e-Informatica Software Engineering Journal, Volume 2, Issue 1, 2008

A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams

Philip Samuel*, Rajib Mall*

*Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur(WB),India-721302

philips@cusat.ac.in, rajib@cse.iitkgp.ernet.in

Abstract

We present a novel methodology for test case generation based on UML sequence diagrams. We create message dependence graphs (MDG) from UML sequence diagrams. Edge marking dynamic slicing method is applied on MDG to create slices. Based on the slice created with respect to each predicate on the sequence diagram, we generate test data. We formulate a test adequacy criterion named slice coverage criterion. Test cases that we generate achieves slice coverage criterion. Our approach achieves slice test coverage with few test cases. We generate effective test cases for cluster level testing.

1 Introduction

Ever since Weiser [51] introduced program slicing, researchers have shown considerable interest in this field probably due to its application potential. Slicing is useful in software maintenance and reengineering [15, 35], testing [19, 28, 42], decomposition and integration [23], decompilation [10], program comprehension [38, 20], and debugging [39]. Most of the works reported on slicing concerns improvements and extensions to algorithms for slice construction [37, 21, 33, 14, 6]. Even though dynamic slicing is identified as a powerful tool for software testing [33, 42], reported work on how dynamic slicing can be used in testing is rare in the literature. In 1993, Kamkar et al. [28] reported how dynamic slicing can be applied to interprocedural testing. This work is reported in the context of testing procedural code. To the best of our knowledge, no work is reported in the literature that describes how dynamic slicing can be used for test case generation in the object oriented context. In this paper, we propose a method to generate test cases by applying dynamic slicing on UML sequence diagrams.

As originally introduced, slicing (static slicing) considers all possible executions of a program. Korel and Laski [32] introduced the concept of dynamic slicing. Dynamic slicing considers a particular execution and hence significantly reduces the size of the computed slice. A dynamic slice can be thought of as that part of a program that “affects" the computation of a variable of interest during a program execution on a specific program input [33]. A dynamic slice is usually smaller than a static slice, because run-time information collected during execution is used to compute the slice. In a later work, Korel has shown that slicing can be used as a reduction technique on specifications like UML state models

[34].

The goal of software testing is to ensure quality. Software testing is necessary to produce highly reliable systems, since static verification techniques suffer from several handicaps in detecting all software faults [5]. Hence, testing will be a complementary approach to static verification techniques to ensure software quality. As software becomes more pervasive and is used more often to perform critical tasks, it will be required to be of very high quality. Unless more efficient ways to perform effective testing are found, the fraction of development costs devoted to testing will increase to unacceptable levels [45].

The most intellectually challenging part of testing is the design of test cases. Test cases are usually generated based on program source code. An alternative approach is to generate test cases from specifications developed using formalisms such as UML models. In this approach, test cases are developed during analysis or design stage itself, preferably during the low level design stage. Design specifications are an intermediate artifact between requirement specification and final code. They preserve the essential information from the requirement, and are the basis of code implementation. Moreover, in component-based software development, often only the specifications are available and the source code is proprietary. Test case generation from design specifications has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. It is therefore desirable to generate test cases from the software design or analysis documents, in addition to test case design using the code.

Now, UML is widely used for object oriented modeling and design. Recently, several methods have been proposed to execute UML models [48, 41, 47, 18, 13, 11]. Executable UML [41, 47] allows model specifications to be efficiently translated into code. Executable UML formalizes requirements and use cases into a set of verifiable diagrams. The models are executable and testable and can be translated directly into code by executable UML model compilers. Besides reducing the effort in the coding stage, it also ensures platform independence and avoids obsolescence. This is so because the code often needs to change when ported to new platforms or fine tuning the code on efficiency or reliability considerations. It also allows meaningful verification of the models by executing them in a test and debug environment. Our test generation approach can also work on executable UML models.

UML-based automatic test case generation is a practically important and theoretically challenging topic. Literature survey indicates, testing based on UML specifications is receiving an increasing attention from researchers in the recent years. In using UML in the software testing process, here we focus primarily on the sequence diagrams where sequence diagrams model dynamic behavior. This is because most of the activities in software testing seek to discover defects that arise during the execution of a software system, and these defects are generally dynamic (behavioral) in nature [52]. Software testing is fundamentally concerned with behavior (what it does), and not structure (what it is) [27]. Customers understand software in terms of its behavior, not its structure. Further, UML is used in

the design of object-oriented software, which is primarily event-driven in nature. In such cases, the concept of a main program is minimized and there is no clearly defined integration structure. Thus there is no decomposition tree to impose the question of integration testing order of objects. Hence, it is no longer natural to focus on structural testing orders. Whereas, it is important to identify in what sequence objects interact to achieve a common behavior. In this context, UML sequence diagrams forms an useful means by which we can generate effective test cases for cluster level testing.

In this paper, we concentrate on UML sequence diagrams to automatically generate test cases. This paper is organized as follows: A brief discussion on sequence diagrams is given in the next section. In Section 3 we discuss few basic concepts. Section 4 describes our methodology to generate test cases from sequence diagrams and explains our methodology with an example. Section 5 discusses an implementation of our test methodology. Related research in the area of UML based testing is discussed in the Section 6 and conclusions are given in Section 7.

2 UML Sequence Diagrams

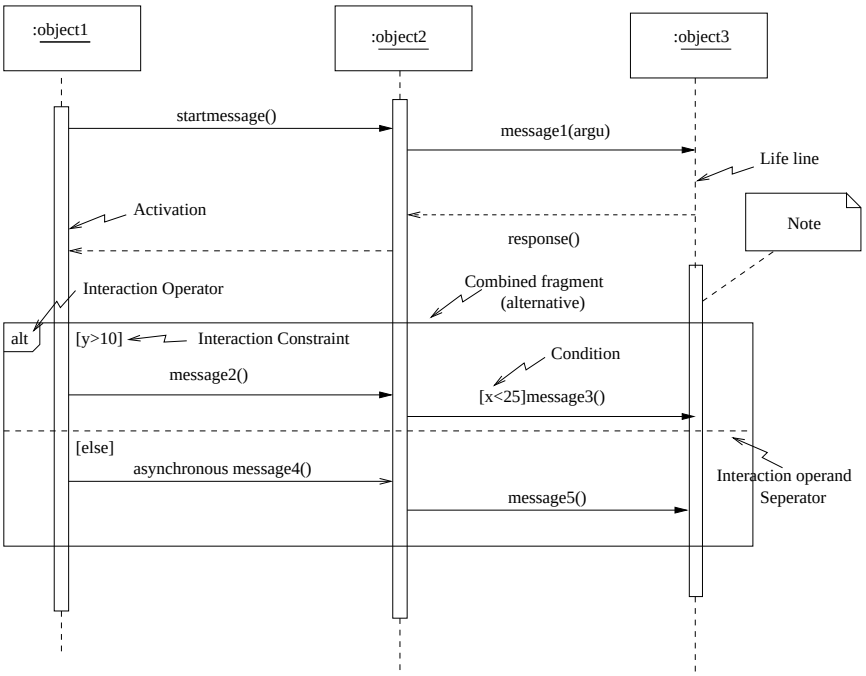

Figure 1: A Sequence Diagram Showing Various Notations

UML Sequence diagrams capture time dependent (temporal) sequences of interactions between objects. They show the chronological sequence of the messages, their names and responses and their possible arguments. A sequence diagram has two dimensions: the

vertical dimension represents time, and the horizontal dimension represents different instances. Normally time proceeds from top to bottom [44]. Message sequence descriptions are provided in sequence diagrams to bring forth meanings of the messages passed between objects. Sequence diagrams describe interactions among software components, and thus are considered to be a good source for cluster level testing. In UML, a message is a request for a service from one UML actor to another, these is typically implemented as method calls. We assume that each sequence diagram represents a complete trace of messages during the execution of a user-level operation.

An example of a UML sequence diagram is shown in Fig. 1. The vertical dashed line in the diagram is called a lifeline. A lifeline represents the existence of the corresponding object instance at a particular time. Arrows between the lifelines denote communication between object instances using messages. A message can be a request to the receiver object to perform an operation(of the receiver). A synchronous message is shown with a filled arrowhead at the end of a solid line. An asynchronous message is depicted with an open arrowhead at the end of a solid line. Return messages are usually implied. We can explicitly show return messages using an open stick arrowhead with a dashed line as shown in Fig. 1. An object symbol shown with a rectangle is drawn at the head of the lifeline. An activation (focus of control) shows the period during which an instance is performing a procedure. The procedure being performed may be labeled in text next to the activation symbol or in the margin.

UML 2.0 also allows an element called note, for adding additional information to the sequence diagram. Notes are shown with dog-eared rectangle symbols linked to object lifeline through a dashed line as shown in Fig.1. Notes are convenient to include pseudocode, constraints, pre-conditions, post-conditions, text annotations etc. in sequence diagram. However, in our approach we restrict the notes to contain only executable statements. Messages in the sequence diagram are chronologically ordered. So we have numbered them based on their timestamps. Further, we have numbered the notes in an arbitrary manner.

In UML 2.0, a set of interactions can be framed together and can be reused at other locations. Different interaction fragments can be combined to form a combined fragment. A combined interaction fragment defines an expression of interaction fragments. A combined interaction fragment is defined by an interaction operator and corresponding interaction operands. Through the use of combined fragments, the user will be able to describe a number of traces in a compact and concise manner. A combined fragment with an operator *alt* (for alternative) is shown in Fig. 1.

3 Basic Concepts

In this section, we discuss a few basic concepts that are useful to understand the rest of this paper.

Class, Cluster and System Level Testing: In object oriented systems, generally testing is done at different levels of abstraction: class level, cluster level and system level[9, 50, 30]. Class level testing tests the code for each operation supported by a class as well as all possible method interactions within the class. Class level testing also includes testing the methods in each of the states that a corresponding object may assume. At cluster level testing, the interactions among cooperating classes are tested. This is similar to integration testing. The system level testing is carried out on all the clusters making up the complete system.

Executable UML: Executable UML [41, 47] allows model specifications to be efficiently translated into code. Executable UML can formalize requirements and use cases into a rich set of verifiable diagrams. The models are executable and testable and can be translated directly into code by executable UML model compilers. The benefits of this approach go well beyond simply reducing or eliminating the coding stage; it ensures platform independence, avoids obsolescence (programming languages may change, the model doesn't) and allows full verification of the models by executing them in a test and debug environment.

Test Case: A test case is the triplet (I,D,O) where I is the state of the system at which the test data is input, D is the test data input to the system, and O is the expected output of the system [2, 40, 43]. The output produced by the execution of the software with a particular test case provides a specification of the actual software behavior.

4 Dynamic Slicing based Test Case Generation from Sequence Diagrams

In this section we describe our proposed methodology for automatic test case generation from UML sequence diagrams using dynamic slicing. We first define a few terms and the relevant test coverage criteria.

4.1 Definitions

The following definitions would be used in the description of our methodology.

Message Dependency Graph (MDG): We define MDG as a directed graph with (N,E), where N is a set of nodes and E is a set of edges. MDG shows the dependency of a given node on the others. Here a node represents either a message or a note in the sequence diagram and edges represent either control or data dependency among nodes. Here we have assumed that notes are attached to objects and the statements on the notes are executed when its corresponding lifeline is activated. MDG does not distinguish between control or data dependence edges. It does however distinguish between stable and unstable edges. Definitions of stable and unstable edges are given subsequently. The induced subgraph of MDG of the sequence diagram in Fig.2 on the Node Set(3,4,5,6,7,8,9,10,11,12,13) is shown

in Fig.3. of Subsection 4.7.

Slicing Criteria: Slices are constructed based on a slicing criterion. Weiser's slicing criterion [51] consisted of a set of variables of interest and a point of interest within the original program. Statements which cannot affect the values of variables at a point of interest in the program are removed to form the slice. In our case, the slicing criterion (m, V) specifies the location (identity) of a message m in its corresponding MDG and V is a set of variables that are used by the conditional predicate on the message at m.

Dynamic Slice: A dynamic slice of a sequence diagram is defined with respect to its corresponding MDG. Consider a predicate in MDG on a message m in a sequence diagram. A dynamic slice is the induced subgraph of MDG, induced by the set of nodes in MDG that affect a predicate at m for a given execution. We call this slice as a dynamic slice of sequence diagram. Those nodes of MDG that do not affect the predicate at m are removed to form the slice, for the slicing criterion (m,V).

UseVar(x): It is the set of all nodes in MDG that uses the value of variable x. For example, in the expression $(n = x * y)$ there is a use of the value of the variable x.

AllotVar(x): It is the set of all nodes in MDG that defines the variable x. In addition, consider a conditional guard specifies a condition in a message using variable x. If x is used to specify another condition in another message, such conditional guards are also treated as members of AllotVar(x). We use the term allotment to indicate that a variable x is either defined or if x is used to specify guards in the rest of paper. For example, consider nodes 4 and 8 in MDG as shown in Fig. 3. These nodes correspond to messages $[y < 50]msg4$ and $[y > 120]msg8$ respectively in Fig. 2. For a particular input value for the variable y, only one of these messages will take place. Hence, both nodes 4 and 8 are treated as members of AllotVar(y). A use of y will require only one of the AllotVar(y), not both.

Dependence Edge: If node i is a member of usevar(x) and a node j is a member of AllotVar(x), then there is a directed edge from node i to node j, this edge is also called a dependence edge.

Dependency Path: A dependency path F from some node v_i, to a node v_k is a sequence of nodes and edges in MDG from v_i to v_k.

Unstable Edge: Let M, M_i, M_j be three nodes in MDG. An outgoing dependence edge (M, M_i) in MDG is said to be unstable if there exists an outgoing dependence edge (M, M_j) with M_i not equal to M_j such that the statements M_i and M_j both are members of AllotVar(x). For example, in Fig.2, the messages $[y < 50]msg4$ and $[y > 120]msg8$ will form unstable edges with respect to the message $[a + y > 20]msg11$. These edges corresponds to (4,11) and (8,11) respectively in the MDG shown in Fig.3.

Stable Edge: An edge in a dependency graph is said to be stable, if it is not an unstable edge.

Slice Condition: Consider a slice S of a sequence diagram for the slicing criterion (m,V). The slice condition of the slice S is the conjunction of all the individual predicates present in the dynamic slice for a given execution.

Slice Domain: The slice domain of slice S is the set of all input data values for which the slice condition of S is satisfied.

Boundary: A slice domain is surrounded by a boundary. A boundary is a set of data points. A boundary might consist of several segments and each segment of the boundary is called a border [17]. Each border is determined by a single simple predicate in the slice condition. A border crossing occurs for some input where the conditional predicate changes its Boolean value from true to false or vice versa.

4.2 Test Coverage

A software test data adequacy criterion (or coverage criterion) is used to find out whether a set of test cases is sufficient, or "adequate," for testing a given software. Some of the relevant test criteria are introduced in this section.

4.2.1 Slice Coverage Criterion

Several test coverage criteria such as message path criteria, full predicate coverage etc., have been proposed in the literature[2]. Several other criteria such as slice coverage criterion can easily be formulated based on these criteria. We extend slice coverage criterion from path based criteria. Slice criteria is defined with respect to a dependency graph. We define slice coverage criterion for sequence diagram as follows: Consider a test set T and an MDG corresponding to a sequence diagram SD. In order to satisfy the slice coverage criterion, it is required that T must cause all the dependency paths in MDG for each slice to be taken at least once. Slice coverage ensures that all the dependency paths of an MDG (Message Dependency Graph) are covered.

4.2.2 Full Predicate Coverage

Full predicate coverage criterion requires that each clause should be tested independently[43]. In other words, each clause in each predicate on every message must independently affect the outcome of the predicate. Given a test set T and sequence diagram SD, T must cause each clause in every predicate on each message in SD to take on the values of TRUE and FALSE while all other clauses in the predicate have values such that the value of the predicate will always be the same as the clause being tested. This ensures that each clause in a condition is separately tested.

4.2.3 Boundary Testing Criterion

Testers have frequently observed that domain boundaries are particularly fault-prone and should therefore be carefully checked[25]. Boundary testing criterion is applicable whenever the test input domain is subdivided into subdomains by decisions (conditional predicates). Let us select an arbitrary border for each predicate p. We assume that the conditional predicates on the sequence diagram are relational expressions (inequalities). That is, all conditional predicates are of the following form: $E_1 op E_2$, where $E_1 and E_2$ are arithmetic expressions, and op is one of $\{<, \leq, >, \geq\}$. Jeng and Weyuker[25] have reported that an inequality border can be adequately tested by using only two points of test input domain, one named ON point and the other named OFF point. The ON point can be anywhere on the given border. It does not even have to lie exactly on the given border. All that is necessary is that it satisfies all the conditions associated with the border. The requirement for the OFF point is that it be as close to the ON point as possible, but it should lie outside the border.

The boundary testing criterion can now be defined as follows: The boundary testing criterion is satisfied for inequality borders if each selected inequality border b is tested by two points (ON-OFF) of test input domain such that, if for one of the points the outcome of a selected predicate q is true, then for the other point the outcome of q is false. Also the points should satisfy the slice condition associated with b and the points should be as close as possible to each other [17].

Definitions of boundary testing criteria for equality and non-equality borders are defined in [25, 17]. For conciseness, we do not consider them here. However they can easily be considered in our approach. We use boundary testing as an extension of slice coverage criterion. The number of test cases to be generated for achieving slice coverage criterion can be very large if we use a random approach. We reduce this by using boundary testing along with slice coverage. For example consider a the predicate $(n < 400)$ shown in Fig.2. In the example section we have shown two test data that can be used to test it and they are [21,20] and [21,19] where, the test data has the values of [x,y] for the predicate $(n < 400)$. The slice condition consists of $(x > 20), (y < 50), (n = x * y)$ and the test data is generated subject to slice condition. Instead of generating a set of test cases randomly and selecting the test cases from this set that satisfies this slice condition, we generate two test cases based on a simple predicate using boundary testing.

4.3 Overview of Our Approach

In our approach, the first step is to select a conditional predicate on the sequence diagram. The order in which we select predicates is the chronological order of messages appearing in a sequence diagram. For each message in the sequence diagram, there will be a corresponding node in the MDG. For each conditional predicate, we create the dynamic slice for the slicing criteria (m,V) and with respect to each slice we generate test data. The generated test data for each predicate corresponds to the true or false values of the conditional predicate

and these values are generated subject to the slice condition. This helps to achieve slice coverage. The different steps of our approach are elaborated in the following subsections.

4.4 Dynamic Slice of Sequence Diagrams

In our approach, a dynamic slice of a sequence diagram is constructed from its corresponding MDG (Message dependency graph). An MDG is created statically and it needs to be created only once. For each message in the sequence diagram, there will be a corresponding node in the MDG. From MDG, we create the dynamic slice corresponding to each conditional predicate, for the slicing criteria (m,V). For creating dynamic slices we use an edge marking method. Edge marking methods are reported in [26, 42] for generating dynamic slices in the context of procedural programs. Their edge marking methods uses program dependence graph. We generate a message dependence graph from UML sequence diagram and apply the edge marking technique on it. Edge marking algorithm is based on marking and unmarking the unstable edges appropriately as and when dependencies arise and cease at run time. After an execution of the node x at run-time, an unstable edge(x,y) is marked if the node x uses the value of the variable v at node y and node y is a member of AllotVar(v). A marked unstable edge(x,y) is unmarked after an execution of a node z if the nodes y and z are in AllotVar(v), and the value of v computed at node y does not affect the present value of v at node z. In our approach we generate test data that satisfies all constraints corresponding to a slice.

Before execution of a message sequence M, the type of each of its edges in MDG is appropriately recorded as either stable or unstable. The dependence associated with a stable edge exists at every point of execution. The dependence associated with an unstable edge keeps on changing with the execution of the node. We mark an unstable edge when its associated dependence exists, and unmark when its associated dependence ceases to exist. Each stable edge is marked and each unstable edge is unmarked at the time of construction of the MDG. We mark and unmark edges during the execution of the message sequences, as and when a dependencies arise or cease, and a stable edge is never unmarked. Let dslice(n) denote the dynamic slice with respect to the most recent execution of the node n. Let $(n, x_1), (n, x_2), \ldots, (n, x_n)$ be all the marked outgoing dependence edges of n in the updated MDG after an execution of the node n. It is clear that the dynamic slice with respect to the present execution of the node is $dslice(n) = x_1, x_2, \ldots, x_n \cup dslice(x_1) \cup \cdots \cup dslice(x_n)$.

We now present the edge marking dynamic slicing algorithm for sequence diagrams in pseudocode form. Subsequently this method is explained using an example.

Edge Marking Dynamic Slicing Algorithm for Sequence Diagrams

- Do before execution of the message sequence:-
  - Unmark all the unstable edges.
  - Set dslice(n) = NULL for every node n of the MDG.

- For each node n of the message sequence Do
  - For every variable used at n, mark the unstable edge corresponding to its most recent allotment. (Suppose there is a predicate $x > 50$ which is true for the given execution step and inputs then the edge to that predicate is marked. If the predicate is false then it remains unmarked.)
  - Update dslice(n).
  - If n is a member of AllotVar(x) and n is not a UseVar(x) node, then do the following :-
    - Unmark every marked unstable edge (n_1, n_2) with $n_1 \in UseVar(x)$ and n_2 is a node that does not affect the present allotment of the variable var. Hence, the marked unstable edge (n_1, n_2) representing the dependence of node n_1 on node n_2 in the previous execution of node n_1 will not continue to represent the same dependence in the next execution of node n_1.

For example, let x = 30, y = 45, p = 55, q = 40, a = 10 be a data set for the diagram given in Fig. 2. For the slicing criteria (11,y), initially let edges (11,4) and (11,8) are unmarked unstable edges as seen in Fig. 3. During the execution of node 11, for the given data set, we mark the unstable edge (11,4) whereas the unstable edge (11,8) remains unmarked as the value of y at present is 45. Hence the dynamic slice of node 11 for the slicing criteria (11,y) is $4 \cup dslice6$ and do not include 8. Let at some other execution, the data set is x = 30, y = 55, p = 45, q = 40, a = 10. In this case the dynamic slice of node 11 for the slicing criteria (11,y) is $8 \cup dslice6$ and do not include 4.

4.5 Generation of Predicate Function

Consider an initial set of data I_0 that is randomly generated for the variables that affect a predicate p in a slice S. As already mentioned in our approach, we compute two points named ON and OFF for a given border satisfying the boundary testing criterion. We transform the relational expressions of the predicates to a function F (Predicate Function). If the predicate p is of the form (E1 op E2), where E1 and E2 are arithmetic expressions, and op is a relational operator, then F = (E1 - E2) or (E2 - E1) depending on whichever is positive for the data I_0. Next we successively modify the input data I_0 such that the function F decreases and finally turns negative. When F turns negative, it corresponds to the alternation of the outcome of the predicate. Hence as a result of the above predicate transformation, the change in the outcome of predicate p now corresponds to the problem of minimization of the function F. This minimization can be achieved through repeated modification of input data value.

4.6 Test Data Generation

The basic search procedure we use for finding the minimum of the predicate function F is the alternating variable method[31, 17] which consists of minimizing F with respect to each input variable in turn. Each input data variable x_i is increased/decreased in steps of

Ux_i, while keeping all the other data variables constant. Here Ux_i refers to a unit step of the variable x_i. The unit step depends on the data type being considered. For example, the unit step is 1 for integer values. The method works with many other types of data such as float, double, array, pointer etc. However the method may not work when the variable assumes only a discrete set of values. Each predicate in the slice can be considered to be a constraint. If any of the constraint is not satisfied in the slice, for some input data value, we say that a constraint violation has taken place. We compute the value of F when each input data is modified by Ux_i. If the function F has decreased on the modified data, and constraint violation has not occurred, then the given data variable and the appropriate direction is selected for minimizing F further. Here appropriate direction refers to whether we increase or decrease the data variable x_i. We start searching for a minimum with an input variable while keeping all the other input variables constant until the solution is found (the predicate function becomes negative) or the positive minimum of the predicate function is located. In the latter case, the search continues from this minimum with the next input variable.

4.7 An Example

Consider an example sequence diagram as shown in Fig.2. We have selected this example as it demonstrates the concepts in our approach. We illustrate our methodology by explaining the test data generation for the predicate ($n < 400$) shown in Fig.2. Its corresponding MDG is shown in Fig.3. Let the slicing criterion be (6,n). For this slicing criterion, the slice contains of the set of nodes that corresponds to predicates $(x > 20), (y < 50), (n = x * y)$. The function F will be the expression $(n - 400)$. Let I_0 be the initial data: [25,40] where (x = 25, y = 40). The condition $(n < 400)$ is false for I_0 as $(1000 < 400)$. The function F will be the expression $(n - 400)$ and F(I_0) = 600. We should minimize F, in order to alter the boolean outcome of predicate $(n < 400)$, which is false initially.

First we decrease the value of data x in steps. In the first step, we take x =24 and the value of F is calculated as 560 for [x,y] = [24,40]. Observe that the function F reduces by reducing x. Therefore in the next step, the size of the step is doubled and hence the value of variable x is decreased by 2. As we minimize F further in several iterations, we finally arrive at two data points With [x,y] = [21,20], F is positive and the condition $(n < 400)$ is still false. So we take two data sets I_in as [x,y] = [21,20] that makes F positive (or zero) and another data set I_out as [x,y] = [21,19] that makes F negative.

The test cases we generate for the predicate $(n < 400)$ are (object1, [21,19], object2) and (object1, [21,20], object1) correspond to different truth values of the predicate $(n < 400)$. Here test cases has the form (sender object, [test data], receiver object). Test data has the values of [x,y] for the predicate $(n < 400)$. These test cases are generated satisfying the slice condition of the slice. With our proposed method we generate test cases for each such conditional predicates on the sequence diagram.

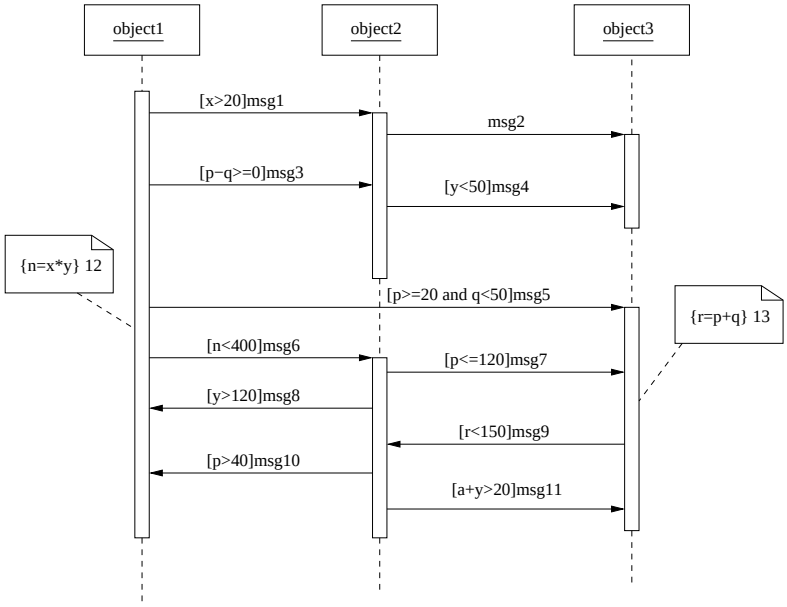

Figure 2: An Example Sequence Diagram

5 An Implementation

To the best of our knowledge, no full-fledged ready made tool exists that are publicly available to execute UML models. Hence, for generating dynamic slices in our experimentation, we have simulated the executions. We made a prototype tool that implements our method. Fig. 4 shows the important classes that we used to generate test cases from sequence diagram in our implementation. *SliceGenerator* class creates the message dependency graph. It makes the sets defSet and useSet for each variable in the sequence diagram. It forms slices based on the slicing criteria for each of the messages in the sequence diagram. *SliceRecord* class keeps a record of slices.

DocumentParser class parses the XML file corresponding to a UML sequence diagram. We used the Document Object Model (DOM) API that comes with the standard edition of the Java platform, for parsing XML files. The package org.w3c.dom.*, provides the interfaces for the DOM. The DOM parser begins by creating a hierarchical object model of the input XML document. This object model is then made available to the application for it to access the information it contains in a random access fashion. This allows an application to process only the data of interest and ignore the rest of the document.

XmlBoundary is the class of the program from which the execution starts. It accepts an XML file of sequence diagram from a user. Then it extracts the parent tag of the XML file and passes the tag (called head) to the TestCaseController class. *TestCaseController* class coordinates the different activities of the program. *TestCaseBoundary* class is responsible for displaying the list of test cases for a collaboration diagram. The source and destination

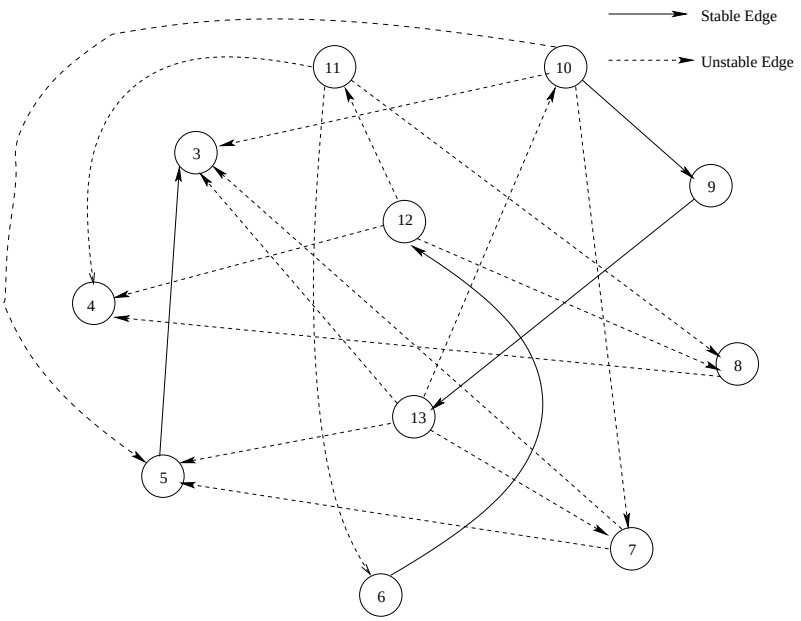

Figure 3: The Induced Subgraph of Dependency Graph of the Sequence Diagram in Fig.2 on the Node Set(3,4,5,6,7,8,9,10,11,12,13)

objects as well as the slice condition is printed along with test data.

In our prototype implementation, we have considered only integer and Boolean variables as part of the conditional expression in sequence diagrams. Other data types however can easily be considered. Further, for the prototype implementation we have assumed that the necessary constraints are available in notes instead of class/object diagrams. Extracting data types of attributes, or constraints from class/object diagrams for our implementation can be easily done. The GUI was developed using the swing component of Java. A GUI screen along with a sample sequence diagram is shown in Fig. 5. The GUI gives the flexibility to view the sequence diagram, its XML representation and the generated test cases. Fig. 6 shows the UTG display of the XML file of example given in Fig. 5. The corresponding test cases generated are shown in Fig. 7. Our tool allows storing the test cases as text files for later processing.

We have implemented our method for generating test cases automatically from UML sequence diagrams in a prototype tool named UTG. Here, UTG stands for UML behavioral Test case Generator. UTG has been implemented using Java and can easily integrate with any UML CASE tools like MagicDraw UML [24] that supports XML (Extensible Markup Language) format. Since UTG takes UML models in XML format as input, UTG is independent of any specific CASE tool. We have used the tool with several UML designs and the tool was found effective in generating test cases. The generated test cases were found to achieve the desired coverage.

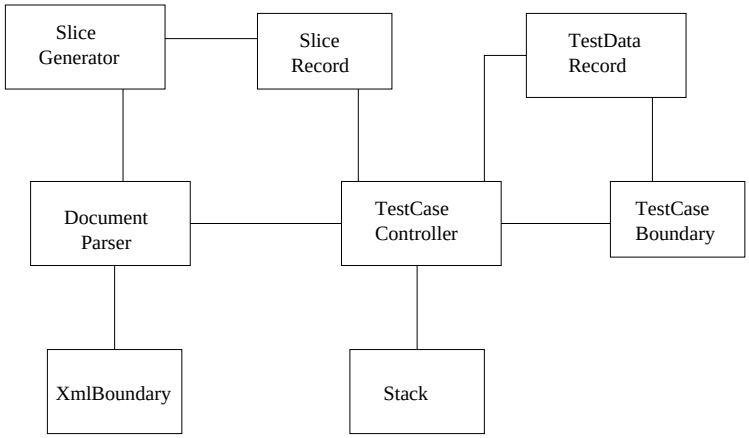

Figure 4: Class Diagram of UTG for Generating Test Cases From Sequence Diagrams

6 Related Work

Bertolino and Basanieri [4] proposed a method to generate test cases following the sequence of messages between components in a sequence diagram. They develop sequence diagrams for each use case and use category partition method to generate test data. They characterize a test case as a combination of all suitable choices of the involved settings and interactions in a sequence of messages. In another interesting work, Basanieri, et al. [3] describe the CowSuite approach which provides a method to derive the test suites and a strategy for test prioritization and selection. CowSuite is mainly based on the analysis of the use case diagrams and sequence diagrams. From these two diagrams they construct a graph structure which is a mapping of the project architecture and this graph is explored using depth-first search algorithm. They use category partition method [46] for generating test cases. They construct test procedures using the information retrieved from the UML diagrams.

Briand and Labiche [7] describe the TOTEM (Testing Object-orienTed systEms with the Unified Modeling Language) system test methodology. Functional system test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and from OCL used in all these artifacts. They represent sequential dependencies among use cases by means of an activity diagram constructed for each actor in the system. The derivation of use case sequences from the activity diagram is done with a depth first search through a directed graph capturing the activity diagram. They generate legal sequences of use cases according to the sequential dependencies specified in the activity diagram. Abdurazik and Offutt [1] proposed novel and useful test criteria based on collaboration diagrams for static checking and dynamic testing based on collaboration diagrams. They recommended a criterion for dynamic testing that involved message sequence paths. They adapt traditional data flow

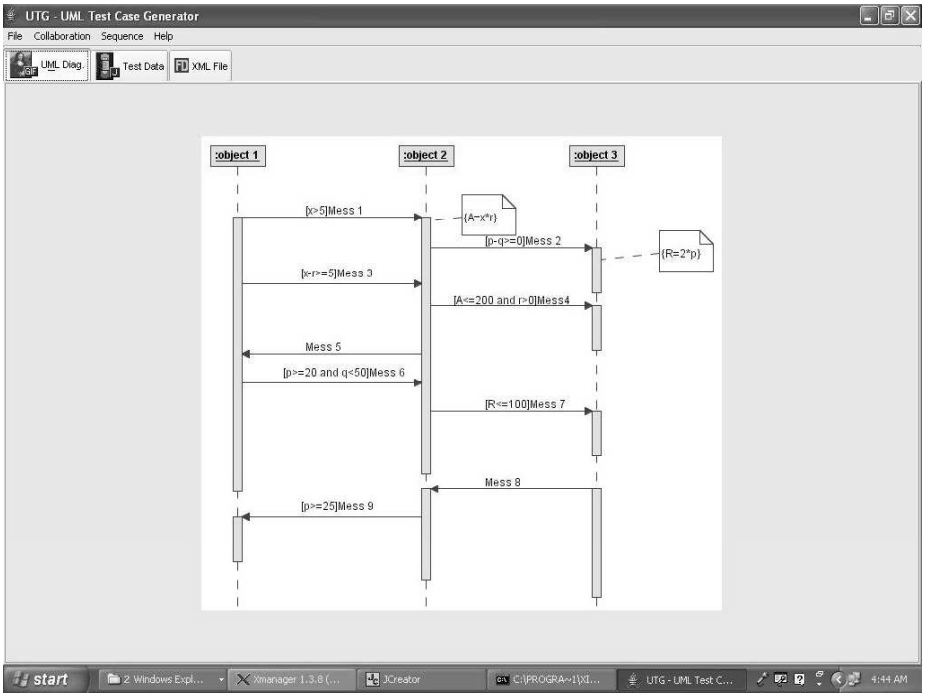

Figure 5: The GUI Screen of UTG With an Example Sequence Diagram

coverage criteria (eg. all definition - uses) in the context of UML collaboration diagrams.

Linzhang, et al. [36] proposed a gray-box testing method using UML activity diagrams. They propose an algorithm to generate test scenarios from activity diagrams. The information regarding input/output sequence, parameters, the constraint conditions and expected object method sequence is extracted from each test scenario. They recommend applying category-partition method to generate possible values of all the input/output parameters to find the inconsistency between the implementation and the design.

Among all UML diagrams, test case generation from state chart diagram has possibly received maximum attention from researchers [8, 22, 29, 30, 43, 49]. Offutt and Abdurazik [43] developed an interesting technique for generating test cases from UML state diagrams which is intended to help perform class-level testing. Their method takes a state transition table as input, and generates test cases for the full predicate coverage criterion. It processes each outgoing transition of each source state, generates a test case that makes the transition taken, and then generates test cases that make the transition untaken. A test case is designed corresponding to each variable in a transition predicate. To avoid redundant test case value assignments, those variables that have already been assigned values are not considered in the subsequent test case value assignment process. After all test case values are generated, an additional algorithm is run on the test cases to identify

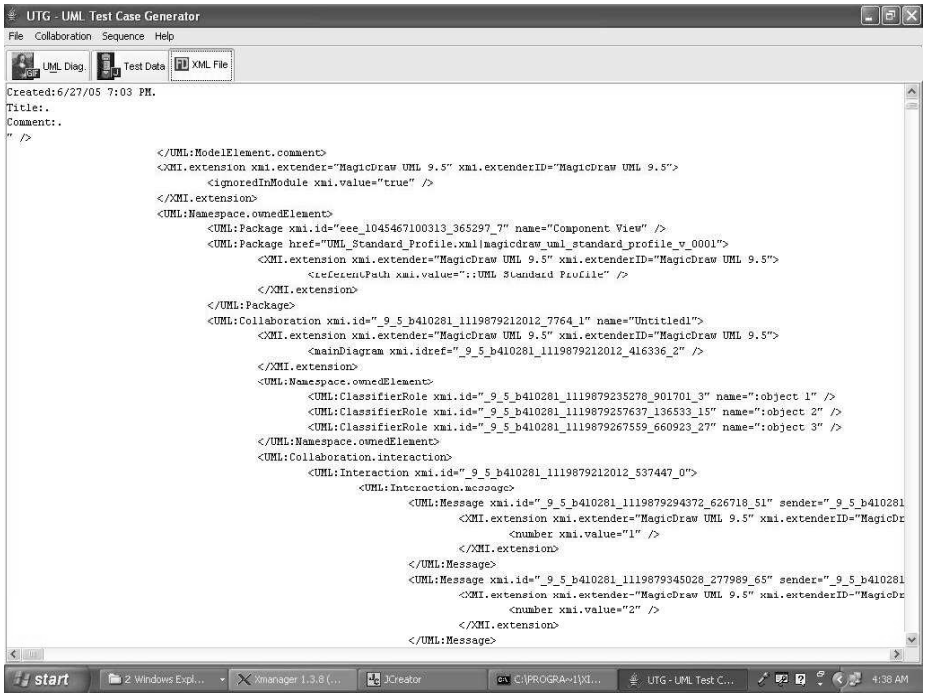

Figure 6: A Screen Shot of UTG with a Portion of the XML File Corresponding to example in Fig. 5

and remove redundant test cases. Kansomkeat, et al. [29] have proposed an alternate method for generating test sequences using UML state chart diagrams. They transform the state chart diagram into an intermediate diagram called Testing Flow Graph (TFG) which is used to generate test sequences. TFG is a flattened hierarchy structure of states. The testing criterion they proposed is the coverage of states and transitions of TFG.

Kim, Y.G et al. [30] proposed a method for generating test cases for class testing using UML state chart diagrams. They transform state charts to extended finite state machines (EFSMs) to derive test cases. The hierarchical and concurrent structure of states is flattened and broadcast communications are eliminated in the resulting EFSMs. Next data flows are identified by transforming EFSMs into flow graphs to which conventional data flow analysis techniques are applied. Hartmann et al. [22] augment the UML description with specific notations to create a design-based testing environment. The developers first define the dynamic behavior of each system component using a state diagram. The interactions between components are then specified by annotating the state diagrams, and the resulting global FSM that corresponds to the integrated system behavior is used to generate the tests.

Scheetz et al. [49] developed an approach for generating system (black box) test cases using an AI (Artificial Intelligence) planner. They used UML class diagrams and state di-

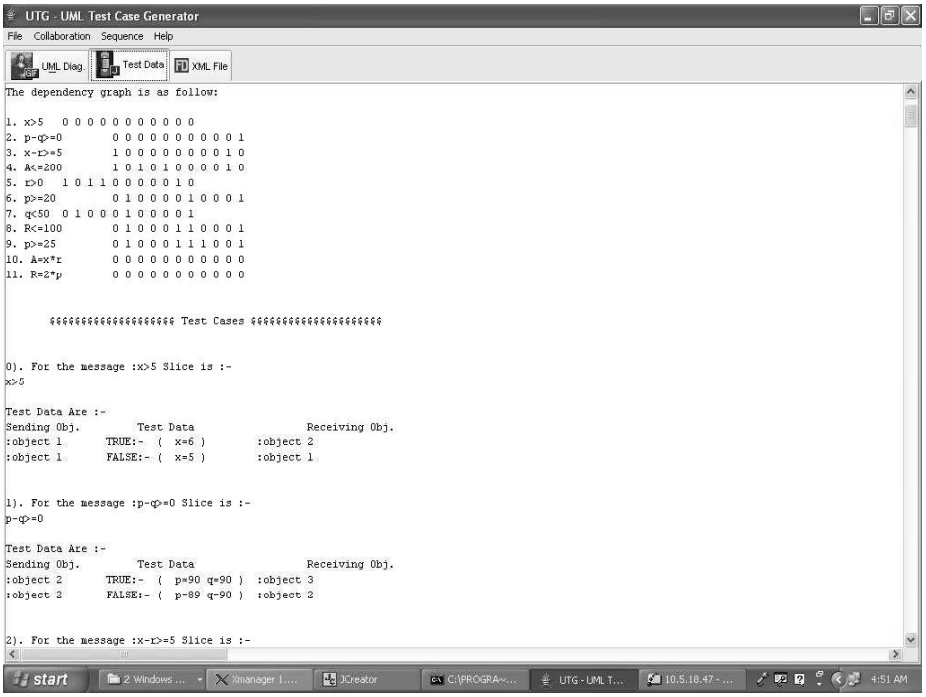

Figure 7: A Screen Shot of UTG with Dependency Graph and Generated Test Cases Corresponding to the Example of Fig. 5

agrams to represent the conceptual architecture of a system under test. They developed a representation method at the application domain level that allows statement of test objectives at that level, and their mapping into a planner representation. Their method maps the initial and goal conditions into a problem description for the planner. The planner generates a plan based on this input. In the next step, they carry out a conversion of the plan to produce executable test cases. The purpose of a test case in a goal directed view is to try to change the state of the overall system to the goal state. The planner decides which operators will best achieve the desired goal states. Cavarra, et al. [8] use UML class diagrams, state diagrams, and object diagrams to characterize the behavior of a system. These UML diagrams are translated into formal behavioral descriptions, written in a language of communicating state machines and used as a basis for test generation. From this they form a test graph, consisting of all traces leading to an accept state, together with branches that might lead to invalid state.

Andrews et al. [2] describe several useful test adequacy criteria for testing executable forms of UML. The criteria proposed for class diagrams include association-end multiplicity criterion, generalization criterion and class attribute criterion. The interaction diagram criteria like condition coverage, full predicate coverage, each message on link, all message paths and collection coverage criteria are used to determine the sequences of messages that should be tested. They also describe a test process. Ghosh et al. [16] present a testing

method in which executable forms of Unified Modeling Language (UML) models are tested. In systematic design testing, executable models of behaviors are tested using inputs that exercise scenarios. This can help reveal flaws in designs before they are implemented in code. Their method incorporates the use of test adequacy criteria based on UML class diagrams and interaction diagrams. Class diagram criteria are used to determine the object configurations on which tests are run, while interaction diagram criteria are used to determine the sequences of messages that should be tested. These criteria can be used to define test objectives for UML designs. Engels et al. [12] discuss how consistency among different UML models can be tested. They propose dynamic meta modeling rules as a notation for the consistency conditions and provide the concept for an automated testing environment using these rules.

In contrast with the above discussed approaches we generate actual test cases from sequence diagrams. Our approach can work on executable forms of UML design specifications and is meant for cluster level testing where object interactions are tested. Corresponding to each conditional predicate on the sequence diagram, we construct dynamic slice from its MDG and with respect to the slice we generate test data. Our test data generation scheme is automatic.

Kamkar et al. [28] explains how interprocedural dynamic slicing can be used to increase the reliability and precision of interprocedural data flow testing. Harman and Danicic [19] presents an interesting work that illustrates how slicing will remove statements which do not affect a program variable at a location thereby simplifying the process of testing and analysis. They also provide a program transformation algorithm to make a program robust. Slicing has been used as a reduction technique on specifications like state models [34]. Anyhow this work [34] do not provide a scheme for test generation

Korel [31] generated test data based on actual execution of a program under test. He used function minimization methods and dynamic data flow analysis. If during a program run an undesirable execution flow is observed (e.g., the "actual" path does not correspond to the selected control path), then function minimization search algorithms are used to automatically locate the values of input variables for which the selected path is traversed. This helps in achieving path coverage. In addition, dynamic data flow analysis is used to determine those input variables that are responsible for the undesirable program behavior, leading to significant speedup of the search process. Hajnal et al. [17] extended the work done by Korel [31]. They reported the use of boundary testing that requires the testing of one border only along a selected path. The test input domain may be surrounded by a boundary and each segment of the boundary is called a border. The task to generate two test data points considering only one border for each path, is much easier. Their testing strategy can also handle compound predicates. Jeng and Weyuker [25] have reported that an inequality border can be tested by only two points of test input domain, one named ON point and another named OFF point. For borders in a discrete space containing no points lying exactly on the border, their strategy allows the ON point to be chosen from beneath the border as long as the distance between the ON and OFF points is minimized.

These works [28, 19, 31, 17, 25] discussed above have focused on unit testing of procedural programs.

7 Conclusion

We have presented a novel method to generate test cases by dynamic slicing UML sequence diagrams. Our approach is meant for cluster level testing where object interactions are tested. Our approach automatically generates test data, which can be used by a tool to carry out automatic testing of a program. Generation of MDG is the only static part in our approach. We identify the conditional predicates associated with messages in a sequence diagram and create dynamic slice with respect to each conditional predicate. We generate test data with respect to each constructed slice and the test data is generated satisfying slice condition. We have formulated a test adequacy criterion named slice coverage criterion. We have implemented our methodology to develop a prototype tool which was found effective in generating test cases. The test cases generated can also be used for conformance testing of the actual software where the implementation is tested to check whether it conforms to the design. The slicing approach was found to be especially advantageous when the number of messages in the sequence diagram is large. We need to consider only the slices for finding test cases instead of having to look at the whole sequence diagram. If the sequence diagram is large it becomes very complex and difficult to find test cases manually. If we know where to look for errors it becomes a great simplification and saves a lot of time and resources. The slices help to achieve this simplification. The generated test cases were found to achieve slice coverage.

Acknowledgements

The authors would like to thank Pratyush Kanth and Sandeep Sahoo for implementing our approach presented in this paper.

References

[1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the 3rd International Conference on the UML, Lecture Notes in Computer Science*, volume 1939, pages 383 – 395, York, U.K., October 2000. Springer-Verlag GmbH.

[2] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing Verification and Reliability*, 13:97 – 127, 2003.

[3] F. Basanieri, A. Bertolino, and E. Marchetti. The cow suit approach to planning and deriving test suites in UML projects. In *Proceedings of the Fifth International Conference on the UML, LNCS*, volume 2460, page 383–397, Dresden, Germany, October 2002. Springer-Verlag GmbH.

[4] A. Bertolino and F. Basanieri. A practical approach to UML-based derivation of integration tests. In *Proceedings of the 4th International Software Quality Week Europe and International Internet Quality Week Europe*, Brussels, Belgium, 2000. QWE.

[5] R. V. Binder. Testing object-oriented software: a survey. *Software Testing Verification and Reliability*, 6(3/4):125 – 252, 1996.

[6] D. Binkley and K. Gallagher. *Program Slicing*, volume 43 of *Advances in Computers*. Academic Press, 1996.

[7] L. Briand and Y. Labiche. A UML-based approach to system testing. In *Proceedings of the 4th International Conference on the UML, LNCS*, volume 2185, pages 194 – 208, Toronto, Canada, January 2001. Springer-Verlag GmbH.

[8] A. Cavarra, C. Crichton, and J. Davies. A method for the automatic generation of test suites from object models. *Information and Software Technology*, 46(5):309 – 314, 2004.

[9] H. Y. Chen, T. H. Tse, and T. Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(4):56–109, January 2001.

[10] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *IEEE International Conference on Software Maintenance (ICSM'97)*, page 188 – 195. IEEE Computer Society Press, Los Alamitos, USA, 1997.

[11] T. T. Dinh-Trong. Rules for generating code from UML collaboration diagrams and activity diagrams. Master's thesis, Colorado State University, Fort Collins, Colorado, 2003.

[12] G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proceedings of the Sixth International Conference on Integrated Design and Process Technology(IPDT)*, USA, 2002. Society for Design and Process Science.

[13] G. Engels, R. Hucking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformations to java. In *Proceedings of the 2nd International Conference on the UML, LNCS*, volume 1723, pages 473 – 488, Berlin / Heidelberg, October 1999. Springer.

[14] F.Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121 – 189, June 1995.

[15] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751 – 761, August 1991.

[16] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 332 – 343. IEEE Computer Society, November 2003.

[17] A. Hajnal and I. Forgacs. An applicable test data generation algorithm for domain errors. In *ACM SIGSOFT Software Engineering Notes, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, volume 23, 1998.

[18] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31 – 42, 1997.

[19] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing Verification and Reliability*, 5(3):143 – 162, September 1995.

[20] M. Harman, C. Fox, R. M. Hierons, D. Binkley, and S. Danicic. Program simplification as a means of approximating undecidable propositions. In *7th IEEE Workshop on Program Comprehension*, page 208 – 217. IEEE Computer Society Press, Los Alamitos, USA, 1999.

[21] M. Harman and K.B.Gallagher. Program slicing. *Information and Software Technology*, 40:577 – 581, December 1998.

[22] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *ACM SIGSOFT Software Engineering Notes, Proceedings of International Symposium on Software testing and analysis*, volume 25, August 2000.

[23] S. Horwitz, J. Prins, and T. Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345 – 387, July 1989.

[24] N. M. Inc. *MagicDraw* UML*, Version 9.5.* Golden, CO, www.magicdraw.com.

[25] B. Jeng and E. J. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology(TOSEM)*, 3(3), 1994.

[26] J.Horgan and H. Agrawal. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, volume 25, pages 246 – 256, White Plains, New York, 1990. SIGPLAN Notices, Analysis and Verification.

[27] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9), September 1994.

[28] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance*, page 386 – 395. IEEE Computer Society, Washington, DC, USA, 1993.

[29] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In *Proceedings of SAICSIT 2003*, pages 296 – 300. ACM, 2003.

[30] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *Proceedings: Software*, 146(4):187 – 192, 1999.

[31] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870 – 879, 1990.

[32] B. Korel and J.Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, October 1988.

[33] B. Korel and J. Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40:647 – 659, 1998.

[34] B. Korel, I. Singh, L. H. Tahat, and B. Vaysburg. Slicing of state-based models. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 34 – 43. IEEE, 2003.

[35] A. Lakhotia and J. C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology Special Issue on Program Slicing*, 40:677 – 689, 1998.

[36] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating test cases from UML activity diagrams based on gray-box method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 284 – 291. IEEE, 2004.

[37] A. D. Lucia. Program slicing: methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, page 142 – 149. IEEE, November 2001.

[38] A. D. Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In *4th IEEE Workshop on Program Comprehension*, page 9 – 18. IEEE Computer Society Press, Los Alamitos, USA, 1996.

[39] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *2nd International Conference on Computers and Applications*, page 877 – 882. IEEE Computer Society Press, Los Alamitos, USA, 1987.

[40] R. Mall. *Fundamentals of Software Engineering.* Prentice Hall, 2nd edition, 2003.

[41] S. J. Mellor and M. J. Balcer. *Executable* UML*: A Foundation for Model Driven Architecture.* Addison-Wesley: Reading, MA, 2002.

[42] G. Mund, R. Mall, and S. Sarkar. An efficient program slicing technique. *Information and Software Technology*, 44:123 – 132, 2002.

[43] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on UML, Lecture Notes in Computer Science*, volume 1723, pages 416 – 429, Fort Collins, TX, 1999. Springer-Verlag GmbH.

[44] OMG. *Unified Modeling Language Specification, Version 2.0.* Object Management Group, www.omg.org, August 2005.

[45] L. Osterweil. Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, 28(4), December 1996.

[46] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6), June 1998.

[47] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable* UML. Cambridge University Press, 2004.

[48] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a uml virtual machine. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 36, pages 327 – 341. ACM SIGPLAN Notices, ACM Press, USA, October 2001.

[49] M. Scheetz, von A. Mayrhauser, and R. France. Generating test cases from an object oriented model with an AI planning system. In *Proceedings of the 10th International Symposium on Software Reliability Engineering, ISSRE 99*, pages 250 – 259. IEEE Computer Society Press, 1999.

[50] M. D. Smith and D. J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45 – 53, June 1992.

[51] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352 – 357, 1984.

[52] C. E. Williams. Software testing and the UML. In *Proceedings of the International Symposium on Software Reliability Engineering, (ISSRE'99)*, Boca Raton, FL, November 1999.