Using UML Collaboration Diagrams for Static Checking and Test Generation

Aynur Abdurazik and Jeff Offutt *

George Mason University, Fairfax VA 22030, USA

The Third International Conference on the Unified Modeling Language (UML '00), pages 383–395, York, UK, October 2000.

Abstract. Software testing can only be formalized and quantified when a solid basis for test generation can be defined. Tests are commonly generated from program source code, graphical models of software (such as control flow graphs), and specifications/requirements. UML collaboration diagrams represent a significant opportunity for testing because they precisely describe how the functions the software provides are connected in a form that can be easily manipulated by automated means. This paper presents novel test criteria that are based on UML collaboration diagrams. The most novel aspect of this is that tests can be generated automatically from the software design, rather than the code or the specifications. Criteria are defined for both static and dynamic testing of specification-level and instance-level collaboration diagrams. These criteria allow a formal integration tests to be based on high level design notations, which can help lead to software that is significantly more reliable.

1 Introduction

There is an increasing need for effective testing of software for safety-critical applications, such as avionics, medical, and other control systems. In addition, the growing application areas of web applications and e-commerce require software that exhibits more reliability than most traditional application areas. Without software that functions reliably, business who operate on the web will lose money, sales, and customers. This paper presents results from an ongoing project to improve the ability to test software with high reliability requirements by developing techniques for generating test cases from formal descriptions of the software. Previous work [7, 8] has focused on generating tests from specifications; the current work focuses on design descriptions. Design descriptions represent a significant opportunity for testing because they precisely describe how the software functions behave in a form that can easily be manipulated by automated means.

Generating test data from high level design notations has several advantages over code-based generation. Design notations can be used as a basis for

* This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111 and in part by Rockwell Collins, Inc.

output checking, significantly reducing one of the major costs of testing. The process of generating tests from design will often help the test engineer discover problems with the design itself. If this step is done early, the problems can be eliminated early, saving time and resources. Generating tests during design also allows testing activities to be shifted to an earlier part of the development process, allowing for more effective planning and utilization of resources. Another advantage is that the test data is independent of any particular implementation of the design.

This paper presents a model for performing static analysis and generating test inputs from Unified Modeling Language (UML) collaboration diagram specifications. This work follows from previous work in developing test cases from UML statecharts [7]. Each class can have a statechart that describes the behavior of the instances of the class. The statechart models the reactions of the class to events received from the environment. The reactions to an event include sending new events to other objects and executing internal methods defined by the class. The previous paper adapted existing specification-based testing criteria [8] to UML statecharts. Our current research is focused on analyzing the behavior of a set of interacting objects. Interactions can be described using UML collaboration diagrams, and this paper presents a novel approach to generating tests data to check aspects of the software that are represented by the collaboration diagrams.

The UML is a language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. The UML does so by providing a collection of views to capture different aspects of the system to be developed. Examples of these views include *use cases* to capture user requirements, *class diagrams* to capture the static structure of objects, *collaboration* and *sequence diagrams* to capture dynamic interactions between objects and systems, and *package* and *deployment* views that organize design elements.

This paper uses collaboration diagrams as a source for software tests. A collaboration diagram consists of objects and associations that describe how the objects communicate. An interaction occurs when two or more objects are used together to accomplish one complete task. A collaboration diagram shows an interaction organized around objects that participate in the interaction and their links to each other. Collaboration diagrams provide the following six pieces of information [3]:

1. The objects that are involved in an interaction and the structure of these objects.
2. Instances of allowable sequences of operation calls to an object.
3. The semantics of an operation.
4. The operations that are imported from other classes, thus enabling a collaboration with objects of the other class.
5. The communication pattern of objects in a collaboration (synchronous or asynchronous).
6. The execution characteristics of objects (parallel or sequential).

This information must be preserved during the transformation of a specification into an implementation. It is possible and necessary to formulate test requirements from the above points, and generate tests from the collaboration diagrams. In particular, it is suitable to consider collaboration diagrams for integration testing since collaboration diagrams specify the interactions among a set of objects.

The remainder of the paper is organized as follows. The next section gives a brief overview of software testing, and Section 3 reviews collaboration diagrams. Section 4 describes a general test case generation mechanism to generate test cases from collaboration diagrams. The paper concludes by discussing to what extent collaboration diagrams can be used in testing and suggests some future work.

2 Software Testing

Software testing includes executing a program on a set of test cases and comparing the actual results with the expected results. Testing and test design, as parts of quality assurance, should also focus on fault prevention. To the extent that testing and test design do not prevent faults, they should be able to discover symptoms caused by faults. Finally, tests should provide clear diagnoses so that faults can be easily corrected [1].

Test requirements are specific things that must be satisfied or covered during testing, for example, reaching statements are the requirements for statement coverage. *Test specifications* are specific descriptions of test cases, often associated with test requirements or criteria. For statement coverage, test specifications are the conditions necessary to reach a statement. A *testing criterion* is a rule or collection of rules that impose test requirements on a set of test cases. A *testing technique* guides the tester through the testing process by including a testing criterion and a process for creating test case values.

Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied. There are various ways to classify adequacy criteria. One of the most common is by the source of information used to specify testing requirements and in the measurement of test adequacy. Hence, an adequacy criterion can be specification-based, design-based, or program-based.

A *specification-based* criterion specifies the required testing in terms of identified features of the specifications of the software, so that a test set is adequate if all the identified features have been fully exercised. Here the specifications are used to produce test cases, as well as to produce the design. A *program-based* criterion specifies testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised. For example, if the criterion of branch testing is used, the tests are required to cover each branch in the program. A *design-based* criterion specifies the required testing in terms of design components and interactions among them, so that a test set is adequate if all the components and interactions have

been exercised. Here the designs are also used to produce test cases, as well as to produce the program.

Design-based testing offers many advantages in software testing. The (formal) design of a software product can be used as a guide for designing system-level, integration-level, and class-level tests for the product. It is possible for the designer to use the design notation to precisely define fundamental aspects of the software's behavior, while more detailed and structural information is omitted. Thus, the tester has the essential information about the software's behavior without having to extract it from inessential details. This research assumes that the design is a valid representation of the system's desired behavior. The tests we generate will primarily evaluate whether the implementation correctly reflects the design.

Formal design descriptions provide a simpler, structured, and more formal approach to the development of tests than non-formal designs do. One significance of producing tests from designs is that the tests can be created earlier in the development process, and be ready for execution **before** the software is finished. Additionally, when the tests are generated, the test engineer will often find problems in the design, allowing the design to be improved before the program is implemented.

3 Collaboration Diagrams

In object-oriented software, objects interact to implement behavior. This interaction can be described in two complementary ways, one focused on individual objects and the other on a collection of cooperating objects. A state machine looks at each object individually. The collective behavior of a set of objects can be modeled in terms of how they collaborate.

A *collaboration* is a description of a collection of objects that interact to implement some behavior within a context. It contains *slots* that are filled by *objects* and *links* at run time. A collaboration slot is called a *role* because it describes the purpose of an object or link within the collaboration [6].

A Collaboration consists of *ClassifierRoles*, *AssociationRoles*, and *Interactions* [6]. A *ClassifierRole* defines a role to be played by an *Object* within a collaboration. An *AssociationRole* defines the relationships of a *ClassifierRole* to other roles. *AssociationRole* is a subset of existing *Links*. A *Link* is an individual connection among two or more objects, and is an instance of an *Association*. The objects must be direct or indirect instances of the classes at corresponding positions in an association. An *association* is a relationship among two or more specified *classifiers* that describes connections among their instances. The participating classifiers have ordered positions within the association.

An *interaction* is a behavioral specification that is composed of a sequence of communications among a set of objects within a collaboration to accomplish a specific purpose. Each interaction contains a partially ordered set of messages. A *message* is a specification of a stimulus, in other words, communication between a sender and a receiver. The message specifies the roles played by the sender

object and the receiver object and it states which *operation* should be applied to the receiver by the sender. A *stimulus* is a communication between two objects that either causes an operation to be invoked or an object to be created or destroyed.

An *operation* is a specification of a transformation or query that an object may be told to execute. It has a name and a list of parameters. A *method* is a procedure that implements an operation. It has an algorithm or procedure description.

A collaboration diagram is a graphical representation of a collaboration. The objects in a collaboration diagram are instances of classes in a class diagram. Without the interaction part, a collaboration diagram is similar to a class diagram. However, they are not the same [9]. For a collaboration, there need not be an object of every class, because some classes will be irrelevant to the particular collaboration being considered. There may be two or more different objects of the same class.

A collaboration diagram has two forms. A *specification level* collaboration diagram shows *ClassifierRoles*, *AssociationRoles*, and *Messages*, where an *instance level* collaboration diagram shows *Objects*, *Links*, and *Stimuli*. The following subsection introduces *specification level* and *instance level* collaboration diagrams separately, and explores their characteristics for test case generation.

3.1 Specification and Instance Level Collaboration Diagrams

Specification level collaboration diagrams show the roles defined within a collaboration. The diagram contains a collection of class boxes and lines corresponding to *ClassifierRoles* and *AssociationRoles* in the *Collaboration*. The arrows attached to the lines map onto *Messages*. Figure 1 is a specification level collaboration diagram that is adapted from the *Unified Modeling Language Specification 1.3* [6]. Graphically, a ClassifierRole uses a class symbol, which is a rectangle. The syntax of the name of a ClassifierRole is: `'/' ClassifierRoleName ':' ClassifierName [',' ClassifierName]*`

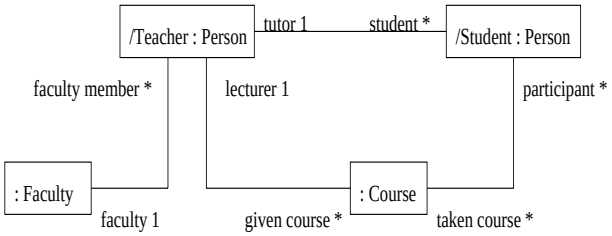

Fig. 1. Collaboration Diagram without Interaction

By examining the collaboration diagram, we can derive the following constraints: (1) `Teacher` and `Student` objects are instances of `Person` class, but

they have different properties, (2) `Student` object cannot be a `Faculty`, (3) each student must have one tutor, and (4) each course must have one lecturer. The implementation can be tested according to these constraints. For example, a `Student` cannot be a `Faculty`. A `Course` cannot have more than one `Lecturer`; a `Student` should have a `Tutor`; a `Student` cannot have more than one `Tutor`, etc.

An instance level collaboration diagram shows the collaboration of the instances of *ClassifierRoles* and *AssociationRoles*. It also includes instances of *Messages* that communicate over the *AssociationRole* instances. An object playing the role defined by a *ClassifierRole* is depicted by an object box.

Figure 2 is an example of an Instance Level Specification Diagram. In this diagram, the arrows point along the line in the direction of the receiving *Object*. The arrowheads have variations that may be used to show different kind of communication. The arrowheads shown in this diagram, which are stick arrowheads, indicate flat flow of control. Each arrow show the progression to the next step in sequence. Normally all of the messages are asynchronous. Several stereotypes are attached to the links: <<parameter>> indicates method parameter; <<local>> means local variable of a method.

Specification and instance level collaboration diagrams describe the structural relationships among the participants of a collaboration and their communication patterns. A realization is a relationship between a specification and its implementation. When collaboration diagrams are used to describe the realization of use cases, they describe only the externally visible actions and their sequences. When collaboration diagrams are used to describe the realization of an operation on an object, they provide more specific information, such as (1) parameters to the operation and their usage, (2) characteristics of participating variables, e.g. local or global, (3) constraints over associations, and (4) construction and/or destruction of objects during the process.

Since collaboration diagrams include both the messages that are passed between objects and their sequences, collaboration diagrams provide both design level data flow and design level control flow information. Traditionally, data flow and control flow information are obtained from the source code. Data flow and control flow information have had significant impact on testing [5, 10, 11]. Hence, using collaboration diagrams in testing helps us in many ways. Obvious benefits are (1) generating test data using existing data flow and control flow testing techniques before the code generation, (2) static checking of specification itself, and (3) static checking of code.

4 Testing Criteria

Collaboration diagrams describe the structure and behavior of the system. A UML collaboration specifies what requirements must be fulfilled by the objects in a system, and what communications must take place between the objects for a specific task to be performed. UML collaboration diagrams are used at different specification abstraction levels. For example, they describe the realization

of use cases, which capture high level system functionality. UML collaboration diagrams also specify the implementation of a class operation as an interaction [6, 4]. Combined specifications, e.g., use cases with collaboration diagrams or class diagrams with collaboration diagrams, can be use to statically check the code and test the running application.

The following subsections explain the above idea in more detail. Subsection 4.1 gives some definitions that will be used in defining testing criteria. Subsection 4.2 analyzes the static checking opportunities within collaboration diagrams. Subsection 4.3 gives criteria for dynamic testing from collaboration diagrams.

4.1 Some Definitions

Figure 2 is a collaboration diagram for an Operation taken from a paper by Overgaard [9]. In this diagram, *processOrder(o)* is a specified operation whose implementation is modeled by the collaboration diagram. :Company is an object on which the specified operation *processOrder(o)* is called. Thus, *processOrder* belongs to the class Company. We can distinguish four types of messages in this diagram. A call to a method through an object that is defined in the calling object is a *local method invocation*. In Figure 2, `d` and `s` are local to :Company, `o` is a parameter to :Company, and `delivery` and `store` are not local. The four message types are: (1) local method invocations without return values, e.g., `deliver(d)` in Figure 2, (2) local method invocations with return values, (3) parameter object's method invocations, e.g., pNr := getpNr(), and (4) object creations, e.g. delivery(o,s). The messages, stereotypes on links, parameters, and sequence numbers that are described in the collaboration diagram are our testing objectives.

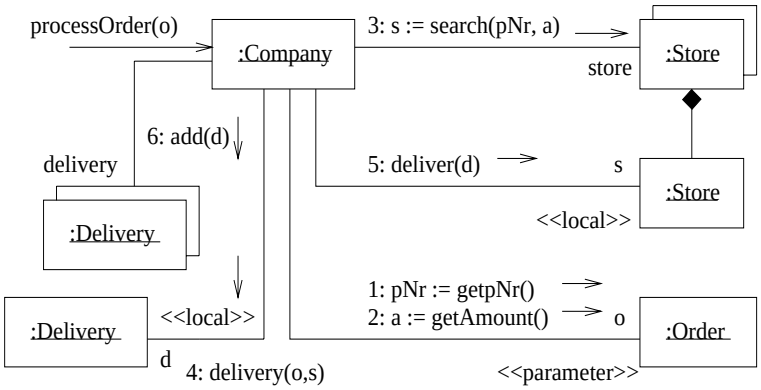

Fig. 2. A Collaboration Diagram for an Operation

The following terms will be used in the definition of testing criteria. Before defining the test criteria, we define *12* types of possible connections in collab-

oration diagrams. The types of connection pairs, or links, are reminiscent of data flow definitions [5], but represent data interactions that occur at a higher level (design) than traditional data flow (intra-procedural). They are classified based on information from class diagrams and collaboration diagrams, but once defined, only collaboration diagrams are needed to generate tests.

A *collaborating pair* is a pair of ClassifierRoles or AssociationRoles that are connected via a link in the collaboration diagram. The ClassifierRole part describes the type of Object with a set of required operations and attributes.

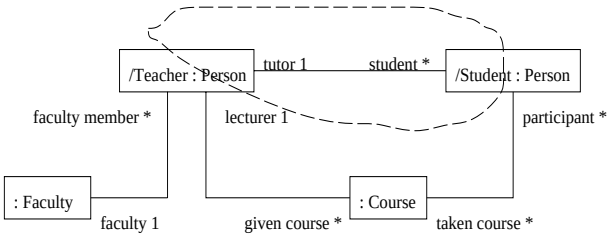

Fig. 3. Collaborating Pairs

A *variable definition link* is a link whose stimulus has a return value that is assigned to local variable. A *variable usage link* is a link whose argument-list of the stimulus includes a local variable. An *object definition link* is a stimulus that returns a reference to an object that is accessible to the target object. An *object creation link* is a link whose stimulus invokes the constructor of the class of target CollaborationRole. An *object usage link* is a link whose stimulus invokes a method (other than constructor and destructor) of a locally accessible object. An *object destruction link* is a link whose stimulus invokes the destructor of a locally accessible object.

Sometimes we are concerned with pairs of links. A *variable def-use link pair* is a pair of links in which a variable is first defined and then used. An *object def-use link pair* is a pair of links in which an object is first defined and then used. An *object creation-usage link pair* is a pair of links in which an object is first created and then used. An *object usage-destruction link pair* is a pair of links in which an object is first used and then destroyed. A *message sequence path* is a path that includes all messages in a collaboration in the order specified by the collaboration diagram.

Variable definition-usage link pairs, object creation-usage link pairs, object usage-destruction link pairs, and object creation-destruction link pairs can be used to statically check the code. Message sequence paths can be used to generate test cases. The following subsections give requirements for static checking and testing. The first subsection describes what should be checked statically. The second subsection describes test data that can be derived from a collaboration diagram.

4.2 Static Checking

Testing can be either static or dynamic. Most discussions of testing focus on dynamic testing, wherein the software is executed on some inputs. Static testing refers to checking some aspects of the software without execution, usually by evaluating the source code.

Collaboration diagrams provide constraints on a system. For example, a collaboration diagram for an operation describes specific information that is needed for this operation to be accomplished, such as return values for an invoked method during the process, parameter types, etc. The pairs that were defined in subsection 4.1 can also also be considered as constraints that cannot be violated.

We have identified four items that should be used to statically check the code. They are described as follows:

1. **ClassifierRoles:** In a collaboration, if two ClassifierRoles originate from the same base class, then they should be distinct in terms of their requested operations and values of attributes. Since they originate from the same class, it is possible that they might be mistaken for each other. For this reason, ClassifierRoles that originate from the same class should be tested to see if they have all the required attributes and operations.
2. **Collaborating Pairs:** The links on a collaboration diagram depict the structural constraints of collaborating pairs and their communication messages. The association tells if it is a one-to-one, one-to-many, or many-to-many relationship. The relationship reflects a constraint in the requirement specification. Hence, testing the structural relationship between objects can serve as verification of requirements. Each collaborating pair on the collaboration diagram should be checked or tested at least once.
3. **Message or stimulus:** Messages provide information on:
   - return value type
   - thread of control
   - operation or method name to be invoked on the target object
   - parameters to the invoked operation or method

   Testing of a message itself may reveal most integration problems. A stimulus is an instance of a message. A stimulus could be a signal, a method or operation call, or an event that causes an object to be created or destroyed. Besides carrying a specific data, messages have a direction. That is, a stimulus originates from a source object (*sender*) and resides on a target object (*receiver*). We can see that stimulus provides early information for integration testing, thus each stimulus should be used as a basis for generating test inputs.
4. **Local Variable Definition-Usage Link Pairs:** Checking variable definition-usage link pairs lets us find data flow anomalies at the design level. The following link pairs should all be checked for data flow anomalies:
   - Global Variable Definition-Use Pairs
   - Object Creation-Use Pairs
   - Object Use-Destruction Pairs
   - Object Creation-Destruction Pairs

4.3 Dynamic Testing

Collaboration diagrams provide a complete path for a use case or the realization of an operation. For the purposes of dynamic testing, this paper focuses on the collaboration diagrams for the realization of operations. We assume that there is one collaboration diagram per operation, and the implementation of an operation conforms to the collaboration. Also, collaboration diagrams provide the information necessary to test object interaction.

Test criterion: *For each collaboration diagram in the specification, there must be at least one test case t such that when the software is executed using t, the software that implements the message sequence path of the collaboration diagram must be executed.*

Each collaboration diagram represents a complete trace of messages during the execution of an operation. Therefore, a message sequence path include all variable def-use link pairs, object def-use link pairs, object creation-usage link pairs, and object usage-destruction link pairs.

We can form a *message sequence path* by using the messages and their sequence numbers. *Message sequence paths* can be traces of system level interactions or component (object) level interactions. Figure 4 shows a message sequence path that is derived from Figure 2. In this diagram, nodes represent method calls and edges represent control flow. Each node has information about the operation name and the object name that the operation is implemented in.

Criteria of this type are normally used in one of two ways. The criterion can be used as a guide for generating tests, or the criterion can be used as a metric for measuring externally created tests. This paper focuses on the measurement use of the criterion; test generation is left for future work.

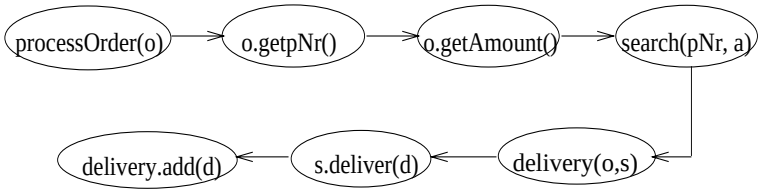

Fig. 4. A Message Sequence Path

To test that the system will produce an event trace that conforms to the message sequence path we derived from the collaboration diagram, we insert instrumentations into the original program. We assume that the type of instrumentation will not affect the performance of the application.

```
algorithm:          Instrument (ColDiagram, Implementation)
input:              A collaboration diagram that models the
                    implementation of an operation, and the
                    implementation of the system.
output:             Instrumented code.
output criteria:    All the nodes on the message sequence path
                    should be instrumented.
declare:            msgPathNode : {actionName, linkEndObjectTypeName,
                    next}
                    curNode is type msgPathNode. curNode represents
                    the current node in the message sequence path.
                    msgPath -- A linked list of type msgPathNode
                    objects.

Instrument (ColDiagram, Implementation)
BEGIN -- Algorithm Instrument
    construct a msgPathNode for each link and its link end object
    put each msgPathNode in msgPath linked list
    curNode = first node in the msgPath
    WHILE (curNode != null)
        className = curNode.linkEndObjectTypeName
        go to the actual implementation of className in the code
        actionName = curNode.actionName
        go to the actual implementation of actionName in the code
        insert instrument in actionName method
        curNode = curNode.next
    END WHILE
END Algorithm Instrument
```

Fig. 5. The Instrumentation Algorithm

Figure 5 gives an algorithm for instrumentation. The algorithm attempts to achieve the following goals:

1. insert instruments for each link
2. help the tester to keep track of the run-time interaction traces

This algorithm can be illustrated through an example. Consider the collaboration diagram in Figure 2, and its message sequence path in Figure 4. The algorithm will start with the collaboration diagram, then sort all its stimuli and their link end objects into a `msgPath` linked list. The algorithm picks the first node in the `msgPath`, {`processOrder(o), Company,` *next*}. From the information contained in the `msgPath` node, the algorithm decides to insert instrumentation in the implementation of the *processOrder()* method in the *Company* class. The instrumentation should reflect the parameter type. Then, the algorithm

takes the next node in `msgPath` and inserts the next instrumentation. The algorithm processes all the nodes in `msgPath` in the order of their message sequence numbers.

5 Conclusions

This paper has introduced new integration-level analysis and testing techniques that are based on design descriptions of software component interactions. There are relatively few formal testing criteria that are based on design descriptions. The techniques in this paper are innovative in that they utilize formal design descriptions as a basis, and have practical value because they can be completely automated from the widely used design notation of collaboration diagrams. Tools already exist for constructing collaboration diagrams, and tools for performing the analysis and testing described here can be built with relative ease. This paper also includes an algorithm for instrumenting a program that is implemented from collaboration diagrams. The instrumentation will ensure that tests satisfy the formal testing criteria developed in this research, and also help ensure traceability from the design artifacts to the code.

This research includes techniques to be used both statically and dynamically. The static analysis allows test engineers to find certain problems that reflect ambiguities in the design and misunderstandings on the part of the detailed designers and programmers. The dynamic technique allows test data to be generated that can assure reliability aspects of the implementation-level objects and interactions among them.

We have three major directions for the future of this work. First, we are currently carrying out an empirical evaluation of this work by implementing software that is based on collaboration diagrams, generating tests, and then using the tests to detect problems in the software. Second, we plan to implement tools to perform the static analysis on the software, and to automatically generate tests from UML collaboration diagrams. We plan to integrate these tools with Rational Software Corporation's Rational Rose tool [2]. This will allow us to carry out more extensive empirical evaluations, and to gain practical experience with using this technique. Finally, we continue to develop analysis and testing techniques that are based on the various design and specification diagrams that are part of the UML.

References

1. B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.
2. Rational Software Corporation. *Rational Rose 98: Using Rational Rose.* Rational Rose Corporation, Cupertino CA, 1998.
3. Gregor Engels, L. P. J. Groenewegen, and G. Kappal. Object-oriented specification of coordinated collaboration. In *Proceedings of the IFIP World Conference on IT Tools*, pages 437–449, Canberra, Australia, September 1996.

4. Gregor Engels, Roland Hucking, Stefan Sauer, and Annika Wagner. Uml collaboration diagrams and their transformation to java. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 473–488, Fort Collins, CO, October 1999. IEEE Computer Society Press.
5. P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
6. Object Management Group. *OMG UML Specification Version 1.3*, June 1999. Available at http://www.omg.org/uml/.
7. Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. IEEE Computer Society Press.
8. Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.
9. Gunnar Overgaard. A Formal Approach to Collaborations in the Unified Modeling Language. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 99–115, Fort Collins, CO, October 1999. IEEE Computer Society Press.
10. A. Spillner. Control flow and data flow oriented integration testing methods. *The Journal of Software Testing, Verification, and Reliability*, 2(2):83–98, 1992.
11. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

This article was processed using the LaTeX macro package with LLNCS style