Generating Test Sequences from UML Sequence Diagrams and State Diagrams

Dehla Sokenou
GEBIT Solutions GmbH
dehla.sokenou@gebit.de

Abstract: UML models offer a lot of information that should not be ignored in testing. By combining different UML components, different views of the program under test are used. The paper concentrates on a technique for generating test cases from a combination of UML sequence and state diagrams. The main information is extracted from sequence diagrams, which is complemented by initialization sequences for the participating objects derived from state diagrams.

1 Motivation

One of the main problems in testing object-oriented programs is test case selection. In most cases, it is impossible to stimulate the program with all data of the input domain. A pragmatic approach is to concentrate on typical message sequences as modeled using the sequence diagram. Testing based on sequence diagrams seems to be intuitive. Each sequence diagram specifies one test case or set of test cases. But normally, modeled sequences are incomplete, and offer no information about the time in the program's life cycle when the modeled behavior will occur nor state information about participating objects. A test based on sequence diagrams must consider these issues. Our approach uses state diagrams to obtain the required information. Each sequence diagram is considered as a set of test cases. An attached state diagram for each participating object defines its states. Each combination of initial state configurations defines at least one test case in the set.

The paper is organized as follows. We introduce the test case generation technique in Section 2. In Section 3, we compare our approach with related work. Finally, Section 4 summarizes the paper and gives an outlook on future work.

2 Test Case Generation

UML models can be interpreted differently depending on their application. Thus, the UML [UML04] defines a set of semantic variation points. For testing purposes, we determine the variation points and concentrate on sequence diagrams and state diagrams. We assume that models are consistent. If not, the test case generation will result in an error in most cases. It is not necessary to provide all models as input for test case generation.

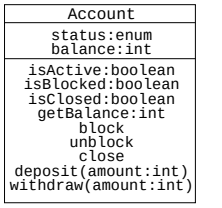

Figure 1: Static view of `Account`

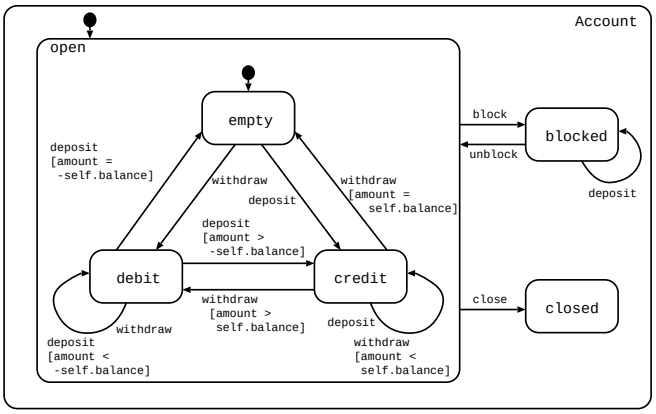

Figure 2: State diagram of class `Account`

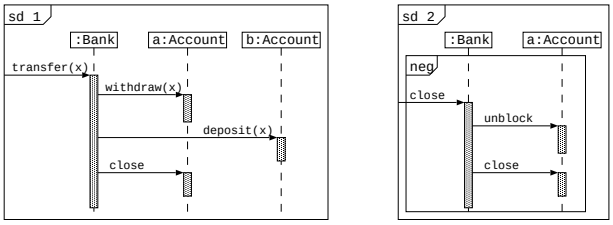

Figure 3: Positive and negative sequence diagram

We use UML protocol state machines for specifying object life cycles. Protocol state machines differ in some points from behavioral state machines, the other kind of state diagram in UML. Protocol state machines do not define actions, but it is possible to specify post-conditions of transitions instead. We assume that events in the state machine are method calls (call events). Methods referenced in the protocol state machine are update methods, all other methods being query methods. The set of states in which an update method triggers a transition defines an implicit precondition for these methods, meaning that the call of an update method in all other states violates the precondition. In sequence diagrams, we consider only messages in the form of method calls, too, focussing on synchronous communication between objects. Complex sequence diagrams with interaction operators like *alt* or *opt* are not considered, but in many cases they can be transformed to simple sequence diagrams. The only operator considered is the operator *neg*.

A bank account is used as example throughout the paper. Fig. 1 shows the static view and Fig. 2 the life cycle of an `Account` object, and Fig. 3 a positive and a negative scenario.

Test cases are generated for class and integration testing. Both use the same test case generation technique. The main information is extracted from sequence diagrams. Each sequence diagram defines a set of test cases differing in terms of the states of the participating objects. A test case is defined as a sequence and an initial object configuration.

For class testing, each participating object is considered separately. The modeled sequence is sent to the object under test. Objects that send messages to the object under test are replaced by test drivers. Objects that receive messages from the object under test are replaced by test stubs. In our example, sequence *sd 1* in Fig. 3, the sequence `withdraw` followed by `close` is sent to the `Account` object `a`. For `Account` object `b`, the scenario models only one message. In this case, the tester can decide if single message sequences are included in the class test cases. Integration testing means testing object collaborations on which a given sequence is executed. Here, we consider the whole scenario. The technique computes initialization sequences for each participating object if a state diagram for the object is attached. The last method called in an integration test case is the first message of the modeled sequence – here a call of `transfer` on the object of type `Bank`. The real sequence executed by the tested collaboration is not yet analyzed. We use a combination of state diagrams and OCL constraints as test oracle instead.

Our technique can be applied to positive and negative test cases. Since a negative scenario must be considered as a whole [Stö], we only derive test cases from negative sequences for integration testing. We distinguish between regular and complementary test cases. Regular test case are test cases where all participating objects are in a state in which the modeled sequence can be executed without violating the implicit preconditions specified by the protocol state machines. Complementary test cases are test cases where at least one of the participating objects is in a state in which the next called message violates the implicit precondition. Regular and complementaty test cases can be extracted from both positive and negative sequences. Even negative scenarios can be stimulated when all objects are in states in which the given sequence can potentially be executed, but the program must not respond with the modeled negative sequence.

To explain the technique, let us start with the `Account` object `a` in sequence diagram *sd 1*. We extract the method sequence sent to object `a`, in the example the sequence `a.withdraw(x); a.close`. This sequence can only be called in some of the states of object `a`, specified by the attached state diagram in Fig. 2. Looking at this state diagram, we can identify three states in which a call of `withdraw` does not violate the implicit precondition. These are all states where a call of `withdraw` triggers a transition, here states `credit`, `debit` and `empty`. All transitions triggered by `withdraw` lead to states where a call of the next message in the sequence, `close`, is possible. Thus, all states (`credit`, `debit`, `empty`) are included in the set of initial states for the given sequence. To initialize the test sequences given by the sequence diagram, we now derive a message sequence from the state diagram, leading to the desired state for all participating objects. Message sequences reference the complete transitions, including the call events and the preconditions (here: the relations between balance and amount) of the transitions.

For class testing, we first execute the initialization sequence and then the sequence from the sequence diagram. The resulting test cases are shown in Fig. 4. For integration testing, all objects are initialized using their intialization sequence. Here, we have to consider the second `Account` object `b` additionally. A test sequence consists of an initialization sequence of `a`, followed by an initialization sequence of `b` and ending with the first message call of the sequence diagram *sd 1*. Note that we have abstracted from the states of the bank to simplify the example. Negative scenarios are treated exactly the same. The only

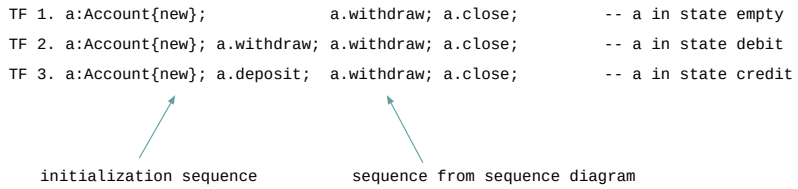

Figure 4: Regular positive test cases for class testing `Account`

difference is that after initializing the test sequence and sending the first modeled message from the sequence diagram the program must not respond with the modeled sequence.

The test system can generate complementary test cases as well. Complementary test cases consider states in which a sequence violates the implicit precondition. These are all states not included in the regular test cases. Generating complementary test cases can lead to state explosion. Generation of complementary test cases is therefore optional. We recommend generating only certain test cases where one of the participating objects is in a state violating the precondition, all other objects being initialized as in regular test cases.

3 Related Work

There are numerous state-based approaches in testing, too many to mention them all. We present three examples describing three strategies that are often applied. [KSG$^+$] describes a technique that traverses the state machine completely. Each transition in the state machine is executed at least once. In [SHS], transitions are selected in a nondeterministic manner. The test case set can be infinite, and cycles in the state machine can be executed more than once. A technique based on model checking is found in [EFM]. Test cases are generated based on counter examples computed by a model checker. In our approach, selection of transitions is based on the initialization of objects and the executed sequence.

Combinations of UML components for testing purpose are presented in various papers. In [FL], sequence diagrams are combined with pre- and postconditions for the referenced methods. The paper proposes three different techniques for initializing objects, but all of these involve initialization from outside, e.g. based on a database of objects. In [BL], the class diagram and the activity diagram are used additionally to the sequence diagram, but the state diagram is not considered. This approach focuses on system-level testing while our approach focuses on small units. In [BB], use case diagrams are combined with sequence and class diagrams. Like our approach, test sequences are derived from the sequence diagrams, but unlike our approach, category partitioning is used to define sets of test cases. Often, use cases are combined with other UML diagrams to generate test cases. Examples are found in [OA], where each use case is described by an attached collaboration diagram, and in [NFTJ06], where sequence diagrams and contracts are attached to each use case. An approach for transforming UML diagrams into diagrams according to the UML Testing Profile [UTP04] can be found in [DGNP].

4 Conclusion and Outlook

We have presented an approach that combines UML components for class and integration testing of object-oriented programs. The main information is extracted from sequence diagrams, which is complemented by the use of state diagrams. State diagrams have two functions: initialization of participating objects in a scenario and –in combination with OCL constraints– serving as a test oracle (not shown in this paper). Beyond the presented technique, we have developed the integration of the derived test oracles into the program under test using aspect-oriented programming techniques. Future work will focus on integrating other UML components into our test system. We are working on techniques to derive test data from UML models, mainly from OCL constraints. For test case generation, we are focusing on other interaction diagrams and activity diagrams. Currently, we are working on the integration of our technique into eclipse, based on the eclipse UML plugin, and an application of our technique on two case studies, an object-oriented and an aspect-oriented system. The evaluation will show if the presented technique is suitable for "real world" systems and if it can be applied to aspect-oriented systems, as well.

References

[BB] F. Basanieri and A. Bertolino. A Practical Approach to UML-Based Derivation of Integration Tests. In *QWE'2000*.

[BL] L. C. Briand and Y. Labiche. A UML-Based Approach to System Testing. In *UML'2001*.

[DGNP] Z. R. Dai, J. Grabowski, H. Neukirchen, and H. Pals. From Design to Test with UML - Applied to a Roaming Algorithm for Bluetooth Devices. In *TestCom'2004*.

[EFM] A. Engels, L. M. G. Feijs, and S. Mauw. Test Generation for Intelligent Networks Using Model Checking. In *TACAS'1997*.

[FL] F. Fraikin and T. Leonhardt. SeDiTeC – Testing Based on Sequence Diagrams. In *ASE'2002*.

[KSG+] D. C. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On Object State Testing. In *COMPSAC'1994*.

[NFTJ06] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jézéquel. Automatic Test Generation: A Use Case Driven Approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.

[OA] J. Offutt and A. Abdurazik. Using UML Collaboration Diagrams for Static Checking and Test Generation. In *UML'2000*.

[SHS] D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts. In *PSI'2003*.

[Stö] H. Störrle. Assert, Negate and Refinement in UML 2 Interactions. In *Workshop on Critical Systems Development with UML at UML'2003*.

[UML04] *Unified Modeling Language Specification, Version 2.0*. OMG, 2004.

[UTP04] *UML 2.0 Testing Profile Specification, Version 1.0*. OMG, 2004.