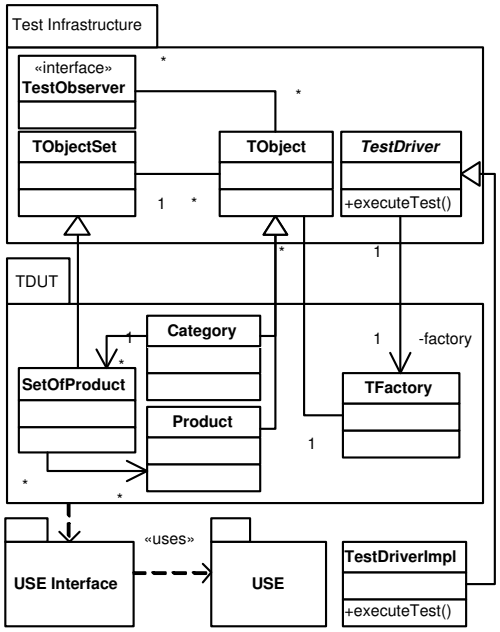



**Figure 5. $TDUT$ Test execution packages**

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 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 is the sequence of system events in the corresponding test case.

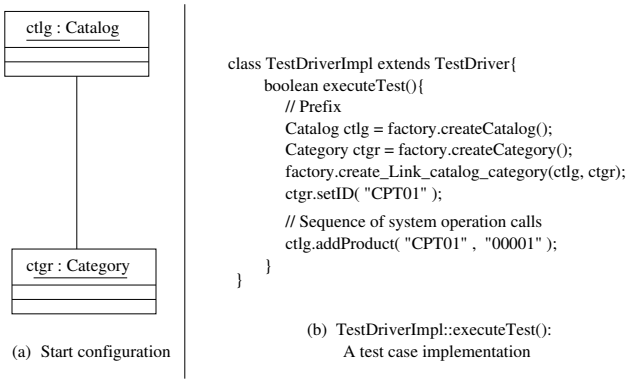



**Figure 6. A sample test case**

Figure 6 shows an example of the method `exe-`

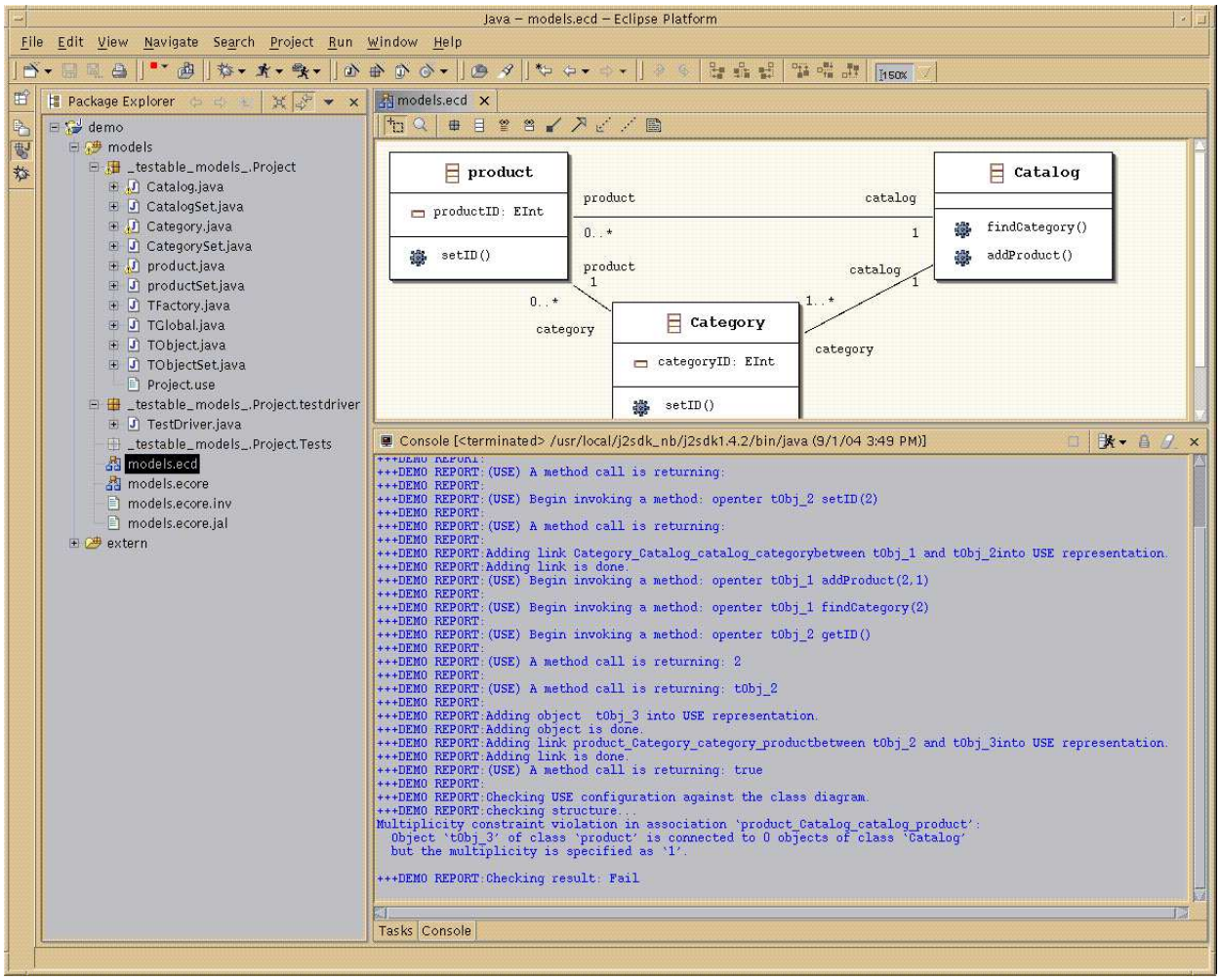



**Figure 7. Test execution screenshot from EPTUD**

`cuteTest()`. Figure 6(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 6(b) shows the method `executeTest()` that a tester provides as a test input. The prefix part of the method creates the start configuration as shown in Figure 6(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*, *EPTUD* invokes the method `executeTest()`. Since USE accepts UML class diagrams in its own format, *EPTUD* transforms the *DUT* into USE format, and provides it to USE. When testing begins, *EPTUD* 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 *EPTUD* always maintain the same configuration.

The *EPTUD* tool 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. Any failure detected by USE or *EPTUD* is reported using the interface, `TestObserver`.

Figure 7 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 6. The failure occurs because the *Product* instance `prd` is not associated with any *Catalog* instance. The class diagram specifies that a *Product* instance must be associated with exactly one *Catalog* instance. The corresponding fault can be fixed by adding

the following *JAL* statement between lines 7 and 8 of Figure 1(b):

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

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

## 4. Case Studies

We performed two case studies in an earlier phase of our research to get insight into the types of faults that can be detected by the 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 had a set of eight use cases, corresponding collaboration diagrams, and one class diagram containing eight classes. The WebC system defined constraints on the values of some of the class attributes using the Object Constraint Language (OCL).

The Poker system model had a set of nine use cases, their corresponding collaboration diagrams, and a class diagram containing six classes. The Poker system also defined pre- and post-conditions for system operations using the OCL.

Both systems had one instance each of the generalization-specialization relationship. Several collaboration diagrams contained conditional constructs, and thus, there is more than one possible execution path in the diagrams. In WebC, the system operations were independent of each other, whereas in Poker, the system operations must occur in a defined sequence. There was insufficient behavioral detail in the model because collaboration diagrams provide incomplete descriptions of behavior. The approach described in this paper provides more details in the form of activity diagrams.

A set of faults that commonly occur in UML models was compiled by identifying common faults that designers normally introduce while modeling behavior. 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 tester.

Table 1 summarizes our results. The following is a list of the types of faults detected and the corresponding checks that are performed by our test approach:

**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 |

1. Missing condition: A condition was not specified causing a call to an operation of an object that was not created. As a result, check 3 failed.
2. Incorrect sequence of actions: A variable is used before initialization. As a result, check 1 failed.
3. Missing a create operation call: This can result in an incorrect configuration or a subsequent call to an nonexistent class instance. Thus, checks 3 and 6 failed.
4. Missing or incorrect operation call: This caused checks 4 and 5 to fail because a pre- or post-condition became false. When a resulting configuration did not conform to the class diagram, check 6 failed.

We expect that with the use of activity diagrams, testers will have access to more detailed behavioral information and thus, be able to uncover faults more effectively.

## 5. Related Work

We describe related work on testing and executing UML designs. We also summarize work on generation of test objectives that can be used in our test approach.

### 5.1. UML Design Testing Approaches

Pilskalns 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). While their approach can help systematically define test inputs, it lacks support for model execution. The proposed approach can complement their approach.

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. The USE tool is limited to be used with class diagrams and related constraints. It cannot utilize information from some of the other UML model artifacts like state charts and activity diagrams for model execution. We are planning to use this tool to validate constraints specified by users in an oracle.

### 5.2. 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 produce executable UML models. Riehle et al. [18] 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.

### 5.3. Generating Test Objectives from UML Designs

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. [19] 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.

## 6. Conclusions and Future Work

We presented an approach to testing UML designs. UML designs under test are transformed into testable forms that include code for performing 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. These failures types are likely to be related to design faults that can be difficult to detect during reviews and walkthroughs.

We have developed a prototype tool as an Eclipse plugin to automate the approach. We are currently in the process of enhancing the tool to include checking of oracle constraints. We also plan to add design debugging capabilities to the tool. Testers will be able to stop execution of the model under test after a chosen action and review the current object configuration.

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

## Acknowledgements

This material is based in part on work supported by the U.S. National Science Foundation under grant CCR-0203285 and an Eclipse Innovation Grant from IBM. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM.

## 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. Labiche. 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. 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. 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.

[11] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[12] IBM. Rational Rose. `http://www-306.ibm.com/software/awdtools/developer/rosexde/`, 2004.

[13] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley Professional, 2002.

[14] U. A. Nickel, J. Niere, R. P. 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] 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.

[19] 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.

[20] The Object Management Group. MDA Guide. Version 1.0.1, OMG, omg/03-06-01, 2003.