Test Adequacy Assessment for UML Design Model Testing *

Sudipto Ghosh, Robert France, Conrad Braganza, Nilesh Kawane
Computer Science
Colorado State University
Fort Collins, Colorado 80523
{ghosh,france,braganza,kawane}@cs.colostate.edu

Anneliese Andrews, Orest Pilskalns
Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164
{aaa,orest}@eecs.wsu.edu

Abstract

Systematic design testing, in which executable models of behaviors are tested using inputs that exercise scenarios, can help reveal flaws in designs before they are implemented in code. We present a testing method in which executable forms of the Unified Modeling Language (UML) models are tested. The method incorporates the use of test adequacy criteria based on UML model elements in class diagrams and interaction diagrams. Class diagram criteria are used to determine the object configurations on which tests are run, while interaction diagram criteria are used to determine the sequences of messages that should be tested. The criteria can be used to define test objectives for UML designs. In this paper, we describe and illustrate the use of the proposed test method and adequacy criteria.

Keywords: *design reviews, software testing, test adequacy criteria, UML, class diagram, collaboration diagram, category partitioning*

1. Introduction

The Unified Modeling Language (UML) [14] is an Object Management Group (OMG) standard for object-oriented modeling that has gained widespread use in the software development industry. Using the UML, developers model large, complex systems and produce a variety of diagrams presenting different views of the system model. Assessing design quality, and detecting and correcting design faults in the model can reduce the total software development costs and time to market. Design models are typically evaluated in informal design inspections and walkthroughs. Use of these informal techniques for evaluating large models can be problematic because of the difficulty of tracking and relating a variety of concepts across multiple diagrams. The informal semantics associated with UML models also makes it difficult to uncover design faults.

We present a testing technique for UML designs that is based on a well-defined semantics that supports the execution of models. Dynamic analysis of UML models involves testing modeled behaviors by executing models using appropriate forms of test inputs. Currently, the semantics of the UML is informally described in the OMG standard document [14]. However, executable forms of the UML, such as the Executable UML [7] and the UML Virtual Machine [12], are being developed.

The effectiveness of a test is based on how well the tests cover and exercise the modeled behaviors. Analogous to program testing where criteria based on coverage of building blocks of a program are used to determine adequacy of tests, criteria can also be defined based on coverage of UML design elements. The basic building blocks considered in our work are elements in the static structural diagrams (e.g., class diagrams) and behavioral diagrams (e.g., interaction and state diagrams).

We present a brief description of executable UML semantics and its use in test execution. For a more detailed account, we refer the reader to Andrews *et al.* [2]. The focus of this paper is on the use of test criteria to define objectives that aid in the creation and improvement of test cases. We demonstrate how individual test cases can cover several el-

*This research was partially supported by National Science Foundation Award #CCR-0203285.

ements of multiple coverage domains. The effectiveness of the test cases with respect to fault detection is not examined in this paper.

The remainder of the paper is structured as follows. We present concepts related to the executable form of UML models and related work on UML testing in Section 2. We present the test criteria based on elements in UML models and a systematic testing approach in Section 3. We illustrate the use of test criteria in generating test cases for a design model of a university course administration system in Section 4. Conclusions and directions for future work are presented in Section 5.

2. Background

Testing executable forms of models is analogous to program testing and involves creation of test cases, the execution of the artifact using the test cases, and the analysis of test results to determine correctness of the tested behavior. The adequacy of the test cases is measured by criteria that define properties to be covered. The criteria are usually based on the building blocks of the software artifact that is being tested. For example, statements and branches are building blocks for code. Class attributes and associations are building blocks for object-oriented design models. Test criteria help in defining *test objectives* (goals) that need to be met while performing software testing. Cost considerations and available resources often determine the selection of one criterion over another. Test criteria can also be used to determine when testing should stop: testing can stop when tests that satisfy all the criteria have been carried out successfully.

The approach to defining test criteria for UML models is based on the category-partition testing [10] approach developed for code. The category partitioning approach utilizes a program's specification to (1) identify separately testable functional units, (2) categorize the inputs for each functional unit, and (3) partition the input categories into equivalence classes. Offutt and Irvine [8] show that the category partitioning technique is effective at detecting faults that involve implicit functions, inheritance, initialization and encapsulation when applied to OO software. In this work, a variant of category-partitioning is used to categorize and partition object configurations specified by UML Class Diagrams. Design-level test criteria determine the configurations that must be covered in an adequate design-level test.

The approach also uses a variant of the method sequence oriented approaches described in the OO code testing literature. Class testing techniques [11] provide for executing sequences of methods, and for varying the order of methods in the sequences. At the end of a sequence, the tester or the test environment verifies whether the resulting states of the objects involved are correct [5, 6, 15]. These method sequence oriented approaches are useful to consider adapting for those parts of the UML descriptions that deal with sequences of object states. The combining of category partitioning with the method sequence oriented technique results in an approach that involves more than just use of graph based criteria.

2.1. UML Modeling Concepts

Our UML model testing approach utilizes requirements and design models. A UML requirements model is used to develop the oracles for design model tests. A requirements model consists of a *conceptual model* (i.e., a *Requirements Class Diagram*) and a set of *use cases*. A conceptual model depicts the problem concepts and their relationships with each other. Each use case specifies a required behavior in terms of a pre-condition that states what must be true before execution if the behavior is to have the effect specified in the post-condition. The pre- and post-condition in a use case are defined in terms of concepts defined in the conceptual model of the requirements model.

Design models in the UML consist of a UML *design class diagram*, an *activity diagram*, and *interaction diagrams*. For details on the types of diagrams, refer to [14]. A design class diagram specifies the valid object structures (configurations) that can exist in an executing system. Classes can realize concepts in the requirements model, or represent objects introduced to support a particular implementation of the system.

An activity diagram is defined for each class and describes the behavior of class objects. The states and transitions in an activity diagram are of various types: action states, assignments, send actions, procedure calls and input signal transitions. A simple model of execution is used: for each object, incoming signals are queued and processed whenever the state machine moves into a new non-action state or after the action in an action state is performed. A UML model is thus interpreted as a collection of communicating state machines.

Interaction diagrams describe how objects collaborate in order to accomplish required behaviors. A *Collaboration Diagram* is an interaction diagram that depicts how objects interact to achieve a behavioral goal. *Sequence diagrams* are another form of interaction diagrams that can contain the same interaction information, but in a different format. In this paper, collaboration diagrams are used because they depict structure as well as interactions, allowing the development of criteria in terms of structures on which behaviors are performed. The structural information is implicit in sequence diagrams.

2.2. Related Work

Binder [3] describes generic test requirements derived from UML models and introduces test design patterns. These patterns focus on determining appropriate test strategies, faults that may be detected, test case development from the model and a test oracle. The test development can be done for different scopes in the implementation (e.g., method, class, class integration, subsystem and integration). This approach does not test UML models directly, but generates code test requirements from them.

Labiche and Briand [4] describe the TOTEM (Testing Object-orienTed systEms with the unified Modeling Language) system test methodology. System test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and the use of the Object Constraint Language across all these artifacts. The test requirements are then transformed into test cases, test oracles and test drivers using more detailed design information. This approach is meant for system testing whereas the proposed approach is targeted toward integration testing related to interactions and behaviors of objects. Moreover, this approach does not evaluate UML artifacts.

Offutt and Abdurazik [9] developed a technique for generating test cases for code (rather than designs) from a restricted form of UML state diagrams. The state diagrams used in their approach utilize only enabled transitions. Test cases are generated using only the change events as a basis. The authors identified four levels of testing based on transition coverage criteria and provided some empirical evidence of the effectiveness of their approach.

Although the work focused on the generation of code-level test cases, it is possible to adapt the approach for generating test cases for executable forms of state diagrams. A limitation of the work is that the approach applies only to restricted forms of state diagrams. Test case generation based on types of events other than change events (e.g., call events and signals) can also be used to increase the chances of uncovering faults related to the generation and handling of these events. The approach does not directly support testing of object interactions.

Abdurazik and Offutt [1] also developed test criteria based on collaboration diagrams for static and dynamic testing of implementation code. Building on this work, they proposed methods for statically checking code relative to a collaboration diagram using classifier roles, collaborating pairs, messages or stimuli and local variable definition-usage link pairs. A criterion for dynamic testing that involved message sequence paths was also proposed. For each collaboration diagram in the specification, there must be at least one test case that results in the execution of the code that implements the message sequence path of the collaboration diagram. At the time the paper was published, empirical evaluation of the criterion had yet to be performed. However, the utility of the criterion seems intuitive.

Both approaches proposed by Offutt and Abdurazik are for testing implementations using information from UML design models (state or collaboration diagrams). The goal of the proposed approach is to test the design models themselves and to use information from different types of diagrams (class and collaboration diagrams) during testing.

Scheetz et al. [13] developed an approach to generating system (black box) test cases for code from UML class diagrams. Test objectives are composed from building blocks. These are derived from defining choices in composing the initial state of objects and desired states of some or all of these objects after the test is executed. Test objectives can be aggregated by conjunction. The desired states for an object are determined by its attribute values and links to other objects.

3. Testing UML Designs

The collection of communicating state machines that models a system is referred to as a *System Model* in the remainder of this paper. In this work, UML diagrams are used to present views of system models. Testing a system model involves executing modeled behavior, starting from a specified configuration (object structure), using a sequence of signals that trigger modeled behaviors. During execution, the start configuration can change as a result of adding and deleting objects and links, and changing the values of object attributes.

We use the example of a web-based university course administration system (UCA) to illustrate the test method. The UCA system keeps track of three types of users: administrator, students and instructors. The system maintains a profile of the users' personal information and login information. Every user belongs to a specific department. Each department may offer several courses and the students may register for courses in various departments. Similarly, instructors may instruct courses in various departments. Only an administrator of the department is allowed to add a user to the system, create courses under the specific department, enroll students, and add assistants and instructors for the courses.

An instructor can edit and view the elements of the course, and examine details of students' records. A student may be assigned as an assistant of a course and can edit information related to the grades of the students and view the students' records. A student can view the courses for which he is currently registered, course information and personal records. A course consists of several elements (e.g., tests, quizzes, labs and tutorials).

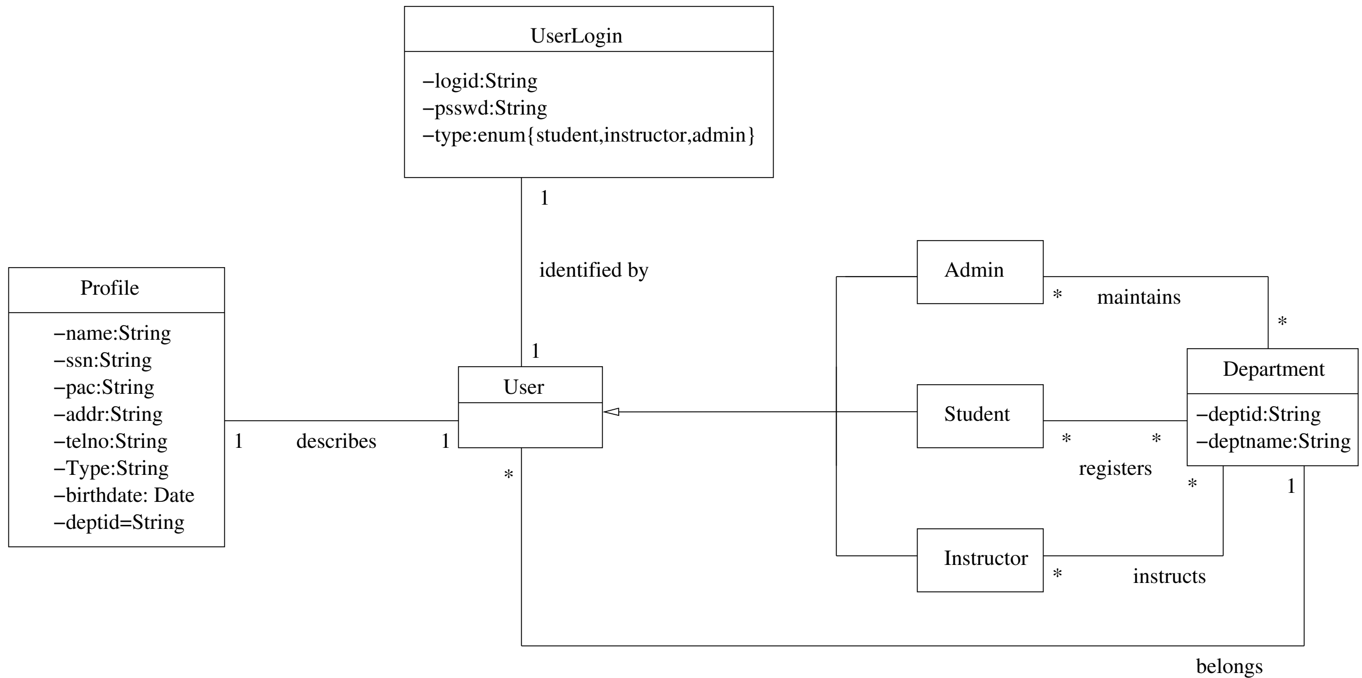

Figure 1. Partial Design Class Diagram

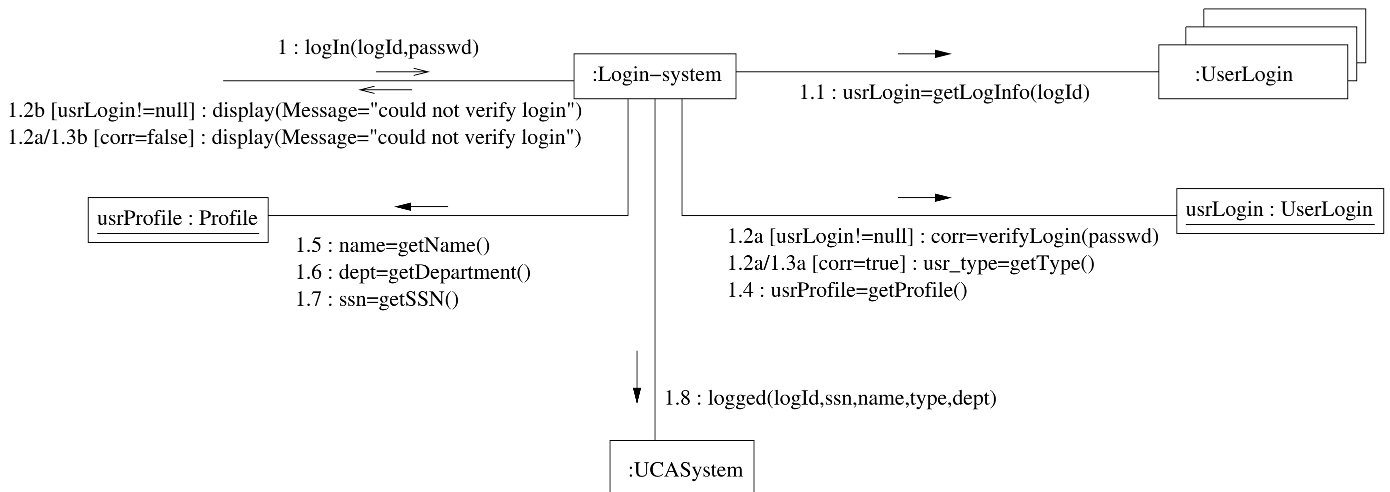

Figure 2. Collaboration Diagram for Log in

A user logs onto the system using a login screen and the system verifies the login information. If the login is verified, the system brings up an appropriate user window. For example, the instructor window displays links to personal information and courses that are taught by the instructor. The student window displays links to personal information, the courses registered and courses assisted by the student. The administrator window displays courses offered by a particular department, the corresponding instructors, assistants and students assigned to a course. The administrator may remove an instructor, assistant or student from a course and delete a course from the system.

We describe two use cases, *Log in* and *Add user*.

1. Use Case: Log in — User logs on to the UCA system.
Precondition: The user has a profile and login information

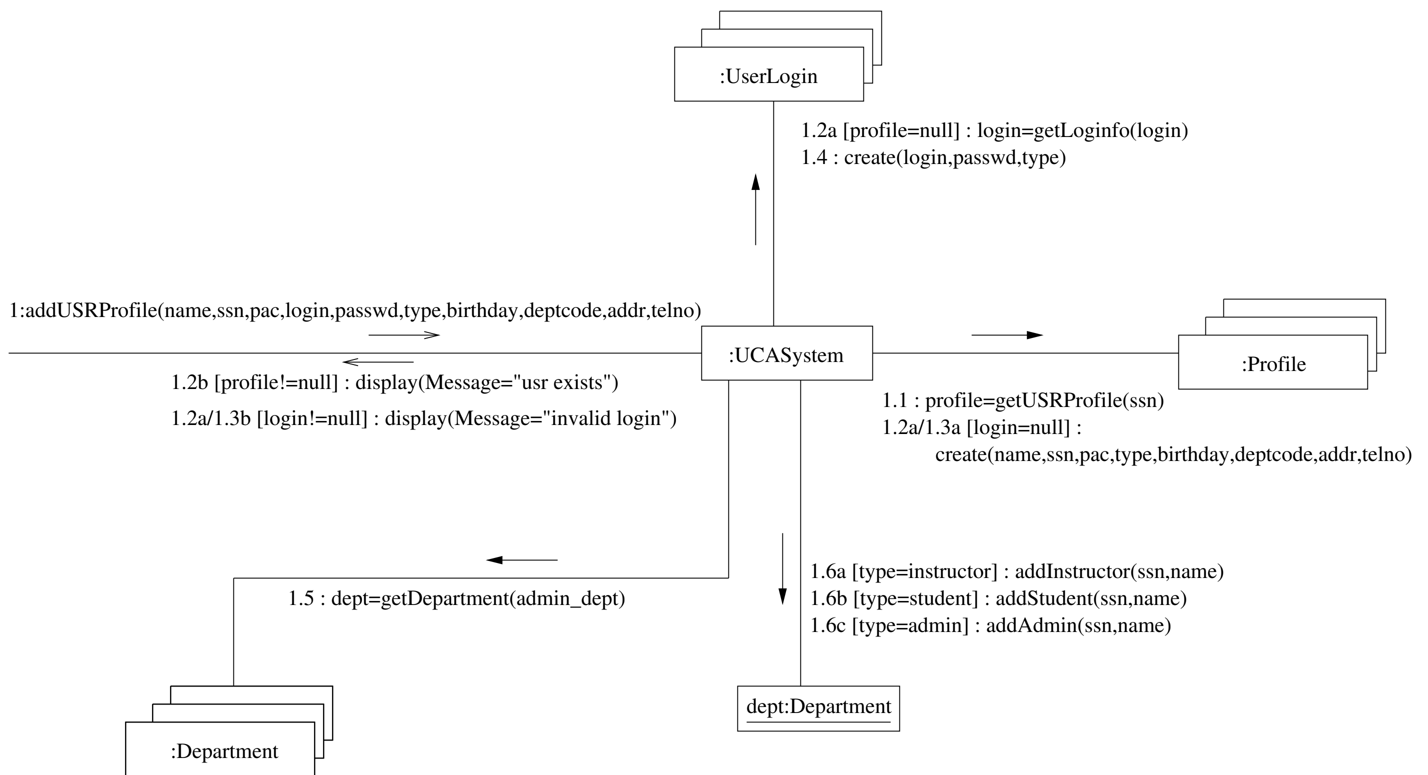

Figure 3. Collaboration Diagram for Add User

in the system.
Postcondition: The user successfully logs on to the system.
Main scenario:

1. The user invokes the login operation of the system.
2. The system prompts the user to enter the login information (login, password).
3. The user enters the required information.
4. The user successfully logs on to the system.

Alternate course of actions:

4a The system cannot verify the information entered by the user. The system notifies the user, and the user's attempt to log on to the system is unsuccessful.

2. Use Case: Add user — Administrator adds a new user.
Precondition: The administrator has successfully logged on to the UCA system.
Postcondition: The user's personal profile and login information are recorded in the system and the user is added to the particular department.
Main scenario:

1. The administrator invokes the add user operation of the system.
2. The system prompts the administrator to enter the user's personal information, (*name, ssn, pac, addr, telno, type, birthdate, deptid*) and desired login (*login, passwd*).
3. The administrator enters the required information.
4. If the system determines that the *ssn* and *login* are unique, the system creates a record of the user's personal and login information.

Alternate course of actions:

4a The system determines that there is another user with the entered *ssn* or *logid*. The system notifies the administrator.

Fig. 1 shows the design class diagram for the UCA system. Fig. 2 and Fig. 3 show the instance level collaboration diagrams *LogIn* and *addUsrProfile* that realize the *Log in* and *Add user* use cases respectively.

3.1. Defining Test Inputs

To test a design, one needs to create a test set, where a test set consists of several test cases. In our approach, a test case is a tuple that has the following form:

$$\ll sequence_of_signals, start_configuration, prefix \gg$$

A configuration is a structure of objects that satisfies the constraints expressed in the design class diagrams. A configuration includes (1) the class objects and the links that exist at a given time, and (2) the value of each attribute in each object in the configuration. The $start_configuration$ is the configuration on which the test is started. The $prefix$ is a sequence of signals that can be used to take the system from an initial configuration to the $start_configuration$. Once the system is in the chosen $start_configuration$, a $sequence_of_signals$ is applied to run the test.

A sample test case for the collaboration diagram shown in Figure 3 is given below:

- Sequence of signals:
  $logIn(logname123, passwd123)$,
  $logIn(logname123, passwd456)$,
  $logIn(logname456, passwd123)$,
- Start configuration: The start configuration consists of a set of users that includes a user with *logId* $logname123$ and *passwd* $passwd123$, but not a user with the *logId* as $logname456$.
- Prefix:
  addUSRProfile(name123, ssn123, pac123, logname123, passwd123, type123, birthday123, deptcode, addr, telno)

Execution of a test case will result in a trace of configurations and the sequence of signals generated as a result of the test input sequence. In order to determine whether a test is successful or not one needs an oracle. In this work, the pre- and post-conditions of use cases associated with the behaviors under test are used to determine success or failure of a test: If the start configuration of the test satisfies the pre-condition of the use case, then at the end of the test the final configuration must satisfy the post-condition of the use case for the test to pass.

3.2. Test Adequacy Criteria

The adequacy of tests executed on system models can be expressed in terms of the covered model elements. In this section, a set of test criteria based on coverage of elements in design class diagrams and interaction diagrams are presented.

3.2.1 Design Class Diagram (DCD) Criteria

DCD criteria determine the configurations that a behavioral test must cover in order to be termed *adequate*. For example, a criterion might specify the valid configurations (object structures) that must be created during execution of a system model. Failure to reach a specified valid configuration may be the result of an inadequate test set or an inconsistency in the system model which prevents the configuration from being reached.

DCD criteria can be based on the form of constraints present in the diagrams. In a DCD, constraints can be expressed as association-end multiplicities, generalizations and Object Constraint Language (OCL) statements. We define three DCD criteria:

1. *Association-end Multiplicity Criterion (AEM):* An association-end multiplicity specifies how many instances of a class at the opposite end of the association link can be associated with a single instance of a class at the association end. Given a test set T and a system model SM, T must cause each representative multiplicity-pair in SM to be created. Examples of association multiplicities that need to be covered in Fig. 1 are as follows:

   - Between *User* and *Profile*: {(0, 0), (0, 1), (1, 0), (1, 1), (1, 2), (2, 1)}. The first number in each pair denotes how many *User* instances are associated with a *Profile* instance. The second number denotes how many *Profile* instances are associated with one *User* instance. The *AEM* values (0, 1), (1, 0), (1, 2), (2, 1) and (1, 2) are all outside boundary conditions.
   - Between *User* and *Department*: {(0, 0), (0, 1), (1, 0), (1, 1), (2, 1), (max, 0), (max, 1), (max+1, 0), (max+1, 1), (0, 2), (1, 2), (max+1, 2)}. The *AEM* values (1, 0), (max, 0), (max+1, 0), (max+1, 1), (1, 2) and (max+1, 2) are all outside boundary conditions.

   The AEM criterion ensures that testers run tests to exercise configurations containing boundary and non-boundary occurrences of links between objects. A link is an instance of an association.

2. *Generalization Criterion (GN):* The generalization criterion defines the *representative* set of specialization types that must be created from a DCD's superclasses during a system model test. Given a test set T and a system model SM, T must cause every specialization defined in a generalization relationship to be created. Examples of generalization-specialization relationships from Fig. 1 are as follows:

   - *User-Admin*
   - *User-Student*
   - *User-Instructor*

   Since the *User* has associations with *Profile*, *UserLogin* and *Department*, the following associations need to be created to satisfy the GN criterion:

   - *Admin* and *Profile*
   - *Student* and *Profile*
   - *Instructor* and *Profile*
   - *Admin* and *UserLogin*

- *Student* and *UserLogin*
- *Instructor* and *UserLogin*
- *Admin* and *Department*
- *Student* and *Department*
- *Instructor* and *Department*

Tests that satisfy the GN criterion are likely to reveal faults that can arise from violation of the substitutability principle which states that an instance of a subclass can be used anywhere an instance of its superclass is expected. Tests that satisfy the GN criterion can uncover substitutability violations by testing behaviors in which superclass objects are expected using specializations of the superclass instead of superclass objects.

3. *Class Attribute Criterion (CA):* Given a test set T, a system model SM, and a class C, T must cause a set of representative attribute value combinations in each instance of class C to be created.

   Attribute values may restrict the behavior of an object. For example, particular attribute values may restrict how an object responds to signals. Thus, the *value* space of attributes provide yet another opportunity to develop test criteria. The value space of an attribute can be restricted by OCL constraints.

   The critical part of the criterion is to define the combinations of representative attribute values. This is done in three steps:

   (a) Use category-partitioning to create representative value sets for each attribute.
   (b) Take the Cartesian product of each value set to create an aggregate set of representative attribute value tuples for each class.
   (c) Identify valid and invalid aggregated sets.

Two of the criteria (AEM and CA) are expressed in terms of representative values. In order to establish the set of representative values, a form of category-partition testing [10] adapted to UML diagrams is used. Using this method, the value domain is partitioned into equivalence classes, and one value from each class is selected for the set of representative values. The partitions can be determined using domain knowledge-based partitioning, or default partitioning (minimum, maximum and non-boundary values).

In Fig. 1, the class *Profile* has an attribute called *birthdate*. If there was a use case in which the age played a role in determining different behaviors, testers could partition the age into different ranges. Thus, test inputs would include different dates of birth according to these partitions. Also, the configuration of the test system would include *Profiles* containing these different dates of birth.

3.2.2 Interaction Diagram Criteria

We define and use four types of interaction diagram criteria.

1. *Condition Coverage (Cond) Criterion:* Certain messages in a collaboration diagram may be executed only under certain conditions. An adequate test set should test all possible branches based on a condition. The $Cond$ criterion applies only to the conditions shown on collaboration diagrams. A test set that satisfies the $Cond$ criterion must include test cases that make this condition true and false. In Fig. 3, the *[useLogin!=Null]* condition can take the following values depending on the parameters passed to the *logIn* operation:
   - *usrLogin!=null:* {*validLogName, invalid-passwd*}
   - *srLogin==null:* {*invalidLogName, anyPasswd*}

2. *Full predicate coverage (FP) criterion:* A condition may consist of more than one clause connected by boolean operators (e.g., AND, OR). An adequate test set should ensure that each clause in every condition take the values of `TRUE` and `FALSE` while all other clauses in the condition have values such that the value of the condition is the same as the clause being tested. This ensures that each clause in a condition is separately tested. In our example, all the conditions have one predicate each.

3. *Each Message on Link (EML) Criterion:* According to this criterion, signals for each message on a link connecting two objects in a collaboration diagram should occur at least once in an adequate test. This criterion ensures that all messages between two objects occur during tests. For example, in Fig. 2, the following messages exist on the link between the objects *Login-system* and *usrProfile:Profile* .
   - 1.5 : $getName()$
   - 1.6 : $getDepartment()$
   - 1.7 : $getSSN()$

4. *All Message Path (AMP) Criteria:* A message path is a sequence of messages. The All Message Path (AMP) criterion ensures that all message paths in a collaboration diagram are exercised. The following paths (shown only by the numerical labels for brevity) exist in Fig. 2:

   (a) path [1, 1.1, 1.2b]
   (b) path [1,1.1, 1.2a, 1.3b]
   (c) path [1, 1.1, 1.2a, 1.3a, 1.4, 1.5, 1.6, 1.7, 1.8]

The DCD and collaboration diagram based test criteria can be used to derive test objectives. For example, an AMP

criterion can be used to define a test objective that stipulates the specific paths to be exercised during tests. The $Cond$ criterion can be used to derive test objectives that stipulate values for specific condition.

3.3. Test Method Using UML Criteria

Testing methods for UML designs are likely to differ depending on the testing criteria used. To illustrate the basic principle and highlight some of the issues that need to be addressed, it is assumed that the class diagram criteria given in this paper need to be met, as well as the AMP (all message paths) criterion for collaboration diagrams.

The test objectives derived from class diagram test criteria define a set of target configurations S. This set can be partitioned into those target configurations that represent initial system configurations (S_0) and those that need a prefix executed (S_1). The AMP criterion leads to test objectives in the form of a set of paths $P = \{P_1, ..., P_k\}$ in the design's collaboration diagram. These paths represent a sequence of messages. There is always at least one collaboration diagram that can apply its paths from an initial system configuration. Let this set be PI and the set that requires a prefix PP. The candidate testing process is as follows [2]:

1. Determine target configurations S_0 and S_1 from test objectives for class diagrams.
2. Determine paths PP and PI as sequences of messages from collaboration diagrams.
3. Apply $p_i \in PI, i = 1, ...k_{PI}$ to $s_j \in S_0$ if the precondition for p_i is met in s_j. Note that paths may be applied to more than one $s_j \in S_0$.
4. Determine whether any intermediate configurations during the execution of one of these test cases meet preconditions for paths $pp_i \in PP$. If so, use the initial state and the stimuli that reach that state as prefix for applying these paths pp_i. This reduces the uncovered paths to a set PP'.
5. Apply and measure coverage. Let S_c be the set of covered target configurations. Any uncovered target configurations must be in S_1. Uncovered states are $S_1' = S_1 \setminus \{s \in S_1 \mid s \in S_c\}$.
6. Determine prefixes for the configurations that meet preconditions for uncovered paths in PP'.
7. Measure coverage and reduce S_1' by any configurations covered in the previous step.
8. Determine prefixes and execute them for the remaining states in S_1.

Given that paths through collaboration diagrams are determined via the ordering of messages (numbers), determining paths (PI) is trivial. In addition, one would naturally assume that designers evaluate their design against requirements. These are commonly reflected in the use cases. Applying use cases would result in at least partial test coverage. Use cases can also be used to determine necessary prefixes to paths (in PP) because use cases have pre- and post-conditions, and thus can be ordered or concatenated if the post-condition of one meets the precondition of the other.

It must be noted that this candidate test process is neither minimal, nor immediately automatable. Part of the reason for this is that it implies determining reachability, an undecidable problem. However, this does not mean that a human is unable to execute this process for a particular design.

4. Case Study

In this section, we present the results of a case study performed on the design model of the university course registration system described in Section 3. The objective of the study was to investige the number of tests required to satisfy the test criteria. The number of test cases has a bearing on the cost of generating and executing test cases. One way to determine the worst case complexity of the number of test cases is to consider the number of partitions, messages and paths, i.e., the size of the different coverage domains. Although it appears that the number of test cases increases exponentially with increasing complexity in the design models, in reality, we observed that several tests that covered elements in one coverage domain, also covered elements in another coverage domain. Thus, the actual number of test cases required to test the model was much less that the number estimated from the size of the coverage domains.

4.1. Test Cases for the $LogIn$ Operation

We first developed test cases to satisfy test objectives based on the design class diagram (DCD) criteria. We identified which elements from the collaboration diagram coverage domains were exercised by these test cases. Then we improved upon the test cases to ensure coverage of the collaboration diagram criteria. We performed this exercise first on the collaboration diagram for the *logIn* operation (see Fig. 2). Table 1 shows the parameters that are passed in the *LogIn* signal (column 2), the type of *User* involved in the login process, and the DCD coverage elements exercised during the test. The *logIn* operation requires the *logId* and *passwd* to be passed as parameters. The *logId* and *passwd* can both be independently valid or invalid. The type of *User* involved in the operation affects the generalization-specialization relationship covered by the test case. The entry NA in the *User* column denotes values that do not matter. The term *satisfied/not-satisfied* in the *Coverage element* column indicates whether the use case condition is satisfied or not. The use case condition states that a user's login information *logId* and *passwd* should match the input

values. GN indicates which generalization specialization relationship was covered. AEM denotes the association end multiplicities between *User* and *UserLogin* that is being considered.

Table 1. Test Cases Derived for DCD Criteria

	Test case parameters	User	Coverage element
1	invalidLogName1,passwd	NA	not-satisfied, AEM(0,0)
2	invalidLogName2,passwd	NA	not-satisfied, AEM(0,0)
3	validLogName3,invalidpasswd	Instructor	not-satisfied, AEM(1,1), GN(Instructor)
4	validLogName3,validpasswd	Instructor	satisfied, AEM(1,1), GN(Instructor)
5	validLogName4,invalidpasswd	Admin	not-satisfied, AEM(1,1), GN(Admin)
6	validLogName4,validpasswd	Admin	satisfied, AEM(1,1), GN(Admin)
7	validLogName5,invalidpasswd	Student	not-satisfied, AEM(1,1), GN(Student)
8	validLogName5,validpasswd	Student	Constr(true), satisfied, GN(Student)

Test cases 1 and 2 test whether a user can log on to the system using an invalid *logId*. The *passwd* parameter can take any value. Test cases 3 and 4 test whether an instructor can log in with a valid *logId* using an invalid and valid *passwd*, respectively. Test cases 5, 6, 7 and 8 repeat the same tests for a student and an admin. The testing of valid users requires prefix stimuli to ensure that the users were added to the system (using *addUSRProfile*) before they can log in. We needed 8 test cases to achieve coverage of the DCD criteria.

Tables 2, 3, and 4 illustrate the coverage of conditions, messages on each link, and message paths in the collaboration diagram. For each coverage element in column 2, we show the test cases that exercised it in column 3. From these tables we see that the test cases together are adequate with respect to the $Cond$, EML and AMP criteria. Since the conditions contained one predicate each, the test cases are also adequate with respect to the FP criterion.

Table 2. $Cond$ Coverage

	Condition covered	Test Cases
1.	usrLogin!=null	3, 4, 5, 6, 7, 8
2.	usrLogin==null	1, 2
3.	corr==True	4, 6, 8
4.	corr==False	3, 5, 7

Table 3. EML Coverage

Link	Message	Test cases
1	1:login	1-8
1	1.2b:display	1, 2
1	1.2a.1b:display	3, 5, 7
2	1.1:getLogin	1-8
3	1.2a:verifyLogin	3, 4, 5, 6, 7, 8
3	1.3a:getType	4, 6, 8
3	1.4a	4, 6, 8
4	1.5:getName	4, 6, 8
4	1.6:getDept	4, 6, 8
4	1.7:getSSN	4, 6, 8
4	1.8:logged	4, 6, 8

Table 4. AMP Coverage

Message paths	Test cases
[1, 1.1, 1.2b]	1, 2
[1, 1.1, 1.2a, 1.3b]	3, 5, 7
[1, 1.1, 1.2a, 1.3a, 1.4	4, 6, 8

4.2. Test Cases for the $addUSRProfile$ Operation

We also developed test cases for the $addUSRProfile$ operation. The tests assume that the administrator is logged in. The parameters are the *name*, *ssn*, *pac*, *login*, *passwd*, *type*, *birthdate*, *deptcode*, *addr* and *telno*. Table 5 shows the test cases developed to satisfy the DCD criteria.

Column 2 of the table shows the key parameters of the signal $addUSRProfile$: *ssn*, *login*, *type* and *deptcode*. Since the attributes *name*, *pac*, *passwd*, *birthdate*, *addr* and *telno* are not involved in any constraint for the $addUSRProfile$ operation, they are irrelevant from the testing point of view and have been omitted from the table. Column 3 indicates the coverage elements that are covered by the test cases. *PreCond1* indicates that the following OCL constraint - every user has a unique *ssn*. *PreCond2* indicates that the following OCL constraint - every user has a unique *logId*. *AEM* denotes the association end multiplicities between *User* and *Department*. *AEM(m, n)* indicates that m *Users* are associated with one *Department* and n *Departments* are associated with one *User*. *AEM(1,0)* indicates that a user is being added to a non-existing de-

partment. *AEM(1,1)* indicates that a user is being added to an existing department. *AEM(1,2)* indicates that a user is being added to a second department. The association end multiplicities between *User* and *Profile*, and *User* and *UserLogin* are not shown in the table. The coverage of *preCond1* and *preCond2* ensures that both association end multiplicities are covered with the values *AEM(0,0)* and *AEM(1,1)*. *GN(Student)* indicates the type of specialization class that is involved in the test case.

Table 5. Test Cases Derived for DCD Criteria

	Test case parameters	Coverage element
1	ssn1, logid1, student, invaliddeptcode	preCond1(true), preCond2(true), AEM(1,0), GN(Student)
2	ssn1, logid1, student, validdeptcode1	preCond1(true), preCond2(true), AEM(1,1), GN(Student)
3	ssn1, logid1, student, validdeptcode1	preCond1(false), preCond2(false), AEM(1,1), GN(Student)
4	ssn1, logid1, student, validdeptcode2	preCond1(false), preCond2(false), AEM(1,2), GN(Student)
5	ssn2, logid1, student, validdeptcode1	preCond1(true), preCond2(false), AEM(2,1), GN(Student)
6	ssn2, logid1, student, validdeptcode2	preCond2(true), preCond2(false), AEM(1,1), GN(Student)
7	ssn1, logid2, student, validdeptcode1	preCond1(false), preCond2(true), AEM(2,1), GN(Student)
8	ssn1, logid2, student, validdeptcode2	preCond1(false), preCond2(true), AEM(1,1) ,GN(Student)
9	ssn2, logid2, student, validdeptcode1	preCond1(true), preCond2(true), AEM(2,1), GN(Student)
10	ssn2, logid2, student, validdeptcode1	preCond1(false), preCond2(false), AEM(2,1), GN(Student)
11	ssn2, logid2, student, validdeptcode2	preCond1(false), preCond2(false), AEM(1,2), GN(Student)

The first test case adds a student to a non-existing department. Test case 2 adds a student to a valid department. Test case 3 adds a previously added student to the same department, something that is prohibited by the *AEM* between *User* and *Department*. Test case 4 adds a previously added student to a different department. Test cases 5 through 8 add a student using a combination of an existent or non-existent *ssn*, and an existent or non-existent *logId*. Test case 9 uses a non-existent *ssn* and a non-existent *logId* to satisfy the preCond1 and preCond2. Test cases 10 adds a user with the same values as in the previous test case. Finally, test case 11 adds a user with the same *ssn* and *logId* values as in test case 9 but to a different department.

Additional test cases are shown in Table 6 and are described as follows: The test cases 12-22 and 23-33 repeat test cases 1-11 using the same values for *ssn*, *logId* and *deptcode* but vary the type for *Instructor* and *Admin*. Test cases 34-44 use new set of *ssn* and *logId* values for adding *Instructors* and follows the same sequence as 1-11. Test cases 45-55 and 56-66 use the same *ssn* and *logId* values as in 34-44 but vary the type field for *Admin* and *Instructor*. Test cases 67-77 use new *ssn* and *logId* values for adding *Admins*. Finally, test cases 88-98 and 99-109 use the same *ssn* and *logId* values as 67-77 but vary the type field for *Instructor* and *Student*. The above test cases consider all combinations by first adding students and adding instructors and admins using the same *ssn* and *logId* values as students. The procedure is repeated by adding instructors and admins.

Table 6. More Test Cases for Other User Types

Number	Test cases
12-22	Repeat test cases 1-11 but with type=instructor
23-33	Repeat test cases 1-11 but with type=admin
34-44	Test cases similar to 1-11 but with new set of ssn and logId values for type=Instructor
45-55	Repeat test cases 34-44 but with type=admin
56-66	Repeat test cases 34-44 but with type=student
67-77	Test cases similar to 1-11 but with new set of ssn and logId values for type=Admin
88-98	Repeat test cases 67-77 but with type=instructor
99-109	Repeat test cases 67-77 but with type=student

Tables 7, 8, and 9 illustrate the coverage of conditions, messages on each link, and message paths in the collaboration diagram. For each coverage element in column 2, we show the test cases that exercised it in column 3. From these tables we see that the test cases together are adequate with respect to the $Cond$, EML and AMP criteria. Since the conditions contained one predicate each, the test cases are also adequate with respect to the FP criterion. The lower bound on the number of test cases required to satisfy the DCD criteria is 109. During the testing process, we found a fault in the collaboration diagram. Test cases 1, 34 and 67 create a configuration where a *UserLogin* and *Profile* are created for an invalid department, even though this is disallowed by the design class diagram.

While this case study showed that test cases cover multiple coverage items for both class diagram and collaboration diagram notations, it is important to notice that it is also limited in its generalizability. Further empirical work is necessary to investigate the degree of “multiple dipping”

Table 7. $Cond$ **Coverage**

	Condition	Test cases
1.	profile!=null	3, 4, 7, 8, 10, 11,12-22, 23-33, 36, 37, 40, 41, 43, 44, 45-55, 56-66, 69, 70, 73, 74, 76, 77, 88-98, 99-109
2.	profile==null	1, 2, 5, 6, 9, 34, 35, 38, 39, 42, 67, 68, 71, 72, 75
3.	login!=null	5, 6, 38, 39, 71, 72
4.	login==null	1, 2, 9, 34, 35, 42, 67, 68, 75
5.	type==admin	68, 75
6.	type==student	2, 9
7.	type==instructor	35, 42

Table 8. EML **Coverage**

Link	Message	Test cases
1	1:addUSRProfile	test cases 1-109
1	1.2b:display	3, 4, 7, 8, 10, 11,12-22, 23-33, 36, 37, 40, 41, 43, 44, 45-55, 56-66, 69, 70, 73, 74, 76, 77, 88-98, 99-109
1	1.2a/1.3b:display	5, 6, 38, 39, 71, 72
2	1.2a:getLoginfo	1, 2, 5, 6, 9, 34, 35, 38, 39, 42, 67, 68, 71, 72, 75
2	1.3a/1.4:create	1, 2, 9, 34, 35, 42, 67, 68, 75
3	1.1:getUSRProfile	test cases 1-109
3	1.2/1.3a:create	1, 2, 9, 34, 35, 42, 67, 68, 75
4	1.5:getDepartment	1, 2, 9, 34, 35, 42, 67, 68, 75
5	1.6a:addInstruictor	35, 42
5	1.6b:addStudent	2, 9
5	1.6c:addAdmin	68, 75

Table 9. AMP **Coverage**

Message paths	Test Cases
[1, 1.1, 1.2b]	3, 4, 7, 8, 10, 11, 12-22, 23-33, 36, 37, 40, 41, 43, 44, 45-55, 56-66, 69, 70, 73, 74, 76, 77, 88-98, 99-109
[1, 1.1, 1.2a, 1.3b]	5, 6, 38, 39, 71, 72
[1, 1.1, 1.2a, 1.3a, 1.4, 1.5, 1.6a]	35, 42
[1, 1.1, 1.2a, 1.3a, 1.4, 1.5, 1.6b]	2, 9
[1, 1.1, 1.2a, 1.3a, 1.4, 1.5, 1.6c]	68, 75

that can be expected for test cases. It is also important to note that theoretical limits on worst case behavior will be high. In general, not all design models will have the same degree of detail as ours. For example, use cases and design class diagrams may lack OCL constraints. In such cases, tests created using DCD criteria may need to be enhanced using interaction diagram criteria.

There is plenty of evidence from other testing techniques that the number of test cases grows linearly, rather than quadratically or exponentially. For example, the theoretical limit on the number of test cases to satisfy dataflow criteria is quadratic in the number of two-way decisions. However, Weyuker [16] showed that in most cases, the number of test cases grows linearly with the number of two-way decisions in a program. There is an expectation of analogous behavior because a single test covers items required by both class diagram and collaboration diagrams. Usually, more than one coverage item of each type will be covered. Indeed, test cases can be constructed to cover the largest possible number of items. Thus, while the number of coverage items may well be the product of numbers of partitions, this has little to do with the number of test cases required.

5. Conclusions and Future Work

We described a test process for executable forms of UML design models. The process incorporated the use of test adequacy criteria based on elements in class diagrams and collaboration diagrams. We illustrated the test case generation process for design models developed for an application. We demonstrated that test cases were able to cover multiple coverage elements. Even though the worst case complexity of the number of test cases based on the number of partitions appeared to be exponential, in reality, the number test cases appeared to grow at a much smaller rate.

We are currently evaluating the effectiveness of the generated test cases in detecting faults in design models. We are also working on automatic execution of test cases.

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. B. France, S. Ghosh, and G. Craig. Test Adequacy Criteria for UML Design Models. *to appear in Journal of Software Testing, Verification and Reliability*, 2003.

[3] R. V. Binder. *Testing Object-Oriented Systems Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, Reading, Massachusetts, October 1999.

[4] L. Briand and Y. Labiche. A UML-based approach to system testing. In *4th International Conference on the UML*, pages 194–208, Oct. 2001.

[5] P. G. Frankl and R.-K. Doong. Case studies on testing object-oriented programs. In *4th Symposium on Testing, Analysis, and Verification*, pages 165–177, Oct. 1991.

[6] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *14th Intl. Conference on Software Engineering*, pages 68–80, May 1992.

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

[8] A. J. Offutt and A. Irvine. "Testing Object-Oriented Software Using the Category-Partition Method". In *Proceedings of the 17th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA) 1995*, pages 293–304, Santa Barbara, California, August 1995.

[9] J. Offutt and A. Abdurazik. Generating test from UML specifications. In *2nd International Conference on the UML*, pages 416–429, Oct. 1999.

[10] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[11] D. E. Perry and G. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-oriented Programming*, pages 13–19, Jan. 1990.

[12] 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.

[13] 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.

[14] The Object Management Group. OMG Unified Modeling Language Specification. Version 1.3, OMG, 1999.

[15] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Conf. on Software Maintenance*, pages 302–311, Sept. 1993.

[16] E. J. Weyuker. The Cost of Dataflow testing: an Empirical Study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.