Semiautomatic Test Case Generation Based on Sequence Diagrams

Yao-Cheng Lei Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
{lyc94,naiwei}@cs.ccu.edu.tw

Abstract*-This article describes a semiautomatic test case generation tool for Java classes based on UML sequence diagrams. Sequence diagrams are used to specify dynamic behaviors among objects. This tool is developed to facilitate the semiautomatic generation of test cases in integration testing level. This tool automatically generates a suite of test paths from sequence diagrams. Given an input data and the corresponding expected output data for each test path from the user, this tool automatically generates a Java method that executes the test path. This tool generates Java test classes for the JUnit framework.*

Keywords: UML, sequence diagrams, test case generation.

1. Introduction

Quality software is still very hard to achieve in general [8]. To be able to consistently develop quality software involves several factors: software processes, software personnel, software resources, software project management, and so on. Among these, software testing activity in software processes remains one of the main activities to assure software quality. Software testing is a systematic attempt to find faults in the software systems. A failure of a software system is any deviation of the observed behavior from the specified behavior. A fault of a software system is the cause of a failure of the system. A test case is a pair of input and expected output that can be used to detect faults by revealing failures caused by faults. The goal of software testing is to use a suite of test cases to maximize the number of faults detected in order to assure the quality of the software system. The costs of software testing usually take about half of the entire software development costs. However, the software testing techniques and tools are still immature so that quality software is still very hard to achieve in general.

There are two types of software testing techniques: black-box testing and white-box testing [2]. The black-box testing techniques are based on the functional specifications and focus on the coverage of the specified external behaviors of the software system. The white-box testing techniques are based on the source code and focus on the coverage of the internal structure of the source code. These two types of testing techniques are complementary. The white-box testing techniques alone may fail to reveal that some portions of the specifications are missed in the source code. On the other hand, the black-box testing techniques alone may fail to reveal that some portions of the source code are not contained in the specifications. Therefore, employing both types of testing techniques is necessary to assure quality software.

Software testing needs to be applied in the different levels: unit testing, integration testing, and system testing [2]. Unit testing finds faults within a single software component by isolating it from the remaining components of the system. Integration testing finds faults among interacting components according to specified use cases. System testing finds faults of the external behaviors of the entire system. Executing test cases on a single software component or a group of software components requires the components under test to be isolated from the rest of the system. A test driver simulates the part of the system that calls the components under test. A test driver passes the test inputs to the components under test and displays the actual outputs. A test stub simulates components that are called by the components under test. The test stubs must provide the same interfaces as the called components. A test oracle generates the expected outputs for the test inputs and checks if the expected outputs are the same as the actual outputs. At present, test input, drivers, stubs, and oracles are usually generated manually by programmers or testers.

The Unified Modeling Language (UML) is a visual modeling language that can be used to specify, visualize, construct, and document the artifacts of a software system [7]. The UML has been widely used as the functional specifications of object-oriented software systems. The UML provides several types of diagrams to model different aspects of software systems. For example, use case diagrams are used to specify the functional behavior of the system from the viewpoint of users. Class diagrams are used to describe the structure of the system in terms of objects, classes, attributes, operations, and their associations. State diagrams are used to specify the dynamic behavior of an individual object as a number of states and transitions between these states. Sequence diagrams are used to formalize the dynamic behavior of the system and to visualize the communication among objects. The UML diagrams have also been widely used to facilitate the analysis and design of object-oriented software systems [3].

Because of the high cost of software testing, automation of software testing is a crucial issue. This article describes a semiautomatic test case generation tool for Java classes based on UML sequence diagrams. Sequence diagrams are used to specify dynamic behaviors among objects. This tool is developed to facilitate the semiautomatic generation of test cases in integration testing level. This tool automatically generates a suite of test paths from sequence diagrams. Given an input data and the corresponding expected output data for each test path from the user, this tool automatically generates a Java method that tests the test path. This tool generates Java test classes for the JUnit framework [1].

The remainder of this article is organized as follows. Section 2 gives a brief introduction to UML sequence diagrams. Section 3 provides an overview of our semiautomatic test case generation tool. Section 4 describes the generation of test paths based on sequence diagrams. Section 5 describes the generation of Java test classes based on test paths and user-provided test data. Second 6 reviews related work. Finally, conclusions are given in Section 7.

2. Sequence diagrams

A sequence diagram is used to model the interaction of a set of objects. Figure 1 is an example of a sequence diagram. This example displays a use case of a store system. A sequence diagram $D = <O, M>$ consists of a set of objects O and a sequence of messages M. Each message $m = <o_1, o_2, f, a> \in M$ represents a sequential flow of control from one object o_1 to another object o_2. Sequential flow of control can be activated via several means. This article only focuses on method invocations and returns. A message $<o_1, o_2, f, a>$ is a method invocation $f(a)$ from o_1 to another object o_2 with arguments a, or a method return from o_2 to another object o_1 with return value a.

Sequence diagrams may have structured control constructs that specify more complex flow of control. This article considers only three structured control constructs: opt, alt, and loop. Each structured control construct is represented as a fragment (or a rectangular box). An opt construct has a guard condition. The sequence of messages in an opt construct is executed if the guard condition is true; Otherwise, it is omitted. An alt construct has two or more subfragments, each of them has a guard condition. If more than one guard condition is true, the sequence of messages in one of the subfragments is selected nondeterministically to execute. If none is true, no execution is consistent with the specification. A loop construct has a guard condition. The sequence of messages in a loop construct is executed repeatedly as long as the guard condition is true.

3. System architecture

Our test case generation tool consists of two components: a test path generator and a test class generator. The system architecture is shown in Figure 2.

The test path generator determines a set of complete paths in the sequence diagram for testing according to a test criterion. We have implemented two sets of test criteria. One set does not consider the usage profile. It includes all-node, all-edge, and all-path coverage. The other set considers the usage profile. It includes usage-based all-node and all-edge coverage. The usage profile is provided by the user.

The test class generator generates a Java method for each test path. Each Java method acts as a test driver that ensures the execution of the test path and verifies its correctness. The input test data and expected output data are provided by the user.

The test path generator can generate the set of constraints that ensures the execution of a test path. The user needs to determine the input test data and expected output data for this test path according to

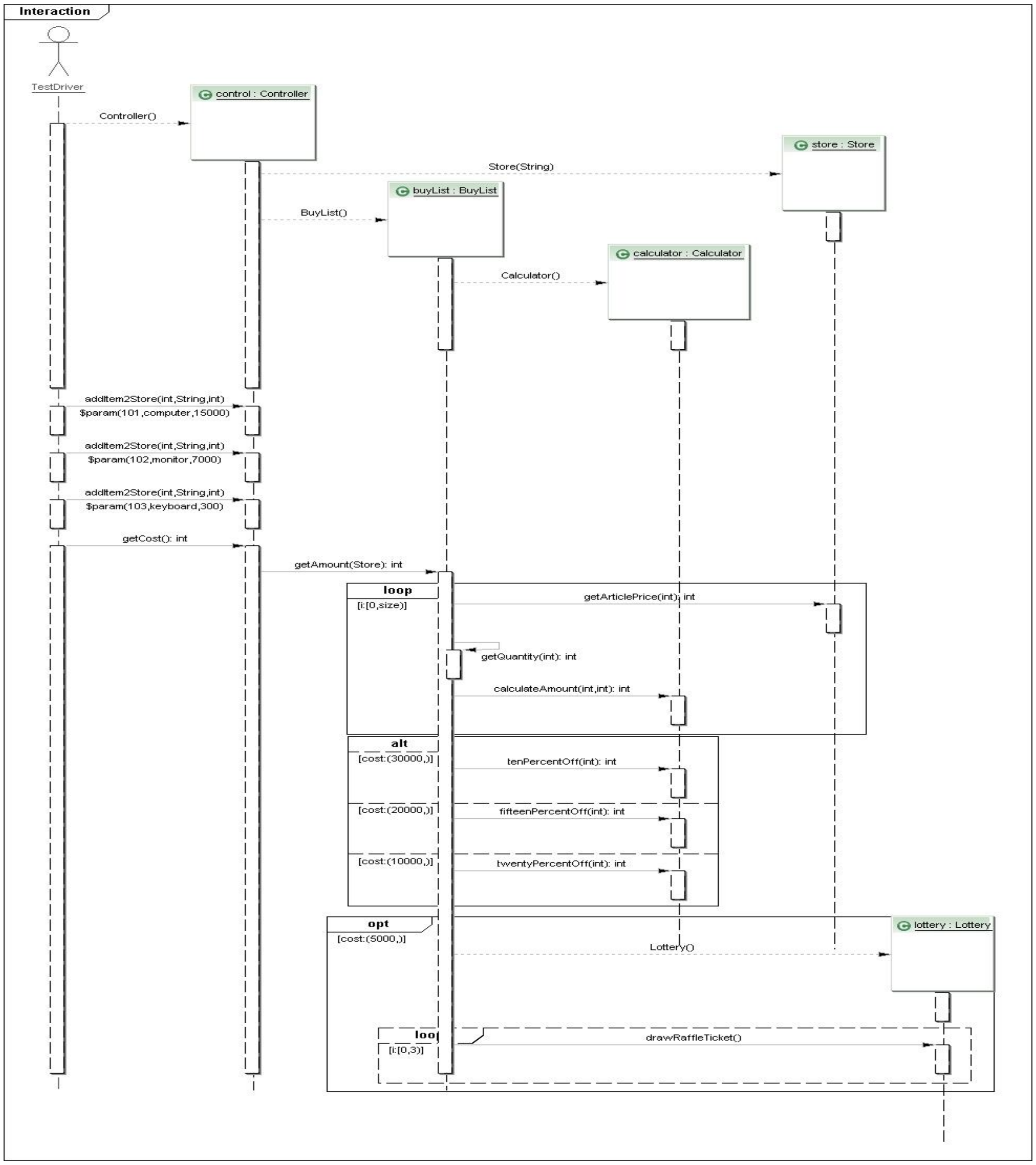


Figure 1. An example sequence diagram.

the set of constraints. The determination of the input test data and expected output data for a test path is the step that is not automatic. The automation of this step is our future work.

4. The test path generator

The generation of test paths consists of two steps. The first step converts the sequence diagram into a message flow graph. The message flow graph defines all possible execution sequences of messages. The message flow graph has a unique entry node and a unique exit node. The other nodes in the message flow graph may be a message node or a control node. A message node represents the execution of a message. Each message in the sequence diagram has a corresponding message node in the message flow graph. A control node may be a branch node or a joint node. A branch node contains a Boolean expression and two outgoing edges. The value of the Boolean expression will determine the outgoing edge

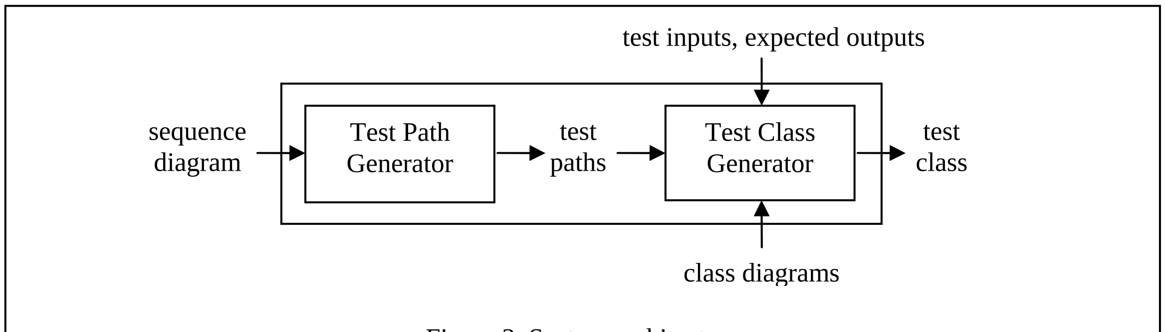


Figure 2. System architecture

control will flow. The joint node merges flows from multiple incoming edges into one outgoing flow.

If the sequence diagram does not contain structured control constructs, the message control flow graph consists of a single path. Each structured control construct is converted to some branch nodes and a joint node. The first branch node is the entering node of the structured control construct and the joint node is the exiting node of the structured control construct.

For each opt construct in sequence diagram, there are a corresponding branch node and a joint node in the message flow graph. The guard

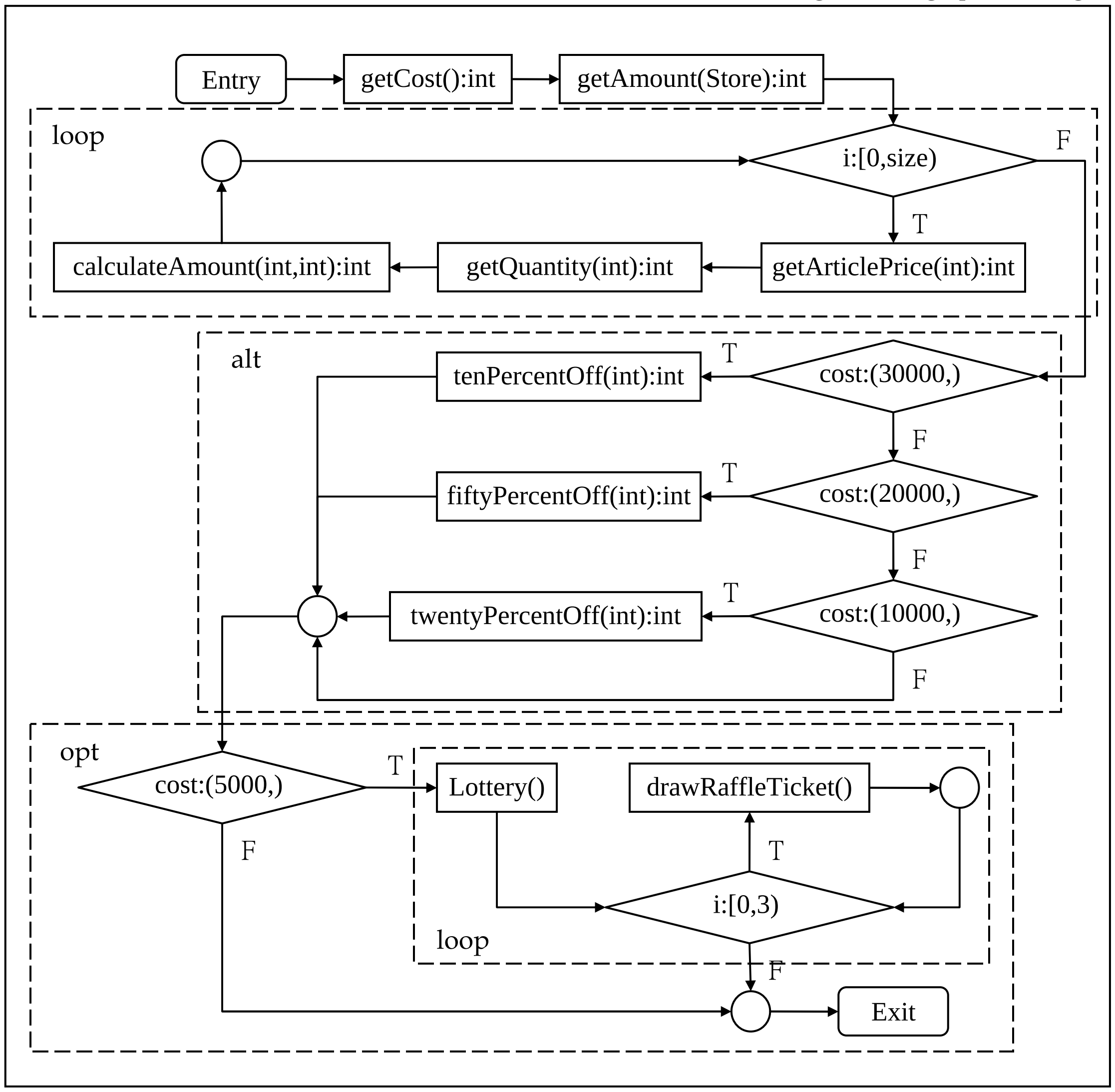


Figure 3. The message flow graph corresponding to the sequence diagram in Figure 1.

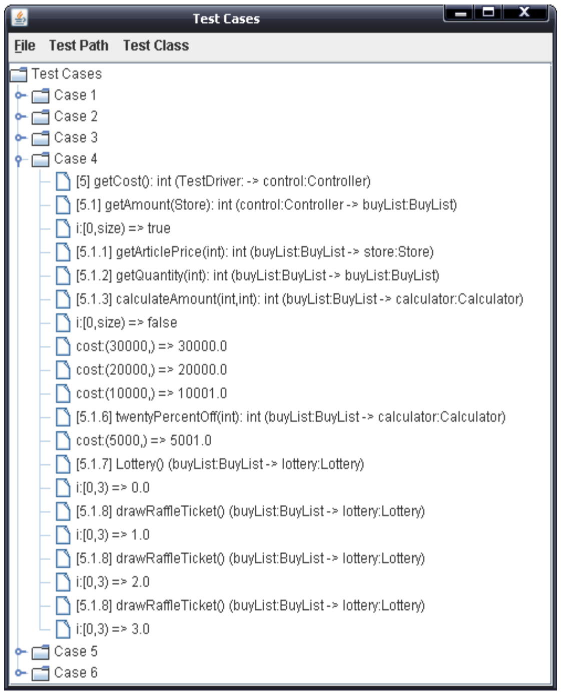
condition in the opt construct is the Boolean expression in the branch node. The true outgoing edge of the branch node connects to the first message in the opt construct. The last message in the opt construct connects to the joint node. Besides, the false outgoing edge of the branch node connects to the joint node.

For each alt construct with *n* subfragments in sequence diagram, there are *n* corresponding branch nodes and a joint node in the message control flow graph. For each subfragment, its guard condition is the Boolean expression in the corresponding branch node. The true outgoing edge of the branch node connects to the first message in the subfragment. The last message in the subfragment connects to the joint node. Besides, the false outgoing edge of the branch node connects to the branch node of the next subfragment except that the false outgoing edge of the branch node for the last subfragment connects to the joint node.

For each loop construct in sequence diagram, there are a corresponding branch node and a joint node in the message flow graph. The guard condition in the loop construct is the Boolean expression in the branch node. The true outgoing edge of the branch node connects to the first message in the loop construct. The last message in the loop construct connects back to the branch node. Besides, the false outgoing edge of the branch node connects to the joint node.

The message flow graph for the sequence diagram in Figure 1 is shown in Figure 3. Since the sequence diagram has four structured control construct (an opt construct, an alt construct, and two loop constructs), the message flow graph has four joint nodes. There is one branch node for each opt or loop construct and there are three branch nodes for alt construct.

The second step of the test path generation generates a set of complete paths of the sequence flow graph based on a test coverage criterion. We support five kinds of test coverage criteria: all-node, all-edge, all-path, usage-based all-node, and usage-based all-edge. We use a depth-first traversal to generate complete paths. The edges are selected using a fixed order for all-node and all-edge criteria. The edges are selected based on usage profile for usage-based all-node and usage-based all-edge criteria. The GUI for displaying the set of test paths for the message flow graph in Figure 3 using all-node coverage criterion is shown in Figure 4. There are a total of six test paths. The complete path for each of these test paths can be shown. In Figure 4, the fourth path is fully shown.



```
Test Cases
  Case 1
  Case 2
  Case 3
  Case 4
    [5] getCost(): int (TestDriver: -> control:Controller)
    [5.1] getAmount(Store): int (control:Controller -> buyList:BuyList)
    i:[0,size) => true
    [5.1.1] getArticlePrice(int): int (buyList:BuyList -> store:Store)
    [5.1.2] getQuantity(int): int (buyList:BuyList -> buyList:BuyList)
    [5.1.3] calculateAmount(int,int): int (buyList:BuyList -> calculator:Calculator)
    i:[0,size) => false
    cost:(30000,) => 30000.0
    cost:(20000,) => 20000.0
    cost:(10000,) => 10001.0
    [5.1.6] twentyPercentOff(int): int (buyList:BuyList -> calculator:Calculator)
    cost:(5000,) => 5001.0
    [5.1.7] Lottery() (buyList:BuyList -> lottery:Lottery)
    i:[0,3) => 0.0
    [5.1.8] drawRaffleTicket() (buyList:BuyList -> lottery:Lottery)
    i:[0,3) => 1.0
    [5.1.8] drawRaffleTicket() (buyList:BuyList -> lottery:Lottery)
    i:[0,3) => 2.0
    [5.1.8] drawRaffleTicket() (buyList:BuyList -> lottery:Lottery)
    i:[0,3) => 3.0
  Case 5
  Case 6
```

Figure 4. Test paths.

The format for message node is as follow:

[*method number*] *method name* (*parameter type*) (*caller object* --> *callee object*)

The format for branch node is as follow:

variable name : {*condition*} ==> *value*

5. The test class generator

The generation of test class consists of three steps. The first step reads in class diagrams for objects in the sequence diagram. The second step asks the user to input three kinds of information: object initialization, test input data and expected output data for each test path. The object initialization includes information for creating objects into appropriate states right before the invoking of the testing method. The test input data include parameters of the testing method. The expected output data include the expected returned value of the testing method and expected updates of objects' data members. The information in class diagrams is used to facilitate the input of the three kinds of information in step 2. For example, method names can be selected from menu instead of being typed in.

The third step of the test class generation generates a Java class for testing the set of test paths. Each method in the class tests a test path. Given the information in step 2, the generation of Java class is straightforward. The generated Java

```
import junit.framework.TestCase;
import storesystem.Controller;


public class TestStoreSystem extends TestCase {

        // Objects
        private Controller control;

        private void setup() {

                // Path-independent initialization
                control = new Controller();
                control.addItem2Store(101, "computer", 15000);
                control.addItem2Store(102, "monitor", 7000);
                control.addItem2Store(103, "keyboard", 300);
        }

        public void testControllergetCost4() {

                // Object initialization
                setup();
                // Path-dependent initialization
                control.addItem2BuyList(101, 2);
                control.addItem2BuyList(102, 1);

                // Invoke testing method
                int results = control.getCost();

                // Check expected output and actual output
                assertEquals(37000, results);
        }
}
```

Figure 5. Java test class.

class for our running example is shown in Figure 5. Due to space limitation, only the code for the fourth test path is shown.

6. Related work

Fraikin and Leonhardt studied the conditions under which sequence diagrams are testable [4]. The seven conditions proposed by them should be satisfied by most sequence diagrams. They also developed a test tool for generating test programs. Their tool also needs the user to input test input data and expected output data. They didn't consider structured control constructs in their work.

Javed et. al. developed a tool for generating Java test programs from sequence diagrams based on model driven architecture [5]. They first convert the sequence diagram into a xUnit model. They then transform xUnit into JUnit code. We propose to use message flow graphs as a model for generating test paths.

7. Conclusions

This article has described a semiautomatic test case generation tool for Java classes based on UML sequence diagrams. This tool is developed to facilitate the semi-automatic generation of test cases in integration testing level. This tool automatically converts a sequence diagram into a message flow graph, and then generates a suite of test paths from the message flow graph based on various test criteria.

Given an input data and the corresponding expected output data for each test path from the user, this tool automatically generates a Java method that tests the test path. This tool generates Java test classes for the JUnit framework. The most difficult jobs in test automation are test input generation and expected output generation. Our future work includes using constraint solving techniques to automate the generation of test input and using abstract state machine language to automate the generation of expected output.

References

[1] K. Beck and E. Gamma, JUnit Cookbook, http://junit.sourceforge.net/.

[2] B. Bezier, Software Testing Techniques, 2nd Edition, Van Nostrand, 1990.

[3] M. Blaha and J. Rumbaugh, Object-Oriented Modeling and Design with UML, 2nd Edition, Pearson Prentice Hall, 2005.

[4] F. Fraikin, and T. Leonhardt, "SeDiTeC - Testing Based on Sequence Diagrams," Proceedings of the 17th IEEE international conference on Automated software engineering. 261-266, 2002.

[5] A. Z. Javed, P. A. Strooper, and G. N. Watson, "Automated Generation of Test Cases Using Model-Driven Architecture," Proceedings of the Second International Workshop on Automation of Software Test. 3. 2007.

[6] Object Technology International Incorporation, Eclipse Platform Technical Overview, 2003.

[7] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, 2nd Edition, Addison-Wesley, 2005.

[8] Standish Group, 2003 CHAOS Report, 2004.