

Test adequacy criteria for UML design models

Anneliese Andrews, Robert France, Sudipto Ghosh^{*,†}
and Gerald Craig

*Computer Science Department, Colorado State University,
Fort Collins, CO 80523, U.S.A.*



SUMMARY

Systematic design testing, in which executable models of behaviours are tested using inputs that exercise scenarios, can help reveal flaws in designs before they are implemented in code. In this paper a technique for testing executable forms of UML (Unified Modelling Language) models is described and test adequacy criteria based on UML model elements are proposed. The criteria can be used to define test objectives for UML designs. The UML design test criteria are based on the same premise underlying code test criteria: coverage of relevant building blocks of models is highly likely to uncover faults. The test adequacy criteria proposed in this paper are based on building blocks for UML class 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. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: design reviews; software testing; test adequacy criteria; UML; class diagram; collaboration diagram; category partitioning

1. INTRODUCTION

The Unified Modelling Language (UML) [1] is an Object Management Group (OMG) object-oriented (OO) modelling language standard that is gaining widespread use in the software development industry. Modelling a large, complex system (e.g., a telecommunication system) can result in a system model that consists of a variety of diagrams presenting different views of the model. Design reviews in which the reviewers trace and relate concepts across the diagrams can be tedious because of the amount of information reviewers need to track. The use of UML models that have informal semantics can also make it difficult to uncover design faults in UML models.

*Correspondence to: Dr Sudipto Ghosh, Computer Science Department, Colorado State University, Fort Collins, CO 80523, U.S.A.

†E-mail: ghosh@cs.colostate.edu

Contract/grant sponsor: National Science Foundation Award; contract/grant number: #CCR-0203285



Static and dynamic analysis tools that utilize well-defined semantics for UML models can enhance a reviewer's ability to uncover faults in the models. Rigorous static analysis tools that can significantly enhance design reviews include state exploration tools (e.g., model checkers [2–4]), Object Constraint Language (OCL) [5] type-checking tools, and tools that utilize well-defined semantic relationships across the views of a UML model (e.g., Rational Rose) to check consistency of information across the views. Semantics that support the creation of executable UML models pave the way for dynamic analyses of models. Dynamic analysis of UML models is concerned with testing modelled behaviours by executing models using appropriate forms of test inputs. In this paper, a technique that supports testing of executable forms of UML models is presented.

Systematic testing techniques for UML designs need to address at least the following concerns:

- *Development of a semantics that supports creation of executable models.* A systematic model testing technique requires a semantics that supports execution of the models. Currently, the semantics of the UML is informally described in the OMG standard document [1]. Executable forms of the UML are beginning to emerge (e.g., see work on Executable UML [6], and the UML Virtual Machine [7]). These works indicate that one can associate formal semantics to UML behavioural elements that support the execution of UML models.
- *Development of criteria that determine the adequacy of tests.* The effectiveness of a test is based on how well the tests cover and exercise the modelled behaviours. In code testing, criteria based on coverage of the building blocks of programs can be used to determine the adequacy of tests. Similar criteria for UML models can be used to guide the selection of test cases.

In this paper a systematic technique for testing executable forms of UML design models is described. The technique utilizes test criteria expressed in terms of UML model element coverage. The remainder of the paper is organized as follows. The concepts underlying the testing technique, the elements of the executable forms of UML models and related work on UML testing are presented in Section 2. In Section 3, an overview of the technique for testing UML designs is given. Test criteria based on elements in UML models are described in Section 4. A systematic testing approach using the proposed criteria is described in Section 5. An example illustrating the creation of test cases is given in Section 6. A preliminary evaluation of the criteria in terms of the types of faults they target is described in Section 7. Conclusions and directions for future work are presented in Section 8. For readers less familiar with UML terminology, definitions of common UML terms based on the most recent standard definition [8] are quoted in Appendix A.

2. BACKGROUND

Testing executable forms of models is analogous to program testing. In general, testing an executable software artifact (program or executable design model) involves (1) the creation of test cases, (2) the execution of the artifact using the test cases, and (3) the analysis of test results to determine correctness of the tested behaviour. In this section, the basic testing concepts that underly this work, and the UML elements used to build testable UML designs are described. The section concludes with an overview of related work on UML testing.



2.1. Software testing concepts

A *test case* is a sequence of inputs that determines the behaviours that will be tested (e.g., a sequence of program input data or a sequence of events for a state machine design model), along with the expected behaviour. A test case is successful if the observed behaviour matches the expected behaviour; otherwise, the test fails. Success can be determined with the help of an *oracle* that compares the observed behaviour with the expected (correct) behaviour. A *test set* is a set of one or more test cases.

A *test adequacy criterion* is a predicate that defines properties that must be covered if the test is to be considered adequate with respect to the criterion [9]. Tests that are adequate with respect to a criterion cover all the elements in the domain determined by the criterion. Test criteria help in defining *test objectives* or goals that are to be achieved 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 [11] 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 [12] 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 [13–15]. 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.

2.2. UML modelling concepts

The UML model testing approach described in this paper 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 to each other. Each use case specifies a required behaviour in terms of a pre-condition that states what must be true before execution if the behaviour 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.

The design model consists of (1) a *design class diagram* (DCD) that specifies the valid object structures (configurations) that can exist in an executing system, (2) an *activity diagram* for each class

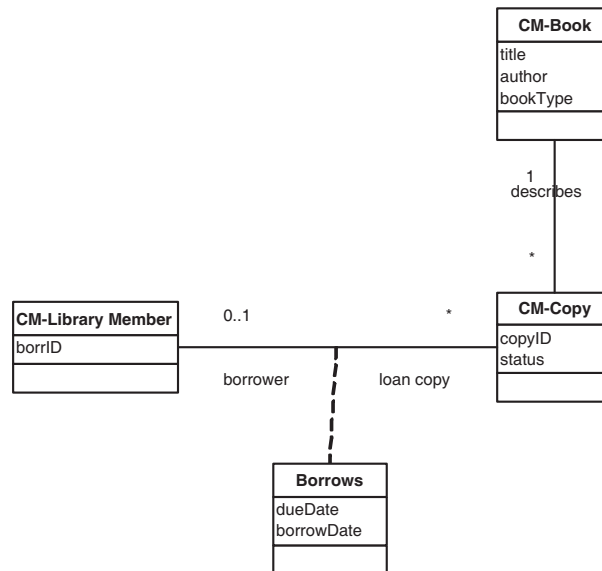


Figure 1. Conceptual model for a simple library system.

that describes the behaviour of class objects, and (3) *interaction diagrams* that model how objects collaborate in order to accomplish required behaviours.

2.2.1. Requirements model artifacts

The conceptual model used in this approach consists of (1) classes that represent problem concepts, (2) associations that model semantic relationships between problem concepts, and (3) specialization relationships between problem concepts. Class properties are represented by attributes and associations. Conceptual model classes do not have operations. An example of a conceptual model for a simple *library system* is shown in Figure 1. A library member (instance of *CM-Library Member*) can have zero or more copies (instances of *CM-Copy*) checked out at any time, while a copy can be checked out by at most one library member. The borrow relationship between a library member and a copy is associated with information about the due date and borrow date. Each copy must be associated with a book (instance of *CM-Book*) that contains information about the book's author, title, and also indicates the book type (for this simple example, a book can either be a general book that can be checked out or a reference book that cannot be checked out).

Use cases specify required behaviours. Figure 2 contains two use cases for the library system, one specifying the behaviour of a successful copy checkout, the other specifying the behaviour of an unsuccessful checkout.

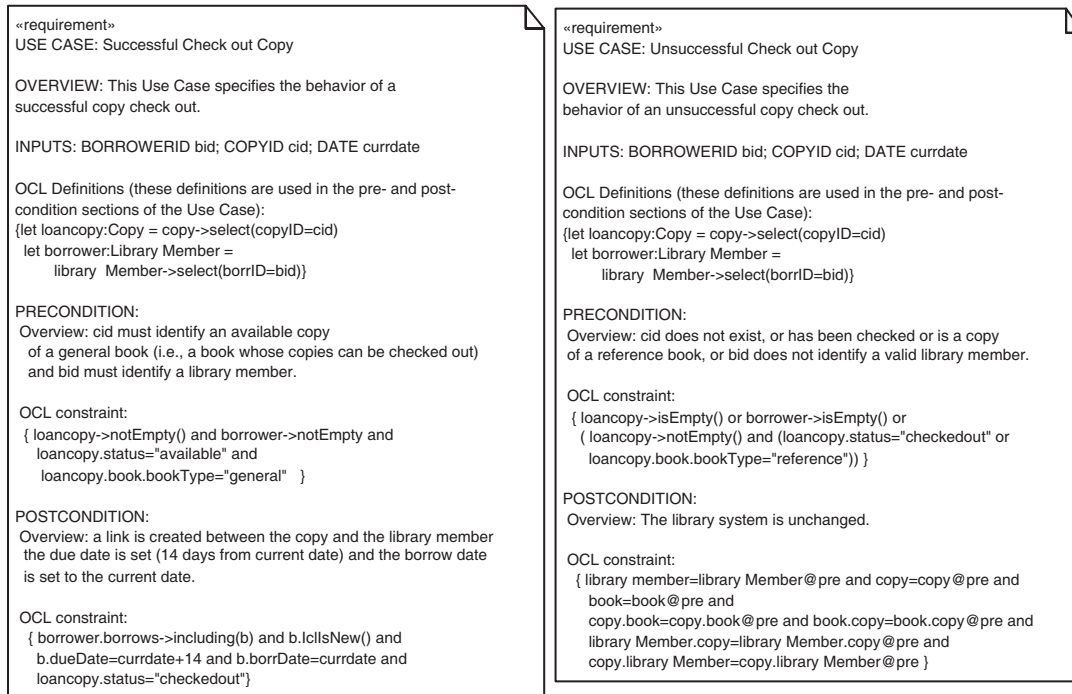


Figure 2. Use cases for checking out a copy.

2.2.2. Design model artifacts

A system is modelled as a collection of state machines that communicate asynchronously. Three diagrams are used to describe different views of a system model: a DCD depicts the design classes and their relationships, an activity diagram[‡] describes the behaviour of class objects, and an interaction diagram describes the intra-object communications. A design model consists of a DCD, an activity diagram for each class in the DCD, and a set of interaction diagrams that present the object interaction view of the system.

An example of a DCD that shows part of a design for the simple library system is shown in Figure 3. The class *Library Member* is intended to realize the *CM-Library Member* concept (see Figure 1), *Copy* is intended to realize the *CM-Copy* concept, while the class *Borrows* is intended to realize the conceptual model association class *Borrows*. The other classes in the DCD represent *design objects*, that is, objects introduced to support a particular implementation of the system. In this case, a *System*

[‡]An activity diagram is a state machine in which actions take place in the states.

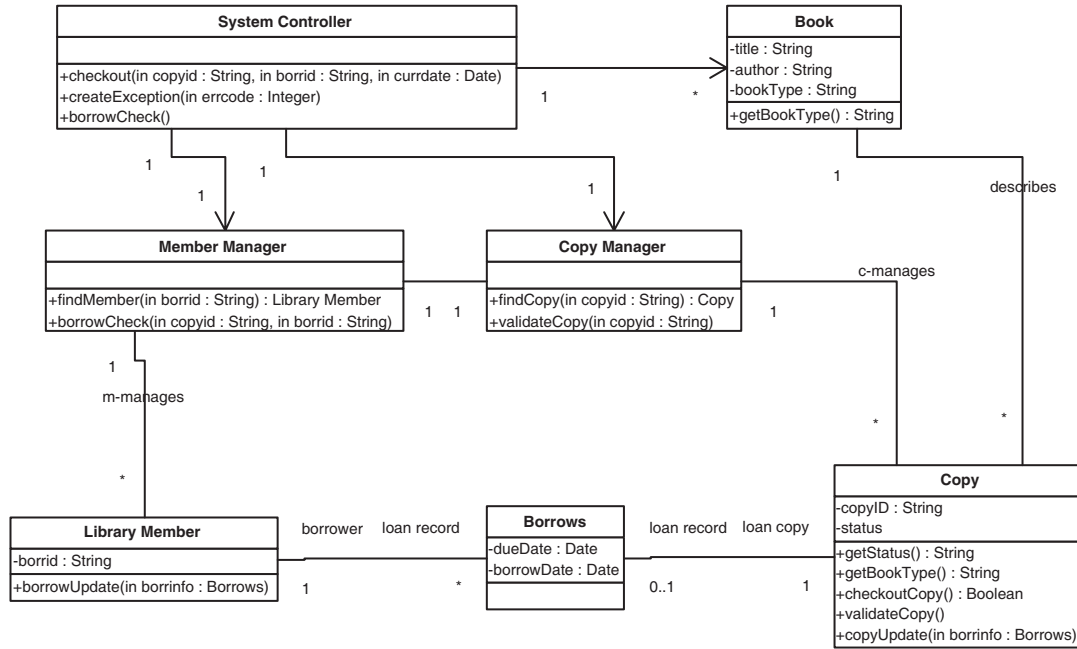


Figure 3. Partial DCD for the library system.

Controller object accepts requests for library services and directs them to appropriate manager objects for handling. The two manager classes shown in Figure 3 are *Member Manager* (performs member management functions such as checking out a copy for a member) and *Copy Manager* (performs copy management functions such as checking the availability of a copy).

Each class in a DCD is associated with an activity diagram that describes the behaviour of objects in the class. Partial activity diagrams for the *System Controller* and the *Member Manager* classes are shown in Figures 4 and 5, respectively.

An activity diagram consists of states and transitions. The following are the types of states and transitions that can be found in the activity diagrams used in the proposed testing approach.

- *Action states*: a state in which actions are performed is called an action state. Action states are associated with only one action in the activity diagrams used in this work. An action in an action state can be one of the following:
 - An *assignment* of the form *element* := *expression*, where *element* can be a local variable, or an attribute, and *expression* can be a value or a call to a value-returning procedure or a constructor (value returned is a reference to the new object). For example, in Figure 4 the action state with the action

borrinfo := create :: *Borrows*(*currdate* + 14, *currdate*)

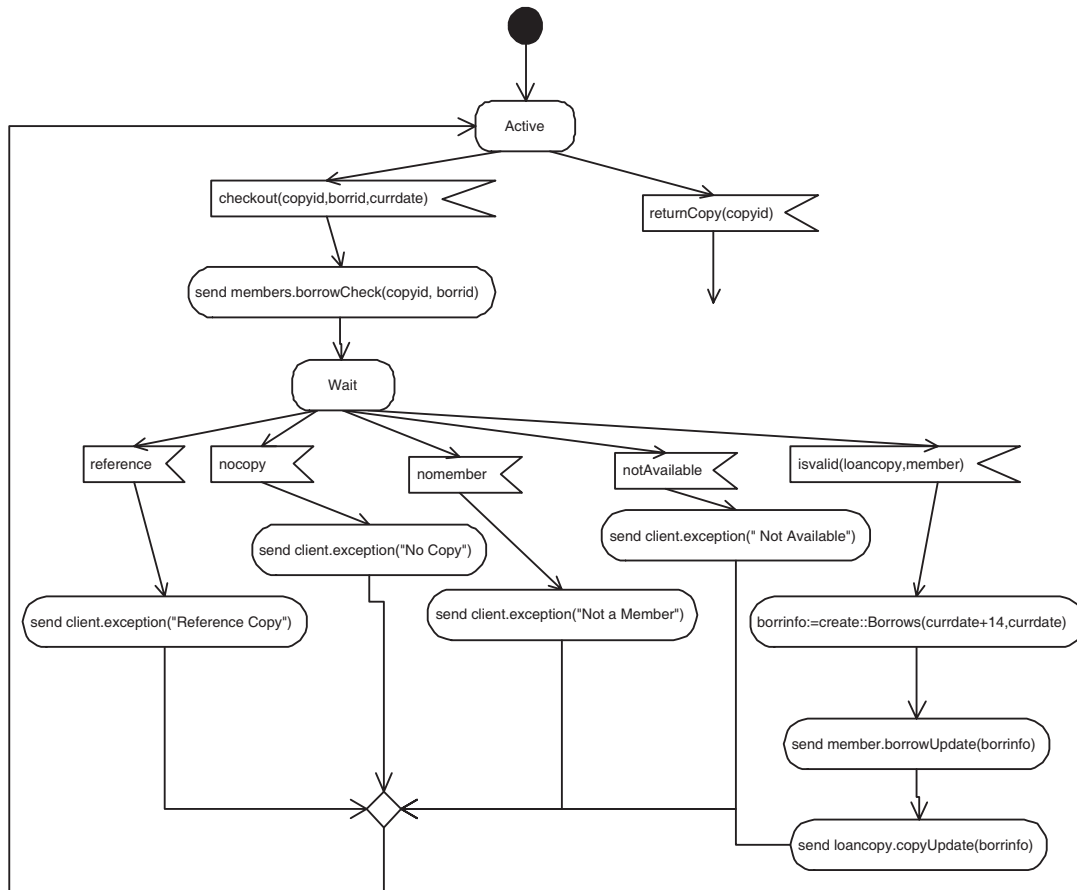


Figure 4. Partial activity diagram for the system controller.

assigns a newly created *Borrows* object to the variable *borrinfo*. Local variables are simply variables used to store information temporarily during the activity. They are not attributes because they do not represent properties of objects.

- A *send action* of the form *send object.signal*. A signal is a communication between objects and can have data associated with it (e.g., the signal *checkout* in Figure 4 has data *copyid*, *borrid*, and *curdate* associated with it). For example, in Figure 4, entering the action state with the action *send members.borrowCheck(copyid, borrid)*, results in the sending of the *borrowCheck(copyid, borrid)* signal to the *members* object.
- A *procedure call* of the form *procedureA(arguments)*, or a call to a constructor of the form *create :: classname(constructor arguments)*. Procedures and constructors are

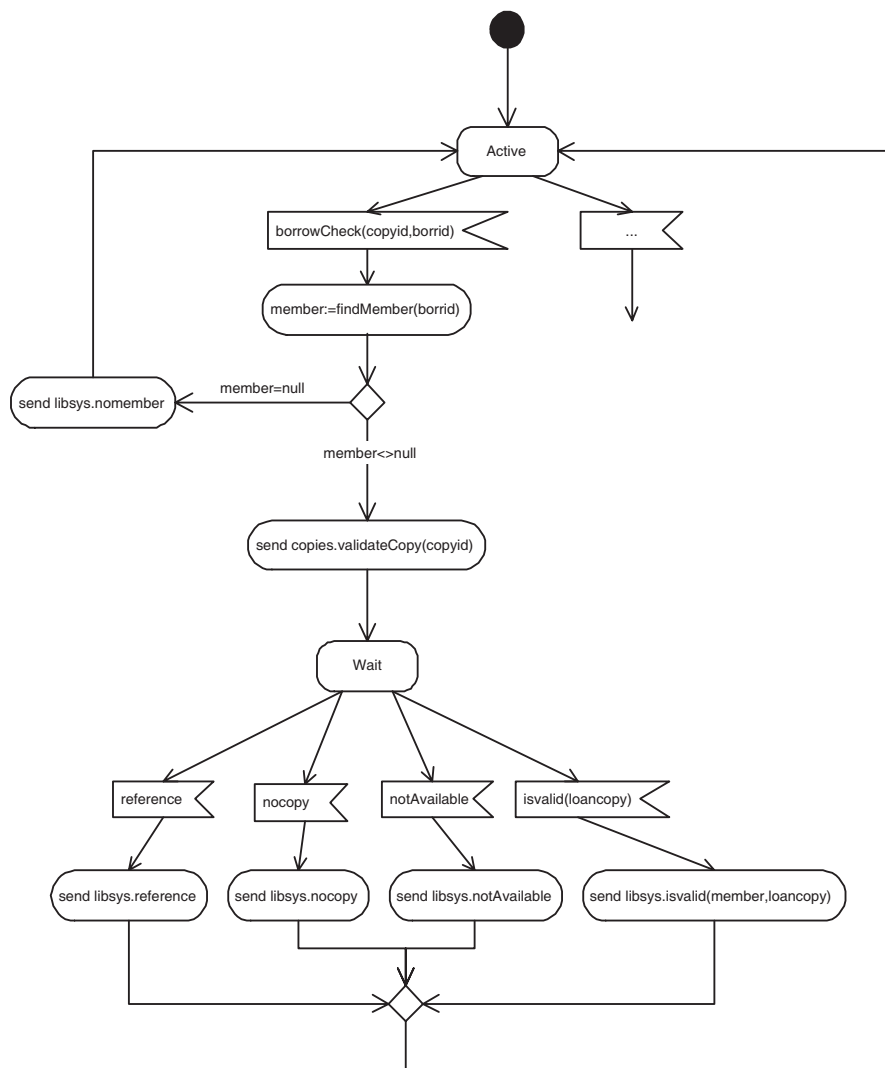


Figure 5. Partial activity diagram for the member manager.



expressed in some programming language. Java is used in this work. An action state with a procedure or constructor call results in the procedure or constructor being executed. Execution of a procedure or constructor cannot be interrupted, that is, a transition to another state can only be made after the procedure or constructor returns. The action *member := findMember(borrid)* in Figure 5 results in a call to *findMember()*, and storing of the return value in the local variable *member*.

- *Input signal transitions*: these transitions take place when the indicated signals are consumed by the state machine. For example, Figure 4 shows a transition from the state *Active* to an action state as result of receiving the input signal *checkout(copyid, borrid, currddate)*.

A simple model of execution for activity diagrams is used. The action in an action state is executed on entry to the state. Each activity diagram has a signal queue for incoming signals. The queue is checked whenever the state machine moves into a new non-action state or after the action in an action state is performed. If there are signals in the queue, the signal at the top of the queue is consumed (i.e. removed from the queue) unless explicitly deferred (each state can define a set of signals to be deferred; those signals are not consumed if they do not cause a transition to another state). If the activity diagram indicates that the signal causes a transition to another state, then the transition is made; if not, the machine remains in the current state. A deferred signal moves back to the top of the queue after a signal has been consumed.

One can derive the interactions between objects from a set of activity diagrams by tracking the send actions specified in the activity diagrams. This view of a system's behaviour is conveniently captured by interaction diagrams. A *collaboration diagram* is an interaction diagram that characterizes how objects interact to achieve a behavioural goal. *Sequence diagrams* are another form of interaction diagrams that can contain the same interaction information, but in a different format (e.g., collaboration diagrams specify the object structure on which the behaviour is carried out; such information is implicit in sequence diagrams). 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 behaviours are performed.

A collaboration diagram consists of two parts: a collaboration and an interaction. A *collaboration* specifies the object structure needed to support a particular behaviour. It specifies the properties that objects must have if they are to participate in the behaviour. The specifications can be expressed in terms of *collaboration roles*, where a role specifies the attributes and operations of a class needed for a particular behaviour, or in terms of prototypical objects. A collaboration diagram that consists of roles is called a *specification-level* collaboration diagram, while a diagram consisting of prototypical objects is called an *instance-level* collaboration diagram. One can use either diagram to specify collaborations. Instance-level collaboration diagrams are used in this paper.

An *interaction* specifies the communications that take place between objects when performing a specified behaviour. An interaction is always defined within a collaboration. Communication stimuli between objects are specified by messages. In the interaction diagrams used in this work, a message specifies a signal communication. An example of a collaboration diagram is given in Figure 6.

In the example 'return' messages are differentiated using 'half arrowheads'. This is only a syntactic distinction; all signals specified by messages are asynchronous communications that are placed in the signal queues of receiving objects. A textual description of the signal sequence for the library checkout behaviour is given in Figure 7.

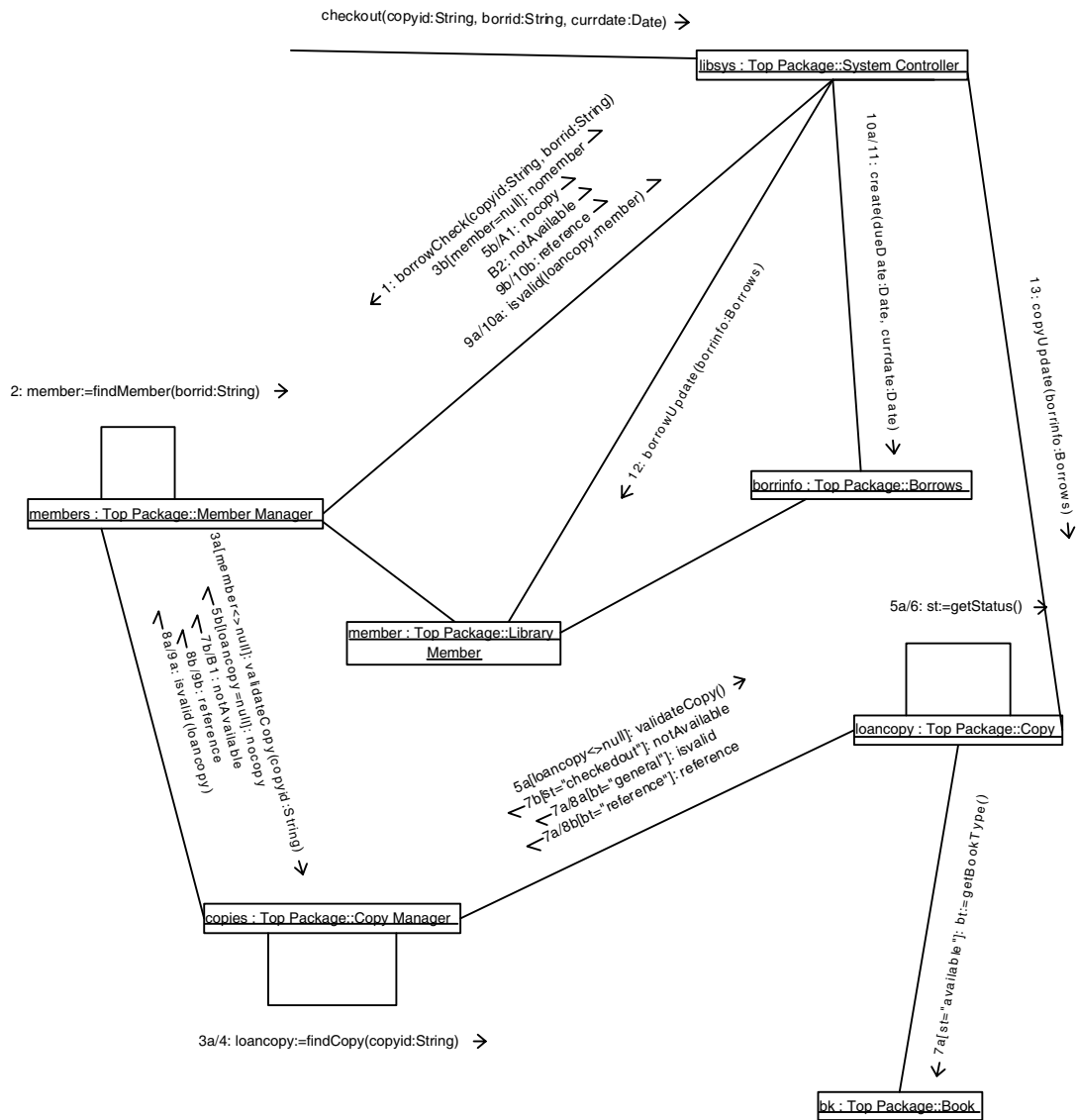


Figure 6. An example of an instance-level collaboration diagram.

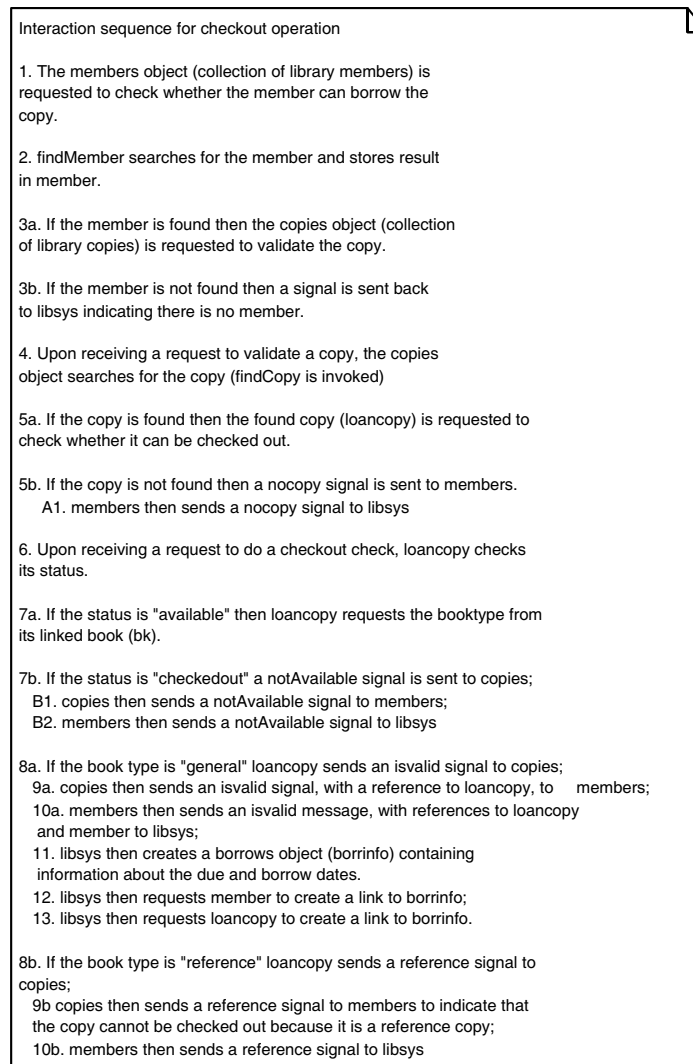


Figure 7. Textual description of signal sequence in library checkout behaviour.



2.3. Related work on UML-based testing

Binder [16] describes generic test requirements derived from UML models and introduces the test design pattern. The pattern focuses on determining appropriate test strategies, faults that may be detected, test case development from the model and 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, but generates code test requirements from them.

Briand and Labiche [17] describe the TOTEM (Testing Object-orientEd systEmS with the Unified Modelling 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 OCL 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 behaviours of objects. Moreover, this approach does not evaluate UML artifacts.

Offutt and Abdurazik [18] 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. For this reason, the approach performs only a limited form of class-level testing. The approach does not directly support testing of object interactions.

Abdurazik and Offutt [19] 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. The static checking approach involved evaluating the code in terms of four elements from collaboration diagrams:

1. *Classifier roles*: test for required attributes and operations.
2. *Collaborating pairs*: each pair on the diagram should be tested or checked at least once.
3. *Stimulus*: each stimulus should be used as the basis for generating test inputs.
4. *Local variable definition-usage link pairs*: check for data flow anomalies at the design level.

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 also to use information from different types of diagrams (class and collaboration diagrams) together during testing.

Scheetz *et al.* [20] developed an approach to generating system (black box) test cases for code from UML class diagrams. A restricted form of UML class diagrams is used to represent the conceptual architecture of the system under test. The class diagrams used in the approach consist only of classes, associations (including aggregation) and specialization structures. From the class diagrams test objectives are derived that cover single classes, groups of classes and their associations as depicted in the 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.

The approach to deriving test objectives is based on the following.

1. Test objectives can be derived for each class (and its instantiated objects) in the diagram separately, considering the class' relationships to other classes.
2. Test objectives for a complete class diagram can be aggregated from those for individual classes (and their instantiated objects), subject to constraints through class relationships.
3. Test objectives can be expressed as states of objects instantiated from classes depicted in the UML class diagrams.

The class-based test objectives can be aggregated into system level test objectives subject to constraints defined as part of the UML model. In [20] these test objectives are converted into test cases with the use of an AI planner as the generation mechanism. When multi-object test objectives are aggregated, the space of possible test objectives can become intractably large. The testing criteria described in this paper can be used to guide the selection of building blocks and aggregation of individual test objectives, thus reducing this complexity.

3. TESTING UML MODELS: AN OVERVIEW

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 modelled behaviour, starting from a specified configuration (object structure), using a sequence of signals that trigger modelled behaviours. During execution the start configuration can change as a result of adding and deleting objects and links and changing the values of object attributes.

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

$$\langle\langle \textit{sequence_of_signals}, \textit{start_configuration}, \textit{prefix} \rangle\rangle$$

A configuration is a structure of objects that satisfies the constraints expressed in the DCDs. 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 6 is given below.

- Sequence of signals: *checkout(copy123, borr1, 10102002)*, *checkout(copy123, borr2, 10102002)*, *checkout(copy321, borr1, 10102002)*.
- Start configuration: consists of a set of copies that includes *copy123*, but not *copy321*, and a set of members that includes *borr1* and *borr2*.
- Prefix: *addCopies({copy123})*, *addMember({borr1, borr2})*.

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 behaviours 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 be deemed a success. Some of the objects in the configuration may not correspond to concepts in the requirements models (e.g., the *System Controller* class does not correspond to any concept in the library system conceptual model), in which case it is necessary to apply abstraction functions to the start and end configurations (which hide details of design objects that do not correspond to requirements concepts) before applying the pre- and post-conditions. Abstraction mappings are not discussed in this paper.

4. TESTING CRITERIA FOR UML DIAGRAMS

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 DCDs and interaction diagrams are presented.

4.1. DCD criteria

DCD criteria determine the configurations that a behavioural 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.

Figure 8 shows a DCD for the *Royal and Loyal System* (RLS) used to illustrate how DCD criteria can be used to obtain test objectives. RLS handles loyalty programs for companies that offer their customers bonuses in the forms of bonus points and air miles. Details of this example can be found in Chapter 2 of the book by Warmer and Kleppe [21].

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 OCL statements (not shown in Figure 8). Table I summarizes the test criteria proposed for classes, associations and generalizations.

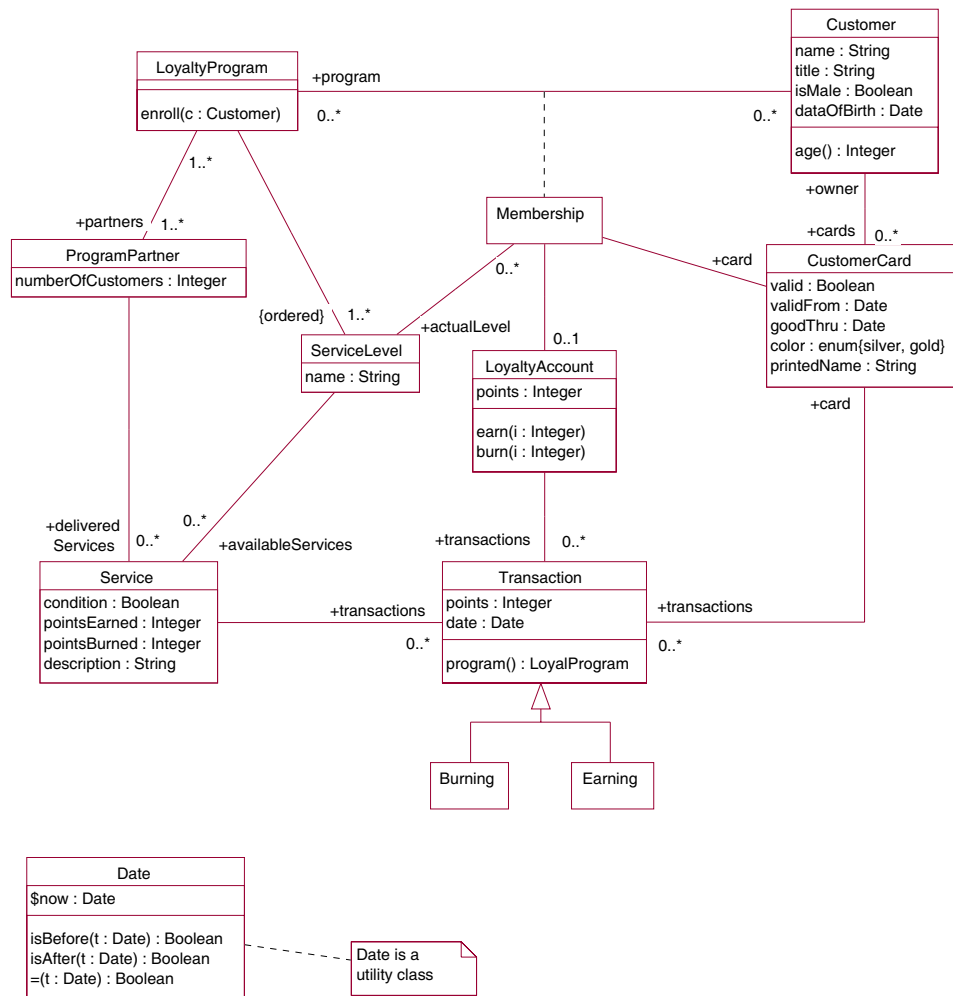


Figure 8. Class diagram for the RLS [21].



Table I. Candidate test criteria for class diagrams.

<p><i>Association-end multiplicity (AEM) criterion</i> Given a test set T and a system model SM, T must cause each representative multiplicity-pair in SM to be created.</p> <p><i>Generalization (GN) criterion</i> Given a test set T and a system model SM, T must cause every specialization defined in a generalization relationship to be created.</p> <p><i>Class attribute (CA) criterion</i> 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.</p>

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 in the following ways.

1. *Knowledge-based partitioning.* Using knowledge of the problem domain, the tester can determine the partitions. Such knowledge may be gained by examining the OCL constraints associated with the model. A value domain for a UML construct may be characterized using one or more constraints. A systematic approach to determining partitions for the value domain may proceed as follows:
 - (a) Using one of the constraints, create an initial partitioning (e.g., a partition in which the constraint is true for each element in the partition, and a partition in which the constraint is false for each partition element). The initial partitioning can also be created using default partitioning (see below).
 - (b) Refine the initial partitioning by further partitioning each initial partition using a different constraint. Repeat this step until there are no more constraints to be applied.

The use of OCL constraints for determining partitions will be illustrated when the CA (Class Attribute) criterion is considered.

2. *Default partitioning.* The tester can use default partitions consisting of minimum, non-boundary and maximum values. For example, given a multiplicity $p..n$, the minimum value partition is $\{p\}$, a non-boundary partition is $\{p + 1, \dots, n - 1\}$ and the maximum value partition is $\{n\}$. In the case where the multiplicity is defined as '*', it will be up to the tester to select an upper limit based on the system requirements.

4.1.1. Association-end multiplicity (AEM) criterion

An association-end multiplicity (AEM) 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.



For example, the *Customer-LoyaltyProgram* association in Figure 8 shows that a single instance of the *Customer* class can be associated with zero to many instances of the *LoyaltyProgram* class, and a single instance of the *LoyaltyProgram* class can be associated with zero to many instances of the *Customer* class. An AEM criterion determines the set of *representative* multiplicity tuples that must be created during a test.

In order to establish the set of representative multiplicity tuples, the adapted form of category-partition testing [10] previously outlined can be used. Using this method, the multiplicity domain is partitioned into equivalence classes, and one value from each class is selected for the set of representative multiplicities. The default partitioning approach is illustrated using the *Customer-LoyaltyProgram* association shown in Figure 8.

1. Create a multiplicity set by arbitrarily selecting a value from each partition. A possible multiplicity set for class *Customer* is $\{0, u, n\}$ and one for *LoyaltyProgram* is $\{0, v, m\}$, where m and n are upper limits determined by the tester, u is a value from the set $\{1, \dots, n-1\}$ and v is a value from the set $\{1, \dots, m-1\}$.
2. Create a set of configurations $\{(r, s)_1, (r, s)_2, \dots\}$ from the Cartesian product of the multiple sets of the *Customer* and *LoyaltyProgram* instances, where r is the number of *Customer* instances linked with s instances of *LoyaltyProgram*. For the association, this configuration set is: $\{(0, 0), (0, v), (0, m), (u, 0), (u, v), (u, m), (n, 0), (n, v), (n, m)\}$.

The AEM criterion will ensure that design testers run tests that exercise configurations that contain boundary and non-boundary occurrences of links between objects (an instance of an association is a link).

4.1.2. Generalization (GN) criterion

Figure 8 shows an example of a generalization relationship where the classes *Burning* and *Earning* specialize the class *Transaction*. The generalization criterion defines the *representative* set of specialization types that must be created from a DCD's superclasses during a system model test. A test that causes one of each type of specialization (e.g., *Burning* and *Earning*) to be created is adequate with respect to the GN criterion.

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 behaviours in which superclass objects are expected using specializations of the superclass instead of superclass objects.

4.1.3. Class attribute (CA) criterion

Attribute values may restrict the behaviour of an object. For example, particular attribute values may restrict how an object responds to signals. Thus, the *value* space of attributes provides 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:



1. use category-partitioning to create representative value sets for each attribute;
2. take the Cartesian product of each value set to create an aggregate set of representative attribute value tuples for each class;
3. identify valid and invalid aggregated sets.

Defining the value space for Boolean attributes is a trivial exercise since their values are either true or false. For scalar and string attributes a form of category partitioning is applied to define a representative set of values. The partitions can be determined by the tester using domain knowledge (e.g., as expressed in OCL constraints), or by using boundary and non-boundary values to determine default partitions (in situations where there is an unbounded limit, a maximum or minimum value can be chosen based on the modeller's knowledge of the problem).

This section illustrates how constraints (expressed in OCL or otherwise) can be used to determine partitions using the RLS class diagram. The RLS has the following constraints from which partitions for attributes can be obtained (in some cases constraints are expressed using a mixture of English and familiar mathematical notation rather than OCL to enhance readability).

1. *Customer*: the *age* property is associated with two constraints: (1) *age* \geq 18, (2) *age* $>$ 60 implies that each *CustomerCard* linked to the *Customer* has *color* = *gold*. Using the constraint (1), three partitions are obtained for the attribute *age*: (0, 18), {18}, (18, 130). Using constraint (2) the above can be further refined to (0, 18), {18}, (18, 60), {60}, (60, 130). Representative values can be selected from each partition, for example, 5, 18, 54, 60, and 100. The partition (0, 18) is an invalid partition. Invalid partitions are useful for testing fault recovery mechanisms and to check if constraints are correctly specified.
2. *CustomerCard*:

```
validFrom.isBefore(goodThru)
```

In the above, *isBefore* is an operation with the following postcondition:

```
Date::isBefore(t:Date):Boolean
  post: if self < t
        then result = True
        else result = False
```

From the postcondition three partitions are derived for the (*validFrom*, *goodThru*) pair.

- (a) *Valid card*: {(*validFrom*, *goodThru*) | *validFrom* < *goodThru*}.
- (b) *Last day of validity*: {(*validFrom*, *goodThru*) | *validFrom* = *goodThru*}.
- (c) *Expired card*: {(*validFrom*, *goodThru*) | *validFrom* > *goodThru*}.

A set of sample input values for {(*validFrom*, *goodThru*)} that can be used to fulfil the test criteria is:

```
{(02/09/97, 03/07/00), (03/03/98, 03/03/98), (03/07/00, 02/09/97)}
```

3. *CustomerCard*:

```
printedName = customer.title.concat(customer.name)
```



Table II. Truth table.

	<code>actualLevel.name('Silver')</code>	<code>actualLevel.name('Gold')</code>
<code>card.color(#silver)</code>	T	F
<code>card.color(#gold)</code>	F	T

Two partitions that evaluate the above expression to True and False, respectively, are obtained.

(a) *Names that match:*

```
{customerCard.printedName, customer.title, customer.name}|  
printedName = customer.title.concat(customer.name)}.
```

(b) *Names that do not match:*

```
{customerCard.printedName, customer.title, customer.name}|  
printedName <> customer.title.concat(customer.name)}.
```

4. *LoyaltyProgram:*

```
partners.deliveredServices.transaction ->  
select (oclType=Burning) -> collect(points) -> sum<10000
```

The following partitions based on the sum of the points in all the instances of Burning are obtained:

(a) the sum is greater than or equal to 10 000;

(b) the sum is less than 10 000.

5. *Membership:*

```
actualLevel.name = 'Silver' implies card.color=#silver
```

and

```
actualLevel.name = 'Gold' implies card.color=#gold
```

Since the domain of `actualLevel.name` is {'Silver', 'Gold'}, and the type of `card.color` is `enum{silver, gold}`, the truth table can be built as shown in Table II.

('Silver', #silver) and ('Gold', #gold) are the aggregated partitions. ('Silver', #gold) and ('Gold', #silver) can be used for fault recovery testing.

4.2. Interaction diagram criteria

Table III lists the candidate test criteria based on coverage of collaboration diagram elements. The library system collaboration diagram shown in Figure 6 will be used to illustrate the criteria defined in Table III.



Table III. Candidate test criteria for collaboration diagrams.

<p><i>Condition coverage (Cond) criterion</i> Given a test set T and collaboration diagram CD, T must cause each condition in each decision to evaluate to both TRUE and FALSE.</p> <p><i>Full predicate coverage (FP) criterion</i> Given a test set T and collaboration diagram CD, T must cause each clause in every condition in CD to take the values of TRUE and FALSE while all other clauses in the predicate (condition) have values such that the value of the predicate will always be the same as the clause being tested.</p> <p><i>Each message on link (EML) criterion</i> Given a test set T and collaboration diagram CD, T must cause each message on a link connecting two objects in CD to be executed at least once.</p> <p><i>All message paths (AMP) criterion</i> Given a test set T and collaboration diagram CD, T must cause each possible message path (sequence of message numbers) in CD to be taken at least once.</p> <p><i>Collection coverage (Coll) criterion</i> Given a test set T and collaboration diagram CD, T must test each interaction with collection objects of various representative sizes at least once.</p>

4.2.1. 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. In Figure 6, a signal specified by message 3a occurs when the condition *member* \neq null is true. A test set that satisfies the Cond criterion must include test cases that make this condition true and false.

4.2.2. 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.

4.2.3. 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.

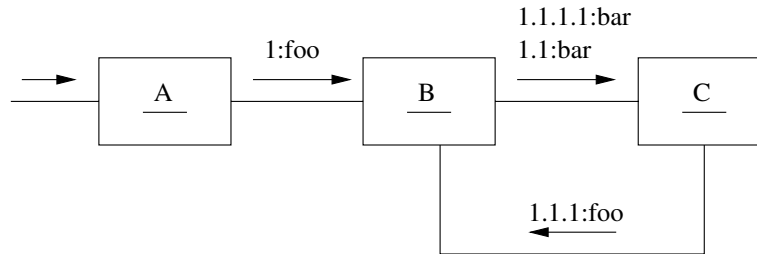


Figure 9. Example collaboration diagram with apparent cycle.

4.2.4. All message paths (AMP) criterion

A message path is a sequence of messages. Examples from Figure 6 are shown below.

- The path[§] consisting of the messages `1:borrowCheck()`, `2:findMember()`, `3a:validateCopy()`, `4:findCopy()`, `5b:nocopy` and `A1:nocopy`, abbreviated as `{1, 2, 3a, 4, 5b, A1}`.
- The path consisting of the messages `1:borrowCheck()`, `2:findMember()`, `3a:validateCopy()`, `4:findCopy()`, `5a:validateCopy`, `6:getStatus()`, `7b/B1:notAvailable` and `B2:notAvailable` abbreviated as `{1, 2, 3a, 4, 5a, 6, 7b, B1, B2}`.

Note that arbitrary numbers of cycles are not possible in a collaboration diagram since messages are ordered and a message with a lower number (for example 1.1) must not be executed after a message with a higher number (for example 1.1.1). Thus, while the graph of a collaboration diagram may have cycles, actual paths are limited by the message number along its edges. In Figure 9, A sends a message `1:foo` to B. B in turn sends a message `1.1:bar` to C. C sends a message `1.1.1:foo` to B. B now sends a message `1.1.1.1:bar` to C. There is an apparent cycle in the graph (`1.1:bar`, `1.1.1:foo`, `1.1.1.1:bar`, `1.1.1.1.1:foo`, ...), but actually there is only one path (`1:foo`, `1.1:bar`, `1.1.1:foo`, `1.1.1.1:bar`). The limitation on the number of actual paths is a feature of UML which requires message numbers to be present along the edges.

The AMP criterion ensures that all message paths in a collaboration diagram are exercised.

4.2.5. Collection coverage (Coll) criterion

Collaboration diagrams may specify communications with collections of objects. The collections can be partitioned into different domains, much like the partitioning of multiplicities of objects and relations in the class diagram. The Coll criterion requires that a collection in each domain be

[§]Parameters and conditions have been omitted for brevity.



instantiated at least once. This criterion ensures that interactions with collections of various sizes, including boundary-sized collections, are tested.

4.3. Deriving test objectives

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 a specific condition. Alternatively, the Coll criterion may be associated with test objectives that require the system to be brought into a specific configuration that has a specified number of objects in a collection appearing in a collaboration diagram.

5. TEST METHOD USING UML-BASED 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 solved, it is assumed that the class diagram criteria given in this paper need to be met, as well as the AMP 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 $P = \{P_1, \dots, P_k\}$ 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.

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 $p_i \in PI$, $i = 1, \dots, k_{PI}$ 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.

6. CASE STUDY

Given the complexity of large systems, it is expected that a large set of test cases would be required to achieve 100% test coverage. However, rarely does each coverage item require a separate test. Thus, the number of coverage items required is a worst case scenario and an unrealistic predictor of the number of test cases required.

There is plenty of evidence for 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 [22] 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 behaviour because a single test covers items required by both class diagrams 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.

The library system described in Section 2 was used to conduct a case study in which a model of the checkout activity is tested. Figure 2 shows the use cases for checking out a copy, Figure 3 shows a partial class diagram and Figure 6 shows the instance-level collaboration diagram for the check-out activity. Each test case consists of a single checkout signal that requires *copyid*, *borrid* and *currdate* parameters.

Table IV shows the parameters that are passed in the test signal, the type of configuration used and the coverage elements exercised during the test. The *Parameters* column represents the three parameters passed to the checkout signal. The test inputs *I-CopyId* and *I-MemberId* indicate invalid values of *copyid* and *borrid*, i.e. those that do not exist in the system. The inputs *V-CopyId* and *V-MemberId* indicate valid values of *copyid* and *borrid*, respectively.

The columns *Copy-Config* and *Member-Config* represent state predicates. *Copy-Config* is true if and only if the copy corresponding to the *copyid* passed in as a parameter to the test signal is in the start configuration of the test. Similarly, *Member-Config* is true if and only if the member corresponding to the *borrid* passed in as a parameter to the test signal is in the start configuration of the test.

The *Status* column represents the states that can be taken by the copy in question (e.g., available, checked-out, or not applicable—NA). The *BookType* column indicates the type of book (e.g., reference, general, or not applicable—NA). The test cases shown in the table were derived using a combination of CA and AEM criteria, and constraints from the check-out use case specified in OCL.

The *Coverage* column shows the coverage elements from the DCD criteria that were exercised during the test. For example, the entry 'AEM: Member(0)–Copy(1)' indicates that the test signal resulted in the coverage of the *Borrows* association between a *Member* and *Copy*, and that the multiplicities



Table IV. Test cases.

	Parameters	Copy-Config	Member-Config	Status	BookType	Coverage
1	<i>I-CopyId, I-MemberId, 02/21/03</i>	False	False	NA	NA	AEM: Member(0)–Copy(0)
2	<i>I-CopyId, V-MemberId, 02/21/03</i>	False	True	NA	NA	AEM: Member(1)–Copy(0)
3	<i>V-CopyId0, I-MemberId, 02/21/03</i>	True	False	Av	Gen	AEM: Member(0)–Copy(1) CA: Av, Gen
4	<i>V-CopyId1, I-MemberId, 02/21/03</i>	True	False	Av	Ref	AEM: Member(0)–Copy(2) CA: Av, Ref
5	<i>V-CopyId2, I-MemberId, 02/21/03</i>	True	False	Chk	Gen	AEM: Member(0)–Copy(3) CA: Chk, Gen
6	<i>V-CopyId3, I-MemberId, 02/21/03</i>	True	False	Av	Gen	AEM: Member(0)–Copy(4) CA: Av, Gen
7	<i>V-CopyId0, V-MemberId1, 02/21/03</i>	True	True	Av	Gen	AEM: Member(1)–Copy(1) CA: Av, Gen
8	<i>V-CopyId1, V-MemberId2, 02/21/03</i>	True	True	Av	Ref	AEM: Member(1)–Copy(2) CA: Av, Ref
9	<i>V-CopyId2, V-MemberId3, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(1)–Copy(3) CA: Chk, Gen
10	<i>V-CopyId3, V-MemberId4, 02/21/03</i>	True	True	Av	Gen	AEM: Member(1)–Copy(4) CA: Av, Gen
<i>Tests after status change: available → checked-out</i>						
11	<i>V-CopyId0, V-MemberId1, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(1)–Copy(2) CA: Chk, Gen
12	<i>V-CopyId3, V-MemberId1, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(1)–Copy(2) CA: Chk, Gen
13	<i>V-CopyId0, V-MemberId2, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(2)–Copy(1) CA: Chk, Gen
14	<i>V-CopyId3, V-MemberId2, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(2)–Copy(1) CA: Chk, Gen
15	<i>V-CopyId1, V-MemberId2, 02/21/03</i>	True	True	Av	Ref	AEM: Member(2)–Copy(1) CA: Av, Ref
16	<i>V-CopyId2, V-MemberId2, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(2)–Copy(1) CA: Chk, Gen
17	<i>V-CopyId4, V-MemberId2, 02/21/03</i>	True	True	Av	Gen	AEM: Member(2)–Copy(1) CA: Av, Gen
18	<i>V-CopyId5, V-MemberId2, 02/21/03</i>	True	True	Av	Gen	AEM: Member(2)–Copy(2) CA: Av, Gen
<i>Tests after status change: available → checked-out</i>						
19	<i>V-CopyId4, V-MemberId1, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(1)–Copy(3) CA: Chk, Gen
20	<i>V-CopyId4, V-MemberId1, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(1)–Copy(3) CA: Chk, Gen
21	<i>V-CopyId4, V-MemberId2, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(2)–Copy(3) CA: Chk, Gen
22	<i>V-CopyId4, V-MemberId2, 02/21/03</i>	True	True	Chk	Gen	AEM: Member(2)–Copy(3) CA: Chk, Gen

Table V. Conditions covered by the test set \mathcal{T} .

Condition	Test cases
member = null	1, 3, 4, 5, 6
member \neq null	2, 7, 8, 9, 10
loancopy = null	2
loancopy \neq null	7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22
available = true	7, 8, 10, 15, 17, 18
available \neq true	9, 11, 12, 13, 14, 16, 19, 20, 21, 22
general = true	7, 10, 17, 18
general \neq true	8, 15

Table VI. Paths covered by the test set \mathcal{T} .

Paths	Test cases
1,2,3b	1, 3, 4, 5, 6
1,2,3a/4,5b,5b/A1	2
1,2,3a,3a/4,5a,5a/6,7b,7b/B1,B2	9, 11, 12, 13, 14, 16, 19, 20, 21, 22
1,2,3a,3a/4,5a,5a/6,7a,7a/8b,8b/9b,9b/10b	8, 15
1,2,3a,3a/4,5a,5a/6,7a,7a/8a,8a/9a,9a/10a,10a/11,12,13	7, 10, 17, 18

at the Member and Copy ends are 0 and 1, respectively. The entry 'CA: Av, Gen' indicates that the following class attribute partitions are covered:

```
Class Copy, Status=Available
Class Book, BookType=General
```

The set (\mathcal{T}) of 22 test cases was found to be adequate with respect to the CA and AEM criteria. \mathcal{T} also covers a number of elements from the collaboration diagram criteria. Tables V and VI show the conditions and paths that are covered by the test set \mathcal{T} . The conditions shown in Table V (e.g., member = null) and paths shown in Table VI (e.g., [1,2,3b]) are taken from the collaboration diagram in Figure 6. Since the conditions have only one predicate each, \mathcal{T} is also adequate with respect to the FP criterion (not shown in the tables). This example clearly shows that one test case will cover quite a few coverage items.

While this case study showed that test cases cover multiple coverage items for both class diagram and collaboration diagram notations, it is important to note that it is also limited in its generalizability. Further empirical work is necessary to investigate the degree of 'multiple dipping' that can be expected for test cases. It is also important to note that theoretical limits on worst case behaviour will be high.



7. PRELIMINARY EVALUATION OF THE CRITERIA

The effectiveness of the criteria is based upon the frequency at which the faults targeted by the criteria occur in practice (the assumption is that tests that satisfy the criteria have a high probability of uncovering the faults targeted by the criteria). Good criteria should be developed to uncover significant faults that can occur in the artifacts under test. The types of faults that can be uncovered by tests that satisfy the criteria are identified in this section. The results of a preliminary investigation into the frequency at which the faults occur in practice are also presented.

7.1. Using the DCD criteria to uncover faults

Obtaining tests that satisfy the DCD criteria involves generating configurations that have the desired multiplicities and attribute values. Tests that use these configurations satisfy the DCD criteria. The generation of configurations from DCDs can be done automatically—a DCD can be viewed as a template for stamping out configurations given desired multiplicities and attribute values. The criteria determine domains of values from which representative values can be selected. A tool that (1) selects a value from each domain of multiplicity and attribute values, and (2) uses the values to stamp out configurations is currently under development.

In this subsection the types of faults that can be uncovered by the criteria are identified. The set of fault types targeted by a criterion is referred to as the *fault model* of the criterion. A test set that satisfies the AEM criterion is likely to uncover faults related to behaviours that manipulate collections of objects that are linked to an object. The classes of faults that can be uncovered using tests that satisfy this criterion include loose association multiplicities and failure of behaviours to add or delete links as required, especially when boundary multiplicities are exercised.

Loose association multiplicities allow more links between objects than are required. For example, one may put a multiplicity ‘0 or more’, when an upper limit is really required. For example, consider a requirements change in the library system described in Section 2 that restricts the number of copies that can be checked out by a library member at any time to five copies. The change was made in the requirements model but not in the design model; thus the checkout behaviour in the design does not ensure that a member has only five books checked out. This fault is uncovered by executing the checkout behaviour on a configuration that includes an upper limit test multiplicity. The test will produce a configuration that violates the constraints specified in the requirements. In general, faults associated with loose multiplicities can be uncovered by running tests on configurations determined by boundary multiplicities.

Failure to add or create links, especially when boundary multiplicities are involved, can also be detected by checking that the configuration at the end of the test satisfies the constraints expressed in the requirements.

The GN criterion can be used to develop tests that check whether objects of subclasses can be used anywhere objects of their superclasses are expected. For example, consider the case where a square class is specified as a subclass of rectangle (see Figure 10). The *Square* inherits the attributes and the *resize()* operation of *Rect*. The inherited attributes are constrained in *Square* ($\{x = y\}$). The *resize* operation defined on *Rectangle* takes in two arguments, *a*, *b*, where *a* is the factor added to *x* and *b* is the factor added to *y* in the operation. A test of *resize()* on a *Square* object, with $a \neq b$ would reveal

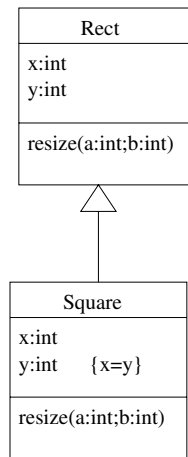


Figure 10. A shapes DCD.

that the result is a rectangle, not a square (indicating that the *resize()* operation needs to be redefined in *Square* if the constraint on the attributes is to be preserved).

The values of attributes can be used during an activity to determine the actions that will take place. Consider an inventory system in which a reorder activity is required when the stock level of a product falls below a particular threshold. If the stock level falls below the threshold during an invoicing activity, a reorder activity must be spawned. To support adequate testing of the invoicing behaviour, the attribute that stores the stock level should be associated with three partitions: (1) a partition consisting of values above the threshold, (2) a partition consisting of a value equal to the threshold, and (3) a partition consisting of values below the threshold. This partitioning is obtained by using domain knowledge (expressed as part of the post-condition for the invoicing operation in the Requirements Model). Tests that use representative values of these partitions are highly likely to uncover situations in which the reorder behaviour is not invoked when it should be.

7.2. Using collaboration diagram criteria to uncover faults

Obtaining test cases that satisfy collaboration diagram criteria should be no more difficult than obtaining test cases that satisfy conditional and path coverage criteria for code testing. Given that models provide a more abstract view of behaviour, the task of obtaining test cases that satisfy the criteria should require less effort.

Tests that satisfy the Cond and FP criteria target are intended to exercise behavioural paths determined by conditions. These criteria target faults related to using inadequate conditional control flow structures.

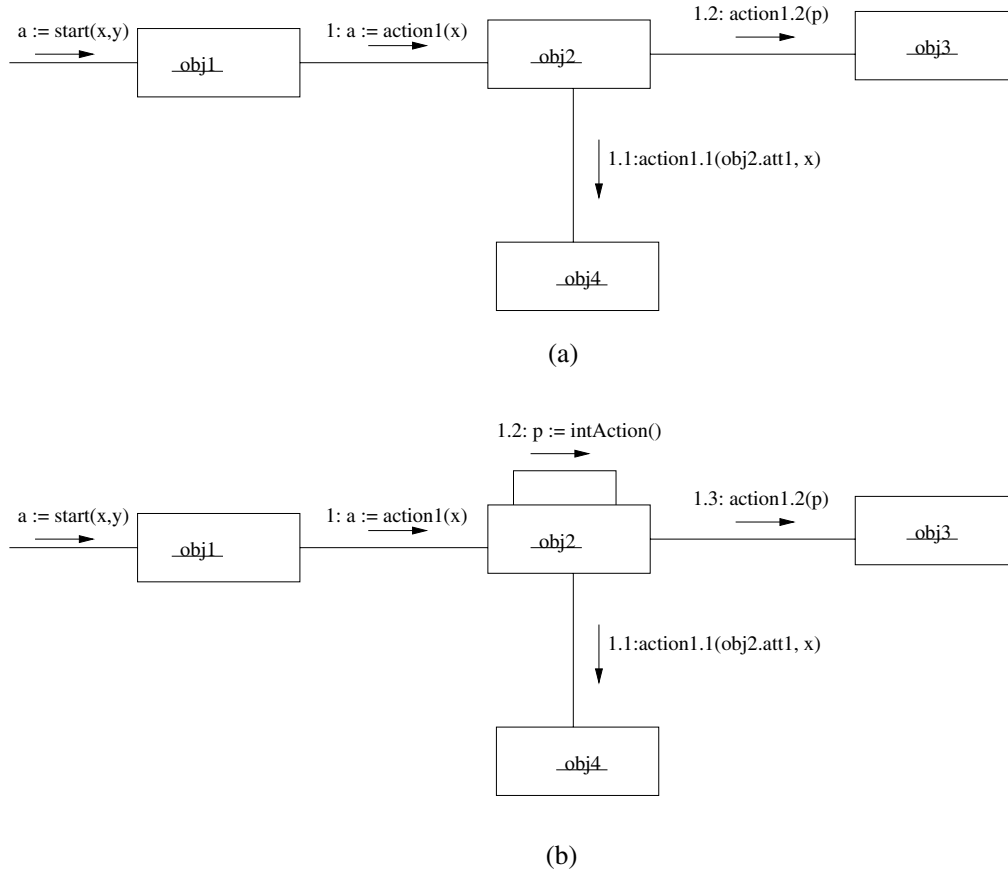


Figure 11. Dataflow gap in collaboration diagram: (a) collaboration diagram with dataflow gap, source of data p is not shown; (b) corrected collaboration diagram.

Tests that satisfy message path and link criteria are intended to uncover faults related to interactions between objects. Faults related to incorrect sequencing of messages, inappropriate control flows (e.g., incorrect iteration and conditional structures), missing flows, and dataflow gaps are targeted by these criteria. Missing flows can result from failing to cover all cases in a conditional. In the execution of a model this type of fault would result in a behaviour in which the behaviour stops in a non-terminal state (a terminal state is one in which a user-generated event is needed to move the system to another state). A dataflow gap occurs when data that were not created internally by an object, nor were they explicitly communicated to the object, appear as an argument in a message emanating from the object. An example of a collaboration diagram with a dataflow gap is shown in Figure 11.



Table VII. Faults identified in student projects.

Fault type	Frequency	Criteria
Incorrect sequence numbering	16	EML, AMP
Missing flows	8	Cond, FP
Dataflow gaps	3	EML, AMP

7.3. Preliminary evaluation of the fault models

The preliminary evaluation of the fault models underlying the criteria involved analysing faults made by novice modellers (senior students in an OO design course) to determine the type and frequency of modelling faults. Confidence in the utility of the criteria to uncover faults made by novice modellers is enhanced if the more frequent faults made by the students are included in the fault models on which the criteria are based. The question addressed in this preliminary study is the following:

- Do the faults included in the fault models of the criteria occur frequently in practice?

The students were assigned a modelling problem that required building a collaboration diagram that ‘implemented’ (realized) a specific scenario of a use case. The scenario was concerned with validating an order submitted by a customer and generating an invoice for the order if the order was valid and could be fulfilled. The students were given one hour to develop the models in the classroom. The problem was intended to test student ability to model interactions that involved iterations and branching. The students were not taught how to test their models using the testing criteria presented in this paper. This ensured that the students did not build models that were tested using the test criteria. The students were not required to systematically test their models.

At the end of the class the student models were collected. The analysis of the models was carried out soon after. Student models with syntactic faults that would prohibit their execution (e.g., use of syntactically invalid sequence identifiers) were excluded from the preliminary evaluation. A total pool of 21 student models was left. The students did not use the UML testing criteria during the development of their models. Each of the 21 models was analysed by the course instructor and the fault types and frequency noted. The following are the major types of faults identified.

- Incorrect sequence numbering (this can result in control jumps where control is ‘magically’ switched from one object to another).
- Missing flows.
- Dataflow gaps.

The results of the analysis are summarized in Table VII. The 21 students made 27 faults of the three fault types described previously. The frequency of each fault type is listed in Table VII, in column 2. The most commonly occurring problem was sequencing of interactions among objects. This is not really surprising given that the students did not have the mechanisms to ‘execute’ their models (as they have with code). This was followed with problems with missing flows, which, again, was not surprising. Column 3 lists the UML testing criteria that would have uncovered the fault. The faults are included in



the fault models targeted by the criteria. All faults made could have been revealed by applying these criteria.

Obviously, this is a small case study and further empirical investigation is necessary to establish the value and limitations of these criteria. The results are certainly not generalizable with respect to their applicability to modellers with more experience in UML modelling, and with respect to other UML designs that are developed in industry. However, the study indicates the potential value of the criteria.

As in the previous section, this was a case study, not a controlled experiment. Thus, there is less control over experimental variables than a controlled experiment would have had. However, it was possible to control several factors that could have affected outcomes.

- *Knowledge and use of test adequacy criteria*: none of the students had been taught the criteria described in this paper, nor had they been published. Thus, it is safe to say that the students did not use them and that the faults left in the designs occur in practice and are not affected by the test criteria.
- *Knowledge of UML design methodology*: all students were taught the same material, the same way. This reduces effects of gaining UML knowledge through other sources that could have affected results, although it does not eliminate them completely.
- The students did not know that their designs would be analysed against the fault model related to the test adequacy criteria. This prevented them from being influenced by that fact ('hypothesis guessing'). Since they were graded on correct work, there was no incentive to please the instructor by 'making mistakes'. Further, the remaining faults can be considered faults that remain after they made their best effort, i.e. they are not trivial to find.

Since the objective of this study was to show that faults detectable by the test adequacy criteria do occur, it is important to point out some of the limitations of this study.

- All subjects were students. This limits generalizability of results—experienced designers may make fewer and different mistakes.
- All subjects worked on the same design problem. While this helps to show that the faults were not random, it again limits generalizability of the results to design artifacts with similar properties as the ones investigated. Unlike a controlled experiment in which tasks are carefully controlled to be the same, case studies try to cover a wide range of applications to elicit many possible behaviours. Further work is needed to answer the research question more fully.

8. CONCLUSIONS AND FUTURE WORK

Using the constraints contained within class diagrams in the form of multiplicities, generalization and OCL, test adequacy criteria were defined for use in design reviews and in dynamic testing of implementations. The criteria defined are (1) association-end multiplicity, (2) generalization and (3) class attributes.

Test criteria were defined based on the elements in a collaboration diagram: (1) condition coverage criterion, (2) full predicate coverage criterion, (3) each message on link criterion, (4) all message paths criterion and (5) collection coverage criterion.

The process for setting up configurations and the use of the category partitioning technique was described. The kinds of faults that each criterion will detect and the types of faults the combined use



of configuration and behaviour-related criteria can uncover were enumerated. It was demonstrated that this combination of criteria can have added value in detecting faults related to inconsistencies between class diagrams and collaboration diagrams. It was also shown that these types of faults occurred in actual designs. This speaks for the usefulness of the criteria.

The criteria will be evaluated further, both analytically, in terms of Weyuker [23] and Parrish–Zweben [24] properties, and empirically in terms of scalability. Criteria based on other elements in UML, such as state diagrams, are also being investigated. Support for test generation for UML designs is being developed. Developing the testing and test generation method has great promise for a large segment of the software industry which is adopting tools like TogetherJ and Rational Rose in high numbers, but lacks a systematic, efficient way to validate the designs.

APPENDIX A. UML TERMINOLOGY

For those less familiar with UML terminology, definitions of common UML terms based on the most recent standard definition [8] are quoted here.

Activity diagram: a state machine that is used to model processes involving one or more classifiers.

Association: the semantic relationship between two or more classifiers that specifies connections among their instances.

Association end: an association consists of at least two AssociationEnds, each of which represents a connection of the association to a classifier. Each AssociationEnd specifies a set of properties that must be fulfilled for the relationship to be valid.

Class: a descriptor of a set of objects that represent software or conceptual elements, and share the same attributes, operations, methods, relationships and behaviour.

Class attribute: a feature within a classifier that describes a range of values that instances of the classifier may hold.

Class diagram: a diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

Classifier: a mechanism that describes behavioural and structural features. Classifiers include interfaces, classes, datatypes and components.

Collaboration: the specification of how an operation or classifier, such as use case, is realized by a set of classifiers and associations playing specific roles used in a specific way.

Collaboration diagram: a diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

Design model: see *model*.

Event: the specification of a significant occurrence that has a location in time and space.



Generalization: a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed.

Link: a semantic connection among a tuple of objects. An instance of an association.

Message: a specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation.

Model: an abstraction of a physical system with a certain purpose.

Multiplicity: a specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions and other purposes.

Object Constraint Language (OCL): lower order predicate language used to specify constraints on objects in the UML.

Operation: a service that can be requested from an object to effect behaviour.

Precondition: a constraint that must be true when an operation is invoked.

Postcondition: a constraint that must be true at the completion of an operation.

Signal: the specification of an asynchronous stimulus communicated between instances.

State: a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

Stimulus: an instance of an action, such as invoking an operation.

State transition: a relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied.

Use case: a specification of a sequence of actions, including variants, that a system can perform, interacting with actors of the system.

ACKNOWLEDGEMENT

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

REFERENCES

1. The Object Management Group. *OMG Unified Modelling Language Specification*, Version 1.3. OMG, 1999.
2. Atlee JM, Gannon J. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering* 1993; **19**(1):24–40.



3. Holzmann GJ. *Design and Validation of Computer Protocols (Software Series)*. Prentice-Hall: Englewood Cliffs, NJ, 1991.
4. Rushby JM. Model checking and other ways of automating formal methods. *Position Paper for Panel on Model Checking for Concurrent Programs, Software Quality Week*, San Francisco, May/June 1995.
5. The Object Management Group. *Object Constraint Language Specification*. OMG, document ad/99-06-08, ch. 7, June, 1999.
6. Mellor S, Balcer M. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley: Reading, MA, 2002.
7. Riehle D, Fraleigh S, Bucka-Lassen D, Omorogbe N. The architecture of a UML virtual machine. *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*. ACM Press: New York, 2001; 327–341.
8. The Object Management Group. *The Unified Modelling Language*, Version 1.4. OMG, formal/2001-09-67, 2001.
9. Goodenough JB, Gerhart SL. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* 1975; **1**(2):156–173.
10. Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 1988; **31**(6):676–686.
11. Offutt AJ, Irvine A. Testing object-oriented software using the category-partition method. *Proceedings of the 17th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA)*, Santa Barbara, CA, August 1995; 293–304.
12. Perry DE, Kaiser G. Adequate testing and object-oriented programming. *Journal of Object-oriented Programming* 1990; **2**(5):13–19.
13. Doong R-K, Frankl PG. Case studies on testing object-oriented programs. *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, Victoria, B.C., Canada. ACM Press: New York, 1991; 165–177.
14. Harrold MJ, McGregor JD, Fitzpatrick KJ. Incremental testing of object-oriented class structures. *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia. ACM Press: New York, 1992; 68–80.
15. Turner CD, Robson DJ. The state-based testing of object-oriented programs. *Proceedings IEEE Conference on Software Maintenance*, Montreal, Canada. IEEE Computer Society Press, 1993; 302–311.
16. Binder RV. *Testing Object-Oriented Systems: Models, Patterns, and Tools (Object Technology Series)*. Addison-Wesley: Reading, MA, 1999.
17. Briand L, Labiche Y. A UML-based approach to system testing. *Proceedings of the 4th International Conference on the UML*, Toronto, Canada (*Lecture Notes in Computer Science*). Springer, 2001; 194–208.
18. Offutt J, Abdurazik A. Generating tests from UML specifications. *Proceedings of the 2nd International Conference on the UML*, Fort Collins, TX (*Lecture Notes in Computer Science*). Springer, 1999; 416–429.
19. Abdurazik A, Offutt J. Using UML collaboration diagrams for static checking and test generation. *Proceedings of the 3rd International Conference on the UML*, York, U.K. (*Lecture Notes in Computer Science*). Springer, 2000; 383–395.
20. Scheetz M, von Mayrhauser A, France R, Dahlman E, Howe AE. Generating test cases from an OO model with an AI planning system. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, FL. IEEE Computer Society Press, 1999; 250–259.
21. Warmer J, Kleppe A. *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley: Reading, MA, 1999.
22. Weyuker EJ. The cost of dataflow testing: An empirical study. *IEEE Transactions on Software Engineering* 1990; **16**(2):121–128.
23. Weyuker EJ. Axiomatizing software test adequacy. *IEEE Transactions on Software Engineering* 1986; **12**(12):1128–1138.
24. Parrish A, Zweben SH. Analysis and refinement of software test data adequacy properties. *IEEE Transactions on Software Engineering* 1991; **17**(6):565–581.