Test Case Generation Based on Use case and Sequence Diagram

Article · January 2010

Source: DOAJ

CITATIONS
62

READS
5,442

3 authors:

Santosh Kumar Swain
KIIT University
19 PUBLICATIONS **212** CITATIONS
SEE PROFILE

Durga Prasad Mohapatra
National Institute of Technology Rourkela
211 PUBLICATIONS **1,084** CITATIONS
SEE PROFILE

Rajib Mall
Indian Institute of Technology Kharagpur
168 PUBLICATIONS **2,269** CITATIONS
SEE PROFILE

Test Case Generation Based on Use case and Sequence Diagram

Santosh Kumar Swain[(1)], Durga Prasad Mohapatra[(2)], and Rajib Mall[(3)]

(1) School of Computer Engineering, KIIT University, Bhubaneswar, Orissa (India)
E-mail: swainsantosh@yahoo.co.in

(2) Department of Computer Science & Engineering, NIT, Rourkela, Orissa (India)
E-mail: durga@nitrkl.ac.in

(3) Department of Computer Science & Engineering, IIT, Kharagpur,W. B.(India)
E-mail: rajib@cse.iitkgp.ernet.in

ABSTRACT

We present a comprehensive test case generation technique from UML models. We use the features in UML 2.0 sequence diagram including conditions, iterations, asynchronous messages and concurrent components. In our approach, test cases are derived from analysis artifacts such as use cases, their corresponding sequence diagrams and constraints specified across all these artifacts. We construct Use case Dependency Graph (UDG) from use case diagram and Concurrent Control Flow Graph (CCFG) from corresponding sequence diagrams for test sequence generation. We focus testing on sequences of messages among objects of use case scenarios. Our testing strategy derives test cases using full predicate coverage criteria. Our proposed test case generation technique can be used for integration and system testing accommodating the object message and condition information associated with the use case scenarios. The test cases thus generated are suitable for detecting synchronization and dependency of use cases and messages, object interaction and operational faults. Finally, we have made an analysis and comparison of our approach with existing approaches, which are based on other coverage criterion through an example.

Keywords: Concurrent control flow graph, full predicate coverage, sequence diagram, UML 2.0 and use case.

1- INTRODUCTION

Software testing plays a crucial role in assuring software quality. It can be used for the purposes of quality assurance, reliability estimation and verification and validation. As the complexity and size of software grow, the time and effort required to do effective testing increase. Studies indicate that more than 50% of the cost of software development is devoted to testing [3]. There is increasing need for effective testing of software for complex, safety-critical applications such as avionics, medical, and other control systems. One of the most important issues in the software testing research is the

generation of the test cases. Both test case design and test case executions are time consuming and labor intensive. Hence, automation of test case generation is an important issue. Manual test design is time consuming and error-prone. It is necessary to develop automatic tests design techniques. Therefore, with continually increasing software sizes, the issue of automatic design of test cases is assuming crucial importance. There are essentially two main approaches to automatic design of test cases. One approach attempts to design test cases from requirement and design specification and the other from code. Test case design from program code is cumbersome and difficult to automate. Generating test models like control flow graph from source code is cumbersome [27]. Design notations can be used as a basis for output checking, significantly reducing one of the major costs of testing. The process of generating tests from design will often help the test engineer to discover problems with design itself. If this step is done early, the problems can be eliminated early, saving time and resources. Generating tests during design also allows testing activities to be shifted to an earlier part of the development process, allowing for more effective planning of test cases. When the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications and design, allowing the specifications and design to be improved before the program is written. Another advantage is that the test data is independent of any particular implementation. Generating test cases at early stages is a good supplement to testing. These test cases can be tested at later stages coding. Automation of the process would result in tremendous improvement in testing process. Most of the requirements and high-level designs of software are documented in the form of models. These models can be used for test case generation of the system by automated means.

Model-driven software development is a relatively recent software development paradigm [26]. Its advantages are the increased productivity with support for visualizing domains like business and problem domains, solution domain and generation of implementation artifacts. In the model-driven software development, practitioners also use the design model for testing software- especially object-oriented programs. Three main reasons for using design model in object-oriented software testing are: (1) traditional software testing techniques consider only static view of code, which is not sufficient for testing dynamic behavior of object-oriented softwares [3], (2) use of code to test an object-oriented software is a complex and tedious task,. in contrast, models help software testers to understand systems in better way and find test information only after simple processing of models compared to code, (3) model-based test case generation can be planned at an early stage of the software development life cycle, allowing software developer to carry out coding and testing in parallel. For these three major reasons, model-based test case generation methodology becomes an obvious choice in software industries and is the focus of this paper.

Unified Modeling Language (UML) is an Object Management Group (OMG) standard for object-oriented modeling that has gain widespread use in the

software industry [15, 16]. Using UML, developers model large, complex systems and produce variety of different diagrams presenting different views of the system model. Although UML provides a powerful mechanism for describing software that is safety-critical or that must be highly reliable, very less work has been done in the area of utilizing these models for testing purpose [25]. With the increasing use of the UML to model object-oriented software, researchers have begun investigating how the UML can be used in the testing phase of the software development process. Consequently, several UML-based approaches to software testing have been proposed [1, 2, 5, 10, 14]. In these approaches, test requirements and coverage criteria are derived from UML models. In this paper, we have proposed a novel technique, which generates the test cases from the UML use case diagrams and sequence diagrams.

UML diagrams can be used individually or combined together to perform unit testing as well as integration testing. For example, the state chart diagrams can be used for unit testing [2] and interaction diagrams like collaboration and sequence diagrams can be used for integration testing of objects. Typically, the complexity of an object-oriented system lies in its object interactions, not within class methods which tend to be small and simple. As a result, complex behaviors are observed when related objects pass messages with each other within a scenario. We define model-based CFG to derive control flow information from the UML diagrams. UML provides ways to model the behavior of an Object-Oriented system using interaction (sequence and collaboration) diagrams. Because of higher level of abstraction, one of the advantages of Model-based CFG over Code-based CFG is easier extraction of concurrent control flow and dynamic control flow information (such as polymorphism), which will be benefited for testing. Model-based CFG can be used for generation of model-based test requirements, needed by various testing techniques, earlier in the software development life cycle. UML provides ways to model behavior of an Object-Oriented system using interaction (sequence and collaboration) diagrams. Furthermore, UML 2.0 [29] has proposed a set of rich features such as interaction and combined fragments which enable designers to model code-like program structures such as conditions, loops and call to other SDs (sequence diagrams). The degree of formality in these modeling features has enhanced considerably in UML 2.0 [29] comparing to its previous 1.x versions (i.e.1.3, 1.5).

To each use case corresponds an interaction model, i.e., either a collaboration or a sequence diagram. These diagrams show how the use case is realized through the interactions of objects. In practice, such diagrams may be decomposed into several interconnected interaction diagrams. In many cases, an interaction diagram models one nominal scenario and a number of error/exceptional (alternative) scenarios, where the object has to react appropriately. So, Interaction diagrams may also be seen as modeling alternative execution sequences of operations belonging to particular scenario within application domain. Complex conditions need to be tested by exercising operation sequences under several alternative

conditions. These alternatives correspond to the different combinations of truth values of predicates in a path realization condition such that this condition holds true.

Several kinds of faults can arise during integration: interaction faults, message synchronization faults and dependency faults due to object interactions. Thus, testing each class independently does not eliminate the need for integration testing. We propose a technique for integration testing accommodating the object message and condition information associated with the scenario, which are otherwise ignored. In this paper we propose an automatic test case generation method using UML [16] models. We use sequence diagram as a source of test case generation. For generating test data, sequence diagram alone may not be enough to decide the different components, i.e. input, expected output and pre- and post- condition of a test case. We propose to collect this information from the use case template and sequence diagram. Use cases, their corresponding sequence diagrams can be used as a source of relevant information for testing purposes. In this paper, we generate use case sequences from use case diagram. We propose a Concurrent Control Flow Graph (CCFG) generated from sequence diagrams of use case sequence, to support control flow analysis of UML 2.0 inter and intra-sequence diagrams of an use case scenarios [1, 30]. Our generated test suite aims to cover various interaction faults, message sequence faults as well as scenario dependency faults. These are associated with the object-oriented systems for which the use case dependencies and inter and intra-sequence diagrams are considered. Thus we can able to generate test cases for integration testing and those can also be lead to system testing after combining all the test cases of all the scenarios of the system. So, the test cases generated can detect use case sequence and initialization, and object interaction and message operational faults.

The rest of the paper is organized as follows. Related work is discussed in Section 2. UML and related diagrams are discussed in Section 3. Basic Terminology and coverage criteria are presented in Section 4. The faults detected in our approach are also described in this section. Our proposed approach for test case generation is discussed in Section 5. In this section, we convert UML model into graph and then generation of test cases from the graph is presented. Implementation of our approach is discussed in Section 6. Result and analysis of coverage are presented in Section 7. Comparisons with related work are described in Section 8. Finally, Section 9 concludes the paper.

2- RELATED WORKS

Winter [19] has proposed several coverage criteria based on use case diagrams which are adaptations of the Myers' criteria [13]. These are use case step coverage, use case branch coverage, use case scenario coverage, use case boundary body coverage, and use case path coverage.

Bertolino and Basanieri [2] proposed a method to generate test cases using

the UML use case and interaction diagrams (specifically, the message sequence diagram). It basically aims at integration testing to verify that the pretested system components interact correctly. They use category partition method [28] and generate test cases manually following the sequences of messages between components over the sequence diagram.

Several research attempts have been reported on scenario coverage based system testing [10, 11, 23]. These attempts are black box approaches and do not take into consideration the structural and behavioral design. Further, these work [11, 23] require using their proposed custom modeling notations. Hartmann et al. [10] proposed an automatic test case generation methodology based on the interactions between the system and its user. To model interactions, Hertmann et al. [10] semi-automatically converted the textual description of use cases into activity diagrams. Their [10, 11] approach manually annotates the design before test generation. The annotated activity diagrams are then processed automatically to generate a set of textual test procedures called executable test scripts. Riebisch et al. [11] generated system-level test cases from usage models. A usage model is derived from state diagrams, and is not amenable to full automation. The approach proposed by Tonella et al. [24] generates test cases based on UML sequence diagrams that are reverse engineered from the code under test.

Briand and Labiche [5] describe the TOTEM (Testing Object orienTed systEms with the unified Modeling language) system testing methodology. System test requirements are derived from early UML analysis artifacts such as, use case diagrams and sequence diagrams associated with each use case and class diagram. They capture the sequential dependencies between use cases into the form of an activity diagram with the intervention of application domain experts and derive test cases from it. Based on these sequential dependencies, they generate legal sequences of use cases for test case generation. Their approach is, in essence, a semi-automatic way of scenario coverage with prespecified initial conditions and test oracles. They proposed all-statement paths coverage criteria by assuming a strict mapping from requirements to code.

Frohlich and Link [23] presented a method to generate test cases from textual description of use cases. They first transform use cases into a state machine representation in which the states are abstractions representing the interval between two successive messages sent to the system by a user. Their test case generation approach essentially converts the test case design problem into a planning problem and uses an AI planning formalism to derive a test suite from the state model.

Falk Fraikin and Thomas Leonhardt [22] introduced the notion of testable sequence diagrams and SeDiTeC, a test tool that supports early automated testing based on testable sequence diagrams with pre- and post-conditions. For validation of tests, SeDiTeC captures the information such as method parameter, return values of all methods, etc. for all objects, which are relevant in the sequence diagrams and with the help of code instrumentation. This information is then used to form the observed sequence diagrams. Finally, observed

sequence diagrams are compared with the input sequence diagrams corresponding to the implementation under test. To support testing at the beginning of the implementation phase, SeDiTeC is also capable to generate test stubs.

Ghosh et al. [9] have discussed some coverage criteria for testing class diagrams. They include Association-End Multiplicity (AEM) criterion, Generalization (GN) criterion and Class Attribute (CA) criterion. Binder [3] proposed condition/iteration coverage criteria for sequence diagrams. Rountev et al. [18] proposed four different criteria: All-IRCFG-paths, All-RCFG-paths, All-RCFG-branches, and All-Unique-branches. IRCFG {Inter-procedural Restricted Control-Flow Graph) was obtained using reverse engineering.

Samuel et al. [31] presented a novel methodology for test case generation based on UML sequence diagrams. They 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, test data are generated. They have used a test adequacy criterion named slice coverage criterion. They generate test cases for cluster/integration level testing.

Sharma et al. [20] transformed a UML use case diagram into a graph called use case diagram graph (UDG) and sequence diagram into a graph called the sequence diagram graph (SDG) and then integrating UDG and SDG to form the System Testing Graph (STG). The STG is then traversed to generate test cases for system testing. They [20] have used state-based transition path coverage criteria for test case generation.

Nayak et.al. [34] proposed an approach of synthesizing test data from the information embedded in model elements such as class diagrams and sequence diagrams. They used OCL constraints. In their approach, they annotated a sequence diagram with attribute and constraint information derived from class diagram and OCL constraints and map it onto a structured composite graph called SCG. The test specifications are then generated from SCG.

3- RELATED UML DIAGRAMS

In object-oriented software, the complexity of the software is not only associated with functions and procedures but also with how the procedures and classes are connected and how objects communicate. In this section, we have described the UML and related UML diagrams used in this paper for test case generation.

UML is a language for specifying, visualizing, constructing and documenting the artifacts of software systems [4]. UML provides a variety of diagrams that can be used to present different views of an object-oriented system at different stages of the development life cycle [16]. Testing techniques which are based on the UML involve the derivation of test requirements and coverage criteria from these UML diagrams. According to UML 2.0 specification [15], seven UML diagrams can be used to specify the behavior of a system. These diagrams are activity, sequence, collaboration (also called communication),

interaction overview, Timing, Use case and state machine diagrams. Sequence and communication diagrams provide message level details of a system, which are needed for Control Flow Analysis (CFA). UML diagrams can be divided into three broad categories: structural, behavioral and interaction diagrams. The UML structural diagrams are used to model the static aspects of the different elements in the system, whereas behavioral diagrams focus on the dynamic aspects of the system. Our test generation methodology uses information present in two diagrams, namely use case and sequence diagrams. A use case comprises different possible sequences of interactions between the user and the computer. Each specific sequence of interactions in a use case is called a scenario. Sequence diagrams describe how a set of objects interact with each other to achieve a behavioral goal. UML 2.0 introduces the concept of a combined fragment [16] to capture complex procedural logic in a sequence diagram.

A use case is an abstraction of a system response to external inputs. It accomplishes a task that is important from user's point of view [3]. Thus, a use case focuses on only those features visible at the external interfaces of a system. A use case instance defines particular input values and expected results. Use case diagrams do not present architectural models, user-interface models or workflow models. Use case lacks following necessary elements of test design:

1. Domain definition of each variable that participates in a use case.
2. Required input/output relationships among use case variables (called parameters).
3. Sequential constraints among use cases.

Use case scenarios are usually not executed in arbitrary orders. Some use case scenarios need to be executed before others. They may have <<extend>> and <<include>> dependencies as well as sequential dependencies which stem from the logic of the business process that the system supports. Hence, data comes in or goes out of the system through use cases. An actor gives input and receives output information through use cases. Thus, we can have dependencies and constraints between use cases for each actor. So use case diagrams can be ornamented with extra ad-hoc information to show relationships and dependencies among use cases of a system [35].

A **sequence diagram** shows a set of interacting objects and the sequence of messages exchanged among them. In a sequence diagram the emphasis is on the time ordering of the messages [4]. Many researchers discussed the use case instance-sequence diagram relationship [5, 20]. In the basic testing practice applied to sequence diagrams, all end-to-end paths should be identified and exercised. This is equivalent to the requirement to identify and exercise all transitive relations formed by the variation of client-sends-message-to-server. Prior to UML 2.0, sequence diagrams were required to show only a single collaboration [3]. The implementation of a use case requires several sequence diagrams. The earlier sequence diagrams posed some additional problems:

- Representing complex control was difficult i.e. all the variant paths could not be represented.
- The distinction between a conditional message and a delayed message was weak.

3-1 UML 2.0 SEQUENCE DIAGRAMS

Sequence diagrams are essential UML artifacts for modeling the behavioral aspects of a system [4, 15]. The diagrams are particularly well-suited for object-oriented software, where they represent the flow of control during object interactions. A sequence diagram shows a set of interacting objects and the sequence of messages exchanged among them. The diagram may also contain additional information about the flow of control during the interaction, such as conditions (e.g. “if condition c then send message m else send message n”) and iteration (e.g. “send message m multiple times”) or state-dependent behavior [15]. The UML 2.0 [15] syntax of Sequence diagrams with respect to inter and intra-SDs is used in this work. Comparing to UML 1.x, UML 2.0 has proposed a set of new features to sequence diagrams. Some of the new features are illustrated with an example given in Figure 1.

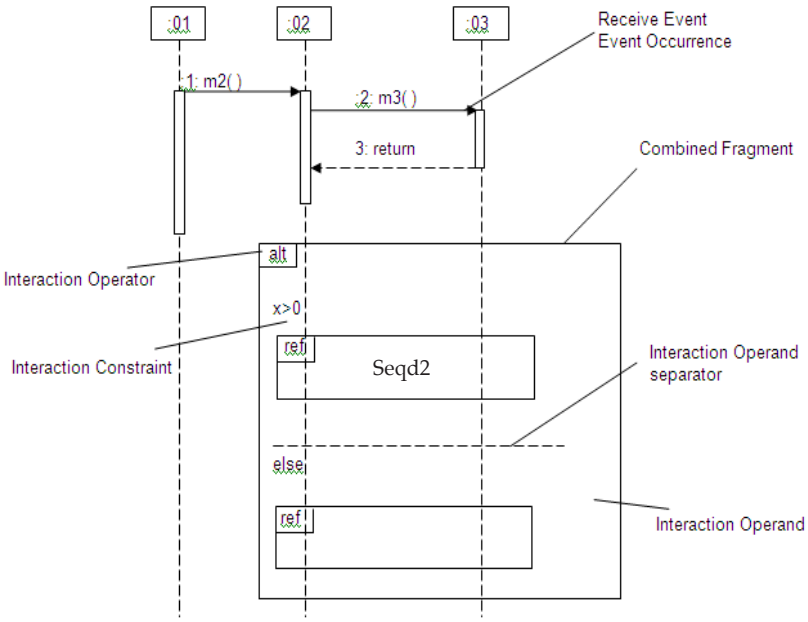

Figure 1Seqd1: An example of UML 2.0 Sequence Diagram

An *interaction* is a sequence of messages passed between objects to accomplish a particular task. The rational behind defining interactions is to reuse them in order contexts as interaction occurrences. For example, sequence diagram Seqd1 in Figure 1 shows an interaction among objects O1, O2 and

O3. An *InteractionOccurrence* is a symbol that refers to an interaction that is used within another interaction or context. Seqd2 and seqd3 are two interaction occurrences of Seqd1 which refer to sequence diagrams seqd2 and seqd3. *EventOccurences* represent moments in time to which actions are associated. It is the basic semantic unit of interactions. Event occurences are ordered along a lifeline. A message has two types of Event Occurences: *Send Event* and *Receive Event*. The *Send Event* is at the base (source) of the message arrow where the message departs from the lifeline of the sending object, while *Receive Event* is at the arrow head of the message arrow where the arrow hits the lifeline of the receiving object. The *Receive Event* of message m3 is shown in Figure 1. A *CombinedFragment* consists of one or more *Interaction Operands*. A *CombinedFragment* has an *Interaction Operator* that defines the number of allowed *Interaction Operands* and how the messages in the Combined Fragment will be treated. As an example, a Combined Fragment with *alt* Interaction Operator is shown in Figure 1. An *Interaction Operand* describes a grouping mechanism inside combined fragments. Interaction operands are features similar to Interactions, except the fact that they are part of a combined fragment. Figure 1 has two interaction operands. An *Interaction Operator* defines how to use the interaction operands within the context of the combined fragment. The following Interaction operators are defined: alternatives(alt), option(opt), break(break), parallel(par), weak sequence(seq),strict sequence (strict), negative (neg), critical region(region), ignore/consider (ignore/consider), assertion(assert), and loop(loop).

Alternatives(alt): It provides various alternatives, out of which only one alternative will be taken. The interaction operands are evaluated on the basis of specified guards. An else guard is provided that evaluates to TRUE if and only if all guards of the other Interaction Operands evaluate to FALSE.

Option(opt): It defines an optional interactions segment. The model for an *opt* combined fragment looks like an alt that offers only one interaction.

Break(break): It is the shorthand for an Alternative operator where one operand is given and the other assumed to be the rest of the enclosing Interaction Fragment. In the course of executing an interaction, if the guard of the break is satisfied , then the containing interaction abandons its normal execution and instead performs the clause specified by the break fragment.

Parallel (par): It supports parallel execution of a set of Interaction Operands.

Loop (loop): It indicates that the interaction operand will be executed repeatedly and also includes a mechanism to stop the iteration.

The detailed description of each of the interaction operators can be found in [15].

In UML 2.0, a message can be one of the following two types:

- *Operation call*: It expresses the invocation of an operation on the receiving object. An operation call must match the signatures of an operation on the target object (receiver of the message).

- *Signal*: It represents a message object sent out by one object and handled by the other object that is equipped to respond to it.

UML also provides different types of messages. Message sorts identify the sort of communication reflected by a message. The sorts of messages supported are defined in a list called *MessageSort* as:

- *SynchCall*: synchronous call
- *AsynchCall*: asynchronous call
- *SynchSignal*: synchronous signal
- *AsynchSignal*: asynchronous signal

UML 2.0 allows SDs to refer to each other through inter-SD control flow using the concept of interaction occurrences in UML 2.0.

4- BASIC CONCEPTS AND TERMINOLOGY

In this section, we present certain basic concepts and terminologies associated to our work that are used in later sections. We first present the intermediate representation of UML models and then different coverage criterion for test case generation. Then the fault model is given to describe different faults used in our proposed work.

Definition 1. Use case Dependency Graph (UDG): It is a directed graph, defined as UDG= {N, E} where N is the set of nodes, which represent use cases and E is the set of edges. Edges of UDG called dependency edges represent dependencies among use cases. The edges of a UDG represent the control dependences among use cases.

Definition 2. Control Flow Graph (CFG) : A Control Flow Graph (CFG) [27] is a static representation of a program/model that represents all alternatives of control flow. For example, both choices of a condition are represented in a CFG. A cycle in a CFG may imply that there is a loop in the code. Control flow paths (CFP) are the possible paths from the start node to the end node in CFG, which show the different paths a program/model may follow during execution.

Definition 4. Concurrent Control Flow Graph (CCFG): A concurrent control flow graph (CCFG) [32] G of an activity graph M is a directed graph (N, E, Start, Stop), where each node $n \in N$ represents an activity node of the activity graph M, while each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are unique nodes representing entry and exit of the model M, respectively. There is a directed edge from node a to node b if control may flow from node a to node b.

In UML 2.0 SDs, the *par* interaction operator is a new feature in UML 2.0 to support parallel execution of a set of interaction fragments. The impacts of the above two modeling features, lead to concurrency of intra and inter-SDs. Concurrency resulting from the above two modeling features has to be taken into account when analyzing the control flow in SDs. However, such concur-

rency can not be analyzed by conventional CFGs. A CCFG will be generated for sequence diagrams of use case scenarios. CCFG has been proposed to show inter and intra-SDs control flow. CCFG analyzes the concurrent control flow of SDs. In cases where a SD calls (refers to) another SD, there will be control flow edges connecting their corresponding CFGs to form an inter-SD CCFG. Inter-SD CCFG here is similar to the concept of inter-procedural CFG [27]. Similar to the concept of inter-procedural CFG and CFPs [27], inter-SD CCFG and CCFPs can be defined.

Concurrent Control Flow Path (CCFP): A CCFP is made by traversing from the initial node to the final node and concatenating all the nodes in the path. Similar to conventional CFP, special considerations regarding decision nodes should be made in terms of conditions and loops, i.e. two different CCFPs for true/false edges of a condition should be derived.

4-1 TEST COVERAGE CRITERIA FOR SEQUENCE DIAGRAM

Test coverage criteria [3] are a set of rules that guide to decide appropriate elements to be covered to make test case design adequate. We discuss the existing test case coverage criteria followed by our proposed criterion, which we consider as an improved test coverage criterion. Several testing approaches [10, 12, 22, 23, 31] have been proposed in literature to consider different test adequacy criteria for sequence diagrams. In this section, some of the test adequacy criteria have been discussed.

All-Messages-Criterion (AMC): Signals for each message on the link connecting two objects should appear at least once in an adequate test. This criterion ensures that all the messages between any two objects occur in the test. Thus for each message, we have to account for the corresponding test case. It is the derivation of AEM criterion stated in [9]. But this criterion is infeasible since the number of messages in all the paths can be exponential to the number of sequence diagrams.

All-Path-coverage-Criterion (APC): A set of concurrent message paths P satisfies the all-message-paths coverage criterion if and only if P contains all start-to-end message paths in a sequence diagram [14]. A start-to-end message path in a sequence diagram is a sequence of messages that begins with an externally generated event and ends with the production of a response that satisfies this event.

Full-Predicate-Coverage-Criterion (FPC): A test set T satisfies the full predicate coverage criterion if and only if for each clause c in each condition in a sequence diagram there exist t1 in T such that t1 causes c to evaluate to TRUE and there exists t2 in T such that t2 causes c to evaluate to FALSE while all other clauses in the condition have values such that the value of the condition will always be the same as the clause under test [9]. This criterion requires all the predicates are checked i.e. all possible combinations of the different predicates in the condition are checked.

Condition-Coverage-Criterion (CC): Given a test set T and sequence diagram SD, T must cause each condition in each decision to evaluate to both

TRUE and FALSE. This criterion covers all conditions in the graph. They include conditions and as well as loops [7].

Branch-Coverage-Criterion (BC): Given a test set T and sequence Diagram SD, if a message is sent under some condition c, the set of test cases should ensure that at least one path covers the condition with c is FALSE. This criterion was initially proposed by Binder [3]. The primary difference between BC and CC is that in BC if a condition appears more than once and if it is covered at least once, the criterion is satisfied.

Iteration-Coverage-Criterion (IC): Given a test set T and Sequence Diagram SD, for each loop L in SD, T must cause the loop to be either bypassed or taken for the minimum number, to be taken at least once or to be taken for the maximum number of iterations. This criterion requires all iterations to be covered once, twice and K times. It was proposed by Binder [3].

4-2 FAULT MODEL

Every test strategy targets to detect certain categories of faults called the fault model [3]. Our test strategy is based on the following fault model.

Interaction fault*:* In object-oriented programs, sequences of messages are exchanged among objects to accomplish some operations of interest [3, 9]. Several faults such as incorrect response to a message, correct message passed to a wrong object or incorrect message passed to the right object, message invocation with improper or incorrect arguments, message passed to yet to be instantiated objects, incorrect or missing output etc. may occur in an interaction [9].

Scenario fault*:* A sequence diagram depicts several operation scenarios. Each scenario corresponds to a different sequence of message path in the sequence diagram. For a given operation scenario, sequence of message may not follow the desired path due to incorrect condition evaluation, abnormal termination etc. [3].

Synchronization fault: This fault occurs when one use case is executed before completion of execution of group of all preceded use cases. In the LIS given in Figure 2 both *AddUser* and *AddItem* need to be executed before *IssueCopy* can, as modeled by a *join* synchronization. *AddTitle* must be executed first in order to execute *RemoveTitle*, but the execution of *AddTitle* does not require the execution of *RemoveTitle*. Like wise in sequence diagram this fault occurs when some objects send messages before completion of execution of group of all preceded messages messages are to be synchronized [25].

Operational Fault*:* A system leads to an operational fault if each use case does not obey the desired input-output relationships [20].

Dependency Fault*:* Errors may occur when a use case begins its execution without satisfying the required dependency [20]. An use case may invoke another use case. The invoked use case is dependent on main use case. That is, one use case may include other use cases. For example in LIS, the Remo-

veTitle use case invokes FindTitle use case. Further, in some cases one use case must be executed before another. For example, in an on-line purchase system, an order should be created through a *Create Order* use case before processing a *Process Order* use case.

5- PROPOSED MODEL

Our proposed approach for test case generation is based on the artifacts produced at the end of the analysis stage of the development. The artifacts include use case diagram and sequence diagram for each use case. Use cases have sequential constraints that have to be specified. Such constraints are the direct results of the logic of the business process the system claims to support. For example, in a Library Information System (LIS), a user needs to register before borrowing a book. In this case, registration and borrowing correspond to two different use cases, and for a library user one has to be performed before the other i.e. registration process should be performed before borrowing. Therefore, in addition to artifacts, the approach requires the specification of such sequential constraints in the form of an activity diagram. The test case generation method consists of the following steps:

1. Deriving Use Case dependency sequences.

2. Deriving test requirements from sequence diagrams.

3. Deriving test cases.

5-1 DERIVING USE CASE DEPENDENCY SEQUENCES

Use cases represent the high-level functionalities provided by the system to the user. These are good sources for deriving test requirements of the system. Usually, they are not independent of each other. Possible execution sequences of use cases are to be identified for planning of test cases for use cases. Thus, for testing, the sequential constraints and dependencies should be derived from use cases. The generation use case dependency sequence from the use case diagram consists of the following steps:

1. First, we construct the activity diagram for the use case diagram.
2. Then, we construct the Use case Dependency Graph (UDG) by using activity diagram of use cases.
3. Then, we generate the use case dependency sequences from the UDG.

5.1.1 Constructing activity diagram from use cases of an actor

Use cases are represented by means of using syntax of an activity diagram for each actor in the system. In use case activity diagram, the vertices (action nodes) are use cases with the dependent data variables and the edges are sequential dependencies between use cases. An edge between two use cases implies that the use case in the tail should be executed first in order to execute the use cases towards the head. In addition, several use cases can be

executed independently, i.e. they are not sequentially dependent. This is modeled by *join* and *fork* synchronization bars in the activity diagram. The activity diagram for the Librarian actor in the LIS is shown in Figure 2.

We use LIS example to illustrate some of the concepts presented in Figure 2. The sequential dependency between *AddTitle* and *RemoveTitle* specifies that *AddTitle* must be executed first in order to execute *RemoveTitle*, but the execution of AddTitle does not require the execution of *RemoveTitle*. Both *AddUser* and *AddItem* need to be executed before *IssueCopy* can, as modeled by a *join* synchronization. Here, we can bypass the loop or take it only once to test loops.

5.1.2 Constructing UDG from activity diagram of use cases of an actor

Next the dependencies between use cases represented using activity diagram is to be converted to Use case Dependency Graph (UDG). In use case dependency graph (UDG), the vertices are use cases with dependent data variable (action nodes in activity diagram) and the edges are dependeny edges of the activity diagram, showing dependencies between use cases. The fork and join bars are deleted in UDG. The UDG for the activity diagram in Figure 2 is shown in Figure 3.

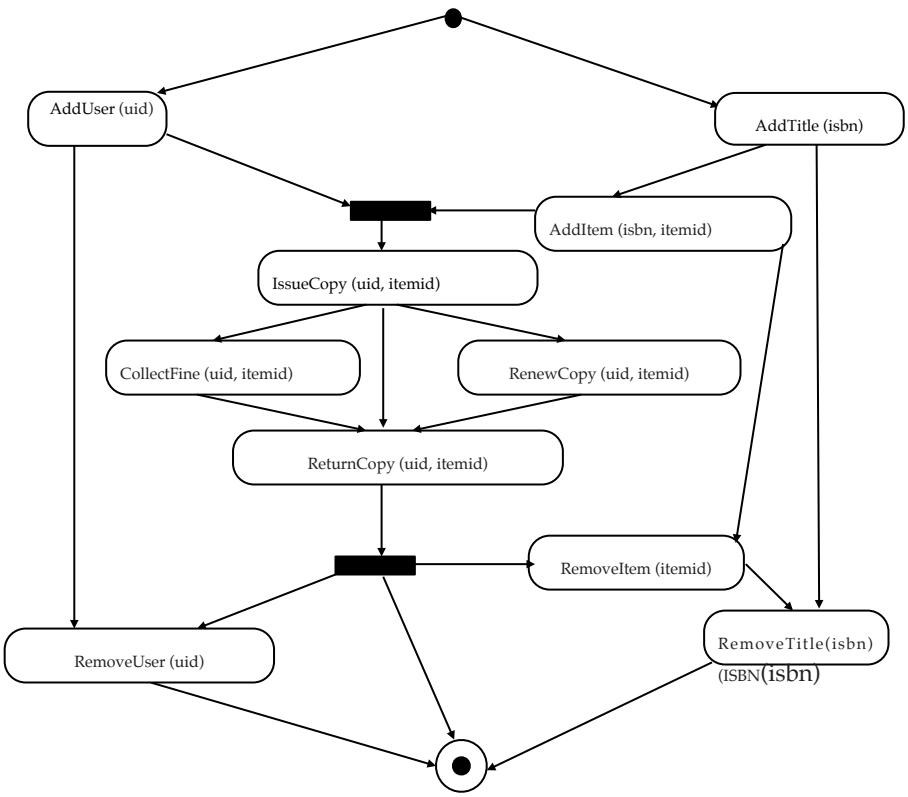

Figure 2 Activity diagram for Librarian actor

5.1.3 Generating Use case dependency sequences

The next objective is to generate use case sequences, in a semi-automated way. They will constitute the first component of the test requirements. Paths in the activity diagram represent a possible life history for an object type (e.g. the path AddUser. RemoveUser) or a combination of object types (e.g. path AddTitle. AddItem. RemoveItem. RemoveTitle). Paths are first determined through a simple depth-first search in the directed graph corresponding to our activity diagram. Then we need to determine dependencies in terms of actual parameter values between the use cases in a path. For instance, in path AddTitle. AddItem. RemoveItem. RemoveTitle, parameter *isbn* for use case AddItem must be identical to parameter *isbn* in AddTitle. Such dependencies among actual parameter values are needed in order to identify the data flow between use case executions. That will be necessary for the generation of test input data. We can find such dependencies by adding actual parameters into the use case sequences, which are referred to as *parameterized use case sequences*.

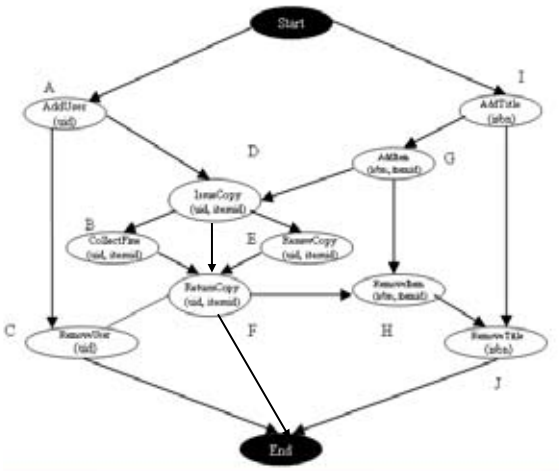

Figure 3 - UDG of activity diagram of Figure 2

If we execute any path between AddTitle and RemoveTitle, such as the one above which is an output of the depth first search, use cases AddTitle and AddItem are executed at most once. That is, at best, each title corresponds to only one item in the library. If we want to test our Library Information System (LIS) under more realistic situation, we obviously need more than one item per title and more than one copy per user and item. We also need to proceed with large number of users, items, and loans if we want to test the scalability of our system. The number of times use case parameters must be instantiated is determined by the information provided by tester in terms of the scale of the test to take place, e.g., the number of users, or items per title. In the simple

example where 2 items and 1 title must be created, we get two instantiated use case sequences (SEQ1, SEQ2) from the parameterized sequence above, where title1, item1, and item2 are symbolic values for the *isbn* and *itemid* parameters:

SEQ1:AddTitle(title1).AddItem(title1,item1).RemoveItem(item1).RemoveTitle(title1)

SEQ2:AddTitle(title1).AddItem(title1,item2).RemoveItem(item2).RemoveTitle(title1)

Then all instantiated use case sequences are combined to generate *complete sequences*. After maintaining the complete sequences of SEQ1 and SEQ2 , a possible combined sequence is:

AddTitle (title1). AddItem (title1, item1). AddItem (title1, item2).RemoveItem (item2) . RemoveItem (item1). RemoveTitle(title1)

The instantiated use case sequences must be combined carefully to preserve the use case dependencies and avoid the duplication of instantiated use cases. This can be formalized by the concept of interleaving [9] (where || denotes interleaving in a sequence). In our example, the two sequences (SEQ1,SEQ2) would be combined as:

AddTitle (title1).((AddItem (title1,item1). RemoveItem (item1))||(AddItem (title1,item2).RemoveItem (item2))). RemoveTitle (title1)

Common instantiated use cases will be identified across pairs of sequences and, for each pair, all the subsequences in between each common, instantiated use cases will be combined using interleaving. This procedure will be performed for all pairs of sequences extracted from the activity diagram. Furthermore, when performing the interleaving to generate sequences, we will only sample a subset of all possible resulting sequencing in order to avoid a combinatorial explosion. Parameterized use case sequences are derived by using depth-first search algorithm through the directed graph capturing the activity diagram.

Algorithm: GenUcSeq-Generating Use case Sequence

Input : Use case Dependency Graph(UDG) representing an Activity,
Input parameter values(IP)
Output: Test Sequences
Let US be set of Use Case Sequences
Let PD be set of parameterized dependencies.
Step 1: Traverse UDG using depth-first search
Step 2: If loop present
Step 3: Make one iteration
Step 4: End if
Step 6: for each pair(A,B) of instantiated sequence in PD
Step 7: US=Merge(A,B)
Step 8: Store all use case sequences US
Step 9: End

Hence in our example, Figure 3 shows the UDG of the activity diagram of the use case given in Figure 2. The parameterized use case sequences are de-

rived by using GenUcSeq algorithm, which are given below.

A (uid).C (uid)

I(title). J(title)

I(title).G (title, item).H (item).J (title)

(A(uid)||I(title).G(title, item)).D(uid,item).F(uid,item).(C(uid)||H(item).J(title))

5-2 DERIVING TEST REQUIREMENTS FROM SEQUENCE DIAGRAMS

We construct a sequence diagram for each use case in the system. It shows how the use case is realized through the interactions of objects. UML 2.0 sequence diagram not only shows message sequence between objects but also represents the control flow among the use cases. Because one use case may invoke other dependent use cases during execution of its scenario. In some cases, to execute one use case, it must be synchronized with other use cases within the system. For this UML 2.0 sequence diagram can show object interactions of number of dependent or synchronizing use case scenarios in a single diagram. Thus, one sequence diagram may contain number of child (dependent/ synchronizing) sequence diagrams (shown in Figure 1). So, we can generate test cases to detect dependency and synchronization faults of use case scenarios and messages. Hence, testing is focused on a level down into details of object interactions to derive test cases for testing sequences of use cases within a scenario. The generation of test cases from the sequence diagram consists of the following steps:

Step 1: The sequence diagram is transformed into the corresponding Activity Graph by the mapping rules given in Table 1.

Step 2: Concurrent Control Flow Graph (CCFG) is obtained from the Activity Graph.

Step 3: All possible Concurrent Control Flow Paths (CCFP) are identified from the CCFG.

Step 4: Path realization conditions in CCFP are identified i.e. all the conditions to traverse the CCFP are identified. These include predicates in the condition.

Step 5: Operation sequences for a particular condition are specified. In this step, the concurrent nodes in the CCFP are interleaved to generate all possible scenarios.

Step 6: After identifying the sequences, the decision table is constructed to generate test cases. The decision table contains all possible conditions derived in Step 4 and all the messages to the actor in Table 2.

Step 1: Converting Sequence Diagram to Activity Graph

The Sequence diagram is converted into an activity graph. This step was pro-

posed in order to incorporate features of Activity diagram as well as sequence diagrams in a single representation. This is done using conversion rules as proposed in Table 1 [30].

Step 2: Obtaining CCFG from Activity Graph

The constructed Activity graph is converted into a Concurrent Control Flow Graph (CCFG) as described in Section 4. The references to other Activity Graph (Interaction Occurrences) are also included into the CCFG. Thus the new graph obtained does not contain the create () and destroy () messages. So, we obtain a complete graph, with all possible scenarios of the use case. The CCFG for RemoveTitle is shown in Figure 6.

Step 3: Extracting CCFP from CCFG

The next step is extracting Concurrent Control Flow Paths from the CCFG. All the different branches taken are identified and all possible concurrent paths are derived. If the number of branches is *n*, then the number of different possible paths can go up to 2*n. In the above example, we obtain five different paths. The different possible paths obtained are:

1. a-b-d-t 2. a-c-t 3. a-b-d-m-n-o-p 4. a-b-d-m-n-o-q-r-s

5. a-b-d-m-n-o-q-r-q-r-s

Table 1 Mapping of UML 2 sequence diagram to activity graph

Rule#	SD feature	Activity Graph feature
1	Interaction Fragment	Activity
2	First Message end	Flow between Initial node and first control node
3	Synch Call/Synch Signal	Call Node
4	Asynch Call or Asynch Signal	(Call node + Fork Node) or Reply Node
5	Message Send Event and message Receive Event	Control Flow
6	Lifeline	Object Partition
7	Par Combined Fragment	Fork Node
8	Loop Combined Fragment	Decision Node
9	Alt/opt Combined fragment	Decision Node
10	Break Combined Fragment	Activity Edge
11	Last message ends	Flows between ending control nodes and Activity Final Node
12	Interaction Occurrence	Control Flow across Activity Graph
13	Polymorphic message	Decision Node
14	Nested Interaction Fragments	Nested Activity graph

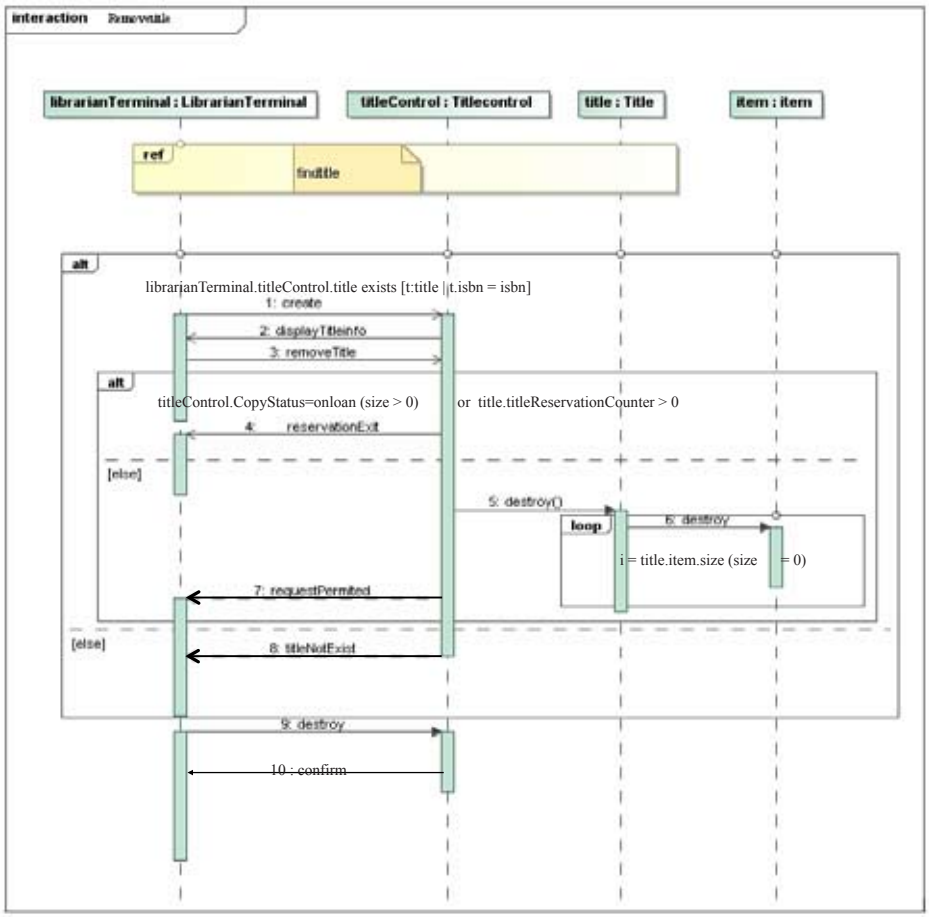

Figure 4 Sequence diagram for Remove Title use case with its dependent use cases

Step 4: Identifying Path Realization Conditions

All possible predicates using all predicate coverage criteria are extracted from the CCFPs of CCFG. Thus for the RemoveTitle use case, we obtain the following predicates by using the sequence diagram given in Figure 4.

P1: titleControl.title exists(t:Title| t.isbn=isbn)

P21: titleControl.copyStatus=onLoan and size > 0

P22: Title.titleReservationCounter > 0

P3: I = Title.item.size

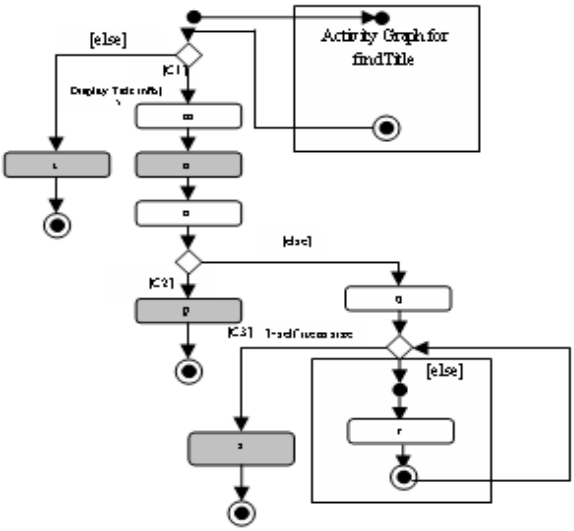

Figure 5 - Activity Graph for RemoveTitle sequence diagram given in Figure 4

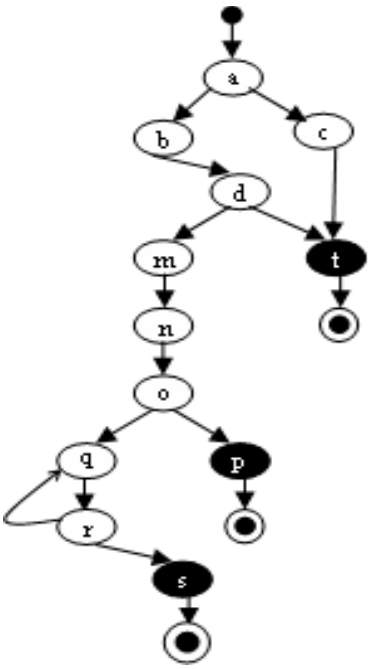

Figure 6 CCFG for the activity graph of RemoveTitle sequence diagram of Figure 4

Step 5: Specifying Operation Sequences for the Predicates

The predicates obtained above are set and unset to traverse all possible paths in CCFG. It refers to finding two sets of values for the variables in a Predicate P, such that one set satisfies the predicate and another does not. The messages to actor (MtA) are considered as the expected output for the operation sequences.

Predicates:

A. P1

B. ~ P1

C. P1&P21&P22

D. P1&~P21&P22

E. P1&P21&~P22

F. P1&~P21&~P22

G. P1&~P21&~P22&P3

H. P1&~P21&~P22&~P3

Messages to Actor:

i. Title does not exist.

ii. Copy on issue or Reservation exists.

iii. Display Titleinfo and iteminfo.

iv. Request permitted for remove title.

v. Confirm the removal

Step 6: Generating Decision Table

The different conditions obtained by setting and unsetting these predicates and the expected outputs from the sequence diagram (Figure 4) are given in Table 2. The algorithm GenSDTest is used to generate the decision table.

Algorithm: GenSDTest

Input : Sequence Diagram SD
Output: Test requirements in the form of decision table
Let MtA be set of messages to Actor
Let SS be set of sequences in CCFP
Step 1: Convert the SD into CCFG
Step 2: Identify all the CCFP in the CCFG
Step 3: For All CCFPs in CCFG
Step 4: Identify Path realization conditions(PRC)
Step 5: Record TS (Test Sequence)
Step 6: Record PRC
Step 7: Add all the messages to actor to MtA
Step 8: End For
Step 9: Construct decision table
Step 10: End

Table 2 Decision Table showing conditions and actions

Predicate Variant #	Condition								Action Messages to Actor				
	A	B	C	D	E	F	G	H	I	II	III	IV	V
A	Yes	No	No	No	No	No	No	No	No	No	Yes	No	No
B	No	Yes	No	No	No	No	No	No	Yes	No	No	No	No
C	No	No	Yes	No	No	No	No	No	No	Yes	Yes	No	No
D	No	No	No	Yes	No	No	No	No	No	Yes	Yes	No	No
E	No	No	No	No	Yes	No	No	No	No	Yes	Yes	No	No
F	No	No	No	No	No	Yes	No	No	No	No	Yes	Yes	No
G	No	No	No	No	No	No	Yes	No	No	No	Yes	Yes	Yes
H	No	No	No	No	No	No	No	Yes	No	No	Yes	Yes	No

5-3 Generating Test Cases

Finally, the test cases are generated by combining all the conditions of messages from each of the use case in an use case sequence. In other words, we need to go from use case sequences to use case variant sequences, using decision tables generated from sequence diagram. Test cases should cover all variants, at least once. So, test cases are generated by taking the predicates as input and Message to Actor (MtA) as output from the decision table. The columns model the initial conditions in which test cases must be run, the actions that are taken as a result of running the test cases. Namely, this corresponds to output messages being sent to actors. Test cases refer to finding two sets of values for the variables in a Predicate P, such that one set satisfies the predicate and another does not. The values set satisfying the predicate should contain boundary values and the set not satisfying the predicate should differ from the set satisfying the predicate only in the value of one variable by one. The decision table is to be accessed in the same way as a two dimensional matrix is read in order to get these predicates and outputs. Assuming, we would have a use case sequence of three use cases A, B, C having the different paths |A|, |B| and |C| respectively, the maximum number of variant sequences would then be |A|*|B|*|C|. Each variant will be covered by a test case for each tested operation sequence. The test cases generated for our example of use case *RemoveTitle* in LIS is given below:

Test Case1 (Title.isbn=isbn, Output: Display Titleinfo and iteminfo)

Test Case2 (Title.isbn≠isbn, Output: Title does not exist)

Test case 3 (Title.isbn = isbn, titleControl.copyStatus = onLoan and size > 0, Title.titleReservationCounter > 0, Output: Display Titleinfo and iteminfo , Copy on issue, Reservation Exist)

Test case 4 (Title.isbn = isbn, titleControl.copyStatus ≠ onLoan, Title.titleReservationCounter > 0, Output: Display Titleinfo and iteminfo, Reservation Exist)

Test case 5 (Title.isbn = isbn, titleControl.copyStatus = onLoan and size > 0, Title.titleReservationCounter = 0, Output: Display Titleinfo and iteminfo, Copy on issue, Reservation Exist)

Test case 6 (Title.isbn = isbn, titleControl.copyStatus ≠ onLoan, Title.titleReservationCounter = 0, Output: Display Titleinfo and iteminfo, Request permitted for remove title)

Test case 7 (Title.isbn = isbn, titleControl.copyStatus ≠ onLoan, Title.titleReservationCounter = 0, I = Title.item.size, Output: Display Titleinfo and iteminfo, Request permitted for remove title, Confirm the removal)

Test case 8 (Title.isbn = isbn, titleControl.copyStatus ≠ onLoan, Title.titleReservationCounter = 0, I ≠ Title.item.size, Output: Display Titleinfo and iteminfo, Request permitted for remove title)

6- IMPLEMENTATION

We have implemented a prototype tool in Java, called Comprehensive Test (**ComTest**). ComTest tool has been developed for generating test cases. It takes UML 2.0 use case and sequence diagrams in the form of XML 2.1[17] format, parses the XML file and makes a virtual graph and finally generates test sequences. The system is modeled using UML in a CASE tool like MagicDraw. Then the model is exported by the CASE tool in XML format. The XML format is an input to our proposed tool **ComTest** tool. ComTest takes the XML file and generates test scripts. These test scripts are given as input to a test execution Tool which executes the test script. The pictorial representation of working of ComTest is shown in the Figure7. The architecture of ComTest consists of three modules.

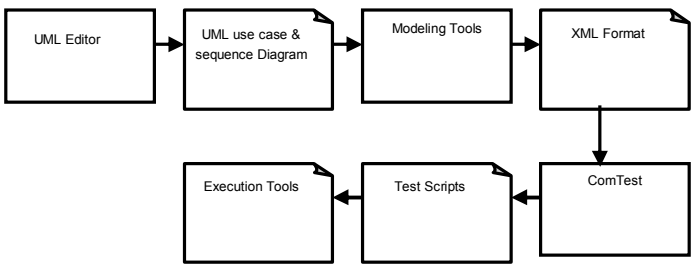

Figure 7Overview of ComTest tool

1. **XML parser**, which parses the file in XML format. The parser supports XMI1.4 and 2.0 formats.

2. **Handler**, which handles the parsed document and converts the UML Elements extracted into objects. It then logically prepares the whole UML dia-

grams (Intermediate Format).

3. **Test Case generator** takes the graph developed as input and traverses the graph. Then, it generates the test cases as specified in the respective methods.

The architecture of comTest with working sequence of above modules is shown in Figure 8.

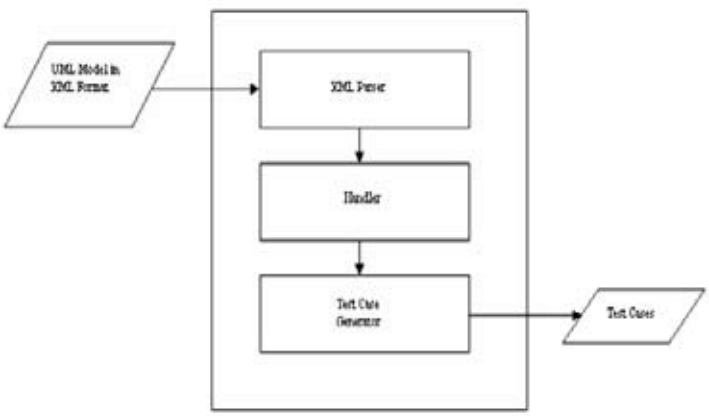

Figure 8 Architecture of ComTest

7- RESULTS AND ANALYSIS

In this section, we have first described the complexity of test case generation. Then, we have shown the coverage analysis of our proposed technique.

7-1 Complexity of Test Case Generation

The automation of the proposed methodology can be divided into 3 main parts.

1. Generating Use case sequences.

2. Generating test coverage path for each use case sequence from sequence diagram.

3. Integrating the Steps 1 and 2.

Generating Use case Sequences

a. The generation of parameterized use case sequences from the UML activity diagram can easily be done with a depth first search of the directed graph

capturing the activity diagram.

b. The derivation of instantiated use case sequences from the test scale information provided by the tester does not require complex algorithm. Thus, this step can be done in complexity proportional to number of sequences and the test data provided.

c. Test sequence can be extracted from the instantiated use case sequences by interleaving any two sequences. Thus we obtain n(n-1)/2 (e.g. n is number of sequences in two parameterized sequences) instantiated parameterized use case sequences and test sequence combinations having (p+q)!/p!q! variants, where p and q are lengths of each of the sequences respectively.

Generating test requirements from sequence Diagrams

a. The transformation of a sequence diagram into an activity graph is a direct step. It is developed directly after parsing the required data and applying the transformation rules. Hence, the complexity of this step is equivalent to the number of distinct UML artifacts present in the sequence diagram.

b. Transformation of the activity diagram to the corresponding CCFG is a simple depth-first search.

c. Obtaining CCFPs from the CCFG is a complex step. Each branch needs to be identified along with the concurrent components. Again it can be done with a depth first search but the complexity increases depending on the number of concurrent paths.

d. Test sequence control flow paths can be obtained from the CCFP by interleaving nodes in the concurrent path. This is a simple step.

e. Identifying the conditions for each of the test sequence control flow path can be done by simple traversal.

f. Construction of the decision table takes **n** steps where **n** is number of distinct paths obtained.

Complexity of Generating Final Test Cases

Generating final test cases from the use case sequences and test requirements derived from sequence diagrams is a complex step.

The complexity of this step=$M * N * \prod_{i=1}^{p+q} x_i$

Here, M = Total number of test sequences generated=n(n-1)/2 where n is number of different use case sequences obtained.

N = (p+q)! /p! q! where p and q are lengths of each of the combined use case sequences.

X_i = Number of variant paths (CCFPs) for i^{th} use case.

7-2 COVERAGE ANALYSIS

Various coverage criteria have been discussed in Section 4.1. We have followed the *full-predicate-coverage* criteria in this work. This criterion is stronger than Condition Coverage (CC), Branch Coverage (BC) and Iteration Coverage (IC) but weaker than All-Message Coverage (AMC) and All-Path Coverge (APC). Consider a sequence diagram (SD) with C conditions, P predicates, I iterations and M messages. In order to satisfy the AMC criterion, the tests should cover all the messages. Since there are lot of messages in a given sequence diagram, this criteria is impractical as the complexity will rise exponentially. But this criterion covers each and every message and hence it is the strongest one. For achieving the APC criterion, we need to cover all possible paths in the CCFG. But the number of these paths could be exponential in size. The CC and BC criteria do cover all the branches, but do not toggle each predicate that make the condition or branch that is taken. Suppose each condition has n predicates on an average, then the number of test cases will be less than a factor of *n*. If the same condition gets repeated k times in the same SD then the number of test cases can still reduce by a factor of $n \times k$. Since every message on the path is not traversed, any important input/output message could be missed. We have performed an experiment to evaluate the different coverage criteria. Table 3 shows the coverage by each of these criteria on our case study, the Library Information System, which is taken from [6]. Table 3 shows the different metrics like number of messages, paths, conditions, predicates and iterations for the Library Information System.

Table 3 Coverage Metrics for Library System

Use case	#Msgs.	#Paths	#Branches	#Predicates	#Iterations
Create User	9	2	1	1	0
Remove User	13	5	4	4	1
Add Title	9	2	1	1	0
Remove Title	16	4	4	5	1
Add Item	18	2	2	2	1
Remove Item	15	3	2	2	0
Borrow Loan copy	16	7	6	7	0
Return Loan copy	13	3	2	2	0
Renew loan	15	6	5	6	0
Collect Fine	6	3	2	2	0
Find Title	4	2	1	1	0
Search User	4	2	1	1	0
Make Reservation	8	2	1	2	0
Remove Reservation	5	2	1	1	0
Total	**146**	**45**	**33**	**37**	**3**

From the above data in Table 3, we can derive the result that compares different coverage criteria. Thus by using the FPC criterion, the coverage data for LIS is obtained and shown in Table. 4.

Table 4 Criteria Coverage for Library Information System

Use case	#Msgs. /Covered	#Paths /Covered	#Conditions /Covered	#Predicates /Covered	#Iterations /Covered
Create User	9/7	2/2	1/1	1/1	0/0
Remove User	13/11	4/4	3/3	3/3	1/1
Add Title	9/7	2/2	1/1	1/1	0/0
Remove Title	16/12	4/4	3/3	4/4	1/1
Add Item	18/13	2/2	1/1	1/1	1/1
Remove Item	15/12	3/3	2/2	2/2	0/0
Borrow Loan-copy	16/14	7/7	6/6	7/7	0/0
Return Loan-copy	13/11	3/3	2/2	2/2	0/0
Renew loan	15/13	6/6	5/5	6/6	0/0
Collect Fine	6/4	3/3	2/2	2/2	0/0
Find Title	4/3	2/2	1/1	1/1	0/0
Search User	4/3	2/2	1/1	1/1	0/0
Make Reser-vation	8/6	2/2	1/1	2/2	0/0
Remove Res-ervation	5/4	2/2	1/1	1/1	0/0
Coverage	**85%**	**100%**	**100%**	**100%**	**100%**

8- COMPARISON WITH RELATED WORK

In this section, we first provide a comparison of our work with other UML based test case generation approaches. Next, we present a comparison of our approach with other integration and system level testing approaches.

Even though model based test case derivation has been extensively discussed in the literature, work using UML models is scarce [1,2,5,10,14,20]. Offut et al. [14] proposed the idea of utilizing UML statechart diagrams, to generate test cases for unit level testing. Abdurazik et al. [1] have proposed a technique using the UML collaboration diagram for generating test cases. In contrast, our work derives use case sequences that should be part of the test plan. Now, using the sequence diagrams associated with use case, we have gone down one more level into details, and derive test cases to test sequences of use cases within the scenario.

Briand et al. [5] proposed a complete system testing methodology. They [5] used the widely used approach of finding sequential dependencies between use cases as proposed in [6] and the use case scenarios as basis of generating test sequences. These scenarios are then identified by interaction diagram (either a sequence diagram or a collaboration diagram). The work uses OCL to present the conditions, guards and messages. Also it makes some modifications to the sequence diagrams for ordering of messages. Our work bears some similarity with the work of [5] on deriving operation sequences from use case diagrams. The main difference lies in the fact that the authors [5] make use of path-wise regular expressions conditions from

models with analysis document like class descriptions and data dictionary rather than full predicate coverage generated from UML models. Furthermore, they [5] have used scenario coverage focusing more on use case diagram. But, we predominantly use the sequence diagram of UML 2.0 for generation of test cases. We have also made the comparison of coverage criteria in our approach.

Our approach is also comparable to the approach reported by Sharma et al. [20] in the sense that in our approach sequential message constraints among objects of the dependent and synchronized use cases are explored using UML 2.0 sequence diagram implicitly, where as in their work [20] they considered states of objects withn a scenario of single use case separately. Sharma et al. [20] generated the test case using structural coverage like use case and message path criterion from sequence diagram for system testing.. Our approach for test case generation uses coverage like full-predicate coverage criteria which also covers the structural coverage implicitly. We have also made the analysis of coverage criterion in our approach. The generated test cases detect the synchronization faults in addition to faults described in [20] and used in integration as well as system level testing.

The primary test coverage criterion for interaction diagrams was first identified by Abdurazik [1], and a comprehensive work was done by Ghose et al. [9]. They [9] identified four different criteria for testing interaction diagrams (1) condition coverage criterion, (2) full predicate coverage criterion, (3) each message in link criterion, and (4) all message path criterion. The work in [18] identifies coverage criteria for sequence diagrams. But testing each message in a SD is not feasible since a system can have very high number of message links. Since the work done was pre-UML 2.0, it did not cover iterations, asynchronous messages and concurrent paths. The work presented in [18] also uses similar test adequacy criteria for testing sequence diagrams, but it overcomes the short comings of [5, 9] in addressing these issues.

The testing approaches attempted in [11, 22, 23] focus on the coverage of use case scenarios. These works can be considered essentially black box in nature and do not take into consideration the structural and behavioral aspects. In comparison to the work [11, 23], our approach considers structural dependency as well as behavioral aspects of design model for deriving test cases. They [11, 22, 23] have used custom modeling notations.Further, these work [11, 22, 23] consider each use case independently and do not consider composition of scenarios from different use cases within a use case diagram. It essentially focuses on functional testing.

In comparison to [18, 20, 22], the advantage of our work is that it explores conditions, iterations, asynchronous messages and concurrent paths. We use use case diagram and corresponding sequence diagrams as compared to the work [11, 18, 22]. The test case generated in our approach will be used for integration as well as system testing. The work reported by Samuel et al. [31] to generate test cases using dynamic slicing criterion. They [31] have considered only one scenario of one use case using UML 1.X sequence diagram. In comparison to the work [31] we have generated test cases using depen-

dency and synchronization among use cases through UML 2.0 sequence diagram. Our work uses only UML diagrams as the input; i.e. it doesn't require input in non UML-formats as required by some approaches [5, 9].

In contrast to [36], we have used graph based approach on use case and UML 2.0 sequence diagram using coverage adequacy criterion. They [36] suggested a model based approach in dealing with object behavioral aspect of the system and deriving test cases based on the Tree structure coupled with Genetic algorithm. Lastly, some [12, 34] of the early approaches are interesting, but too general as they lack detailed, operational descriptions.

9- CONCLUSION

We have presented a strategy for integration testing which combines information from use case and sequence diagram. This methodology predominantly uses the sequence diagram of UML 2.0 for generation of test cases. It incorporates the new features of UML 2.0 sequence diagram such as interaction operand and constraints and combined fragment. Our proposed technique uses only UML diagrams as the input. It doesn't require input in non UML-formats .Our approach exercises object interactions in the context of use case dependencies to fulfill the requirements of the user. In our proposed approach, we derive use case sequences which are parts of the test plan. Now, using the sequence diagrams associated with sequences of use cases, we have to go down one more level into details of message sequences within a use case scenario to derive test cases for integration level testing object-oriented software. We construct Use case Dependency Graph (UDG) from use case diagram and Concurrent Control Flow Graph (CCFG) from corresponding sequence diagrams for test sequence generation. We focus testing on sequences of messages among objects of use case scenarios by traversing CCFP of CCFG. Our generated test suite aims to cover various interaction faults, message sequence faults as well as use case dependency and synchronization faults. These are associated with the object-oriented systems for which the use case dependencies and inter and intra-sequence diagrams are considered. The generated test cases may lead to the part of the test cases for system testing by combining it for all the use case scenarios together without mine it separately. Our proposed technique uses XML, which is latest de-facto standard for exchanging UML models. A semi-automated tool (ComTest) has been developed which takes the UML diagrams in the required format and generates the test cases. Finally, we have made an analysis and comparison with works based on other coverage criteria. Our work may be extended to include the test oracles and handling synchronizations in the CCFP. Also, our work may be extended for test data generation.

REFERENCES

[1] A . Abdurazik A, J. Offut. “Using UML collaboration diagrams for static checking and test generation”. In Proceeding of the 3rd International Conferece on the Unified Modelling language(UML), Lecture Notes in computer Science,Volume 1939, pp.383—395, York, UK, Springer-Verlag, GmbH October 2000.

[2] F. Basanieri, A. Bertomated ,E. Marchetti, A. Rinoline, G. Lombardi, and G. Nucerga. “An Automated Test Strategy Based on UML Diagrams”. In Proceeding of the Ericsson Rational User Conference, Upplands Vasby Sweden, October 10-11,2001,

[3] R.V Binder. “Testing Object-Oriented Systems Models, Patterns, and Tools”. Object Technology Series. Addision Wesley, Reading, Massachusetts, October 1999.

[4] G. Booch, J. Rumbaugh, and I Jacobson. “The Unified Modeling Language User Guide”, Object Technology Series Addision Weslley, 1999.

[5] L.C Briand and Y .Labiche. “A UML-Based Approach to System Testing”. In Proceeding of the 4th International Concepts, And Tools, Springer-Verlag, London, pp.194-208, October 01-05,2002.

[6] B.Bruegge and A.H. Dutoit. “Object-Oriented Software Engineering-Conquering Complex and Challenging Systems”, Prentice Hall, 2000.

[7] E.Dustin. “Effective Software Testing: 50 Specific Way to improve your Testing”. Addision-Wesley, 2003.

[8] H.Eriksson, M.Penker, B.Lyons,and D.Fado. “UML 2 Toolkit”, Wiley, 2003.

[9] S.Ghosh, R.France, C.Braganza, N.Kawane, A.Andrews, O.Pilskalns. “Test Adequacy Asessment for UML Design Model Testing”. In Proceedings of 14th International Symposium in Software Reliability Engineering (ISSRE), pp.332,2003.

[10] J.Hartmann, M.Vieira, H.Foster, and A.Ruder. “ A UML-based Approach to System Testing, Journal of Innovations system Software Engineering, Vol. 1, pp.12-24,2005.

[11] Riebisch M., Philippow I, and Gotze M., UMLBased Statistical Test Case Generation, in the Proceeding of ECOOP 2003, Springer Verlag,LNCS 2591, pp. 394-411, 2003..

[12] J.A McQuillan and J.F Power. “A survey Of UMl-based Coverage Criteria

for Software Testing.", A Technical Report. National University of Ireland.

[13] G.J. Myers, C.sandler, T.Badgett, and T.M.Thomas. "The art of software Testing" , 2nd Edition.Wiley,2004

[14] A.J Offutt and A.Abdurazik. "Generating Tests from UML specifications,"In Proceedings of the 2nd International Conference on the Unified Moderating Language (UML'99), pp.416-429, October, 1999.

[15] OMG, "UML 2.0 OCL Specification,"2004.

[16] OMG, "Unified Modeling Language Specification (v2.0),"2004.

[17] OMG, "XML Metadata Interchange (XMI),v2.1",2004.

[18] A.Rountev, S.Kagan and J. Sawin, "Coverage Criteria for testing of object interactions in sequence diagrams", In Fundamental Approaches to Software Engineering,Edinburgh,Scotland,2-10 April 2005.

[19] M. Winters, "Quality Assurance for object-oriented Software- Requirements Engineering and Testing w.r.t. Requirements Specification(in German)",Ph.D thesis,University of Hagen,Germany,Sept 1999.

[20] M. Sarma, R Mall, "Automatic Test Case Generation from UML Models", 10th International Conference on Information Technology, IEEE computer society, pp. 196-201, 2007.

[21] Emanuela G, Franciso and Patricia, "Test Case Generation by means of UML sequence diagrams and Labeled Transition Systems," IEEE , pp.1292-1297, 2007.

[22] F. Fraikin, T. Leonhardt, "SeDiTeC-testing based on sequence diagrams", In Proceedings of 17th IEEE International Conference on Automated Software Engineering, pp. 261–266, 2002.

[23] Fröhlick P. and Link J., Automated Test cases Generation from Dynamic Models, In Proceedings of the European Conference on Object-Oriented Programming, Springer Verlag, LNCS 1850, pp. 472-491, 2000.

[24] Tonella, P., Potrich, A. "Reverse Engineering of the Interaction Diagrams from C++ Code", In Proceedings of IEEE International Conference on Software Maintenance, pp. 159–168,2003.

[25] Debasish Kundu, Debasis Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams", in Journal of Object Technology, Vol. 8, No. 3, pp.65 -83, May-June 2009.

[26] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, “Model-driven development using UML 2.0: promises and pitfalls”, IEEE Computer, 39(2):59-66, Feb. 2006.

[27] S. Muchnick. Advanced Compiler Design and Implementation, First Ed., Morgan Kaufmann, 1997.

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

[29] D. Ince, Object-Oriented Software Engineering with C++, McGraw-Hill, 1991.

[30] V. Garousi, L. Briand, Y. Labiche. “Control flow analysis of UML 2.0 sequence diagrams”, Technical report. Available at http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-09.pdf.

[31] Philip Samuel, Rajib Mall, “A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams”, e-Informatica: Software Engineering Journal, Volume 2, Issue 1, 2008.

[32] D. P. Mohapatra, R. Mall, R. Kumar, “Computing dynamic slices of concurrent object-oriented programs”, Information and Software Technology 47, pp. 805–817, 2005.

[33] Rountev, A., Volgin, O., Reddoch, M.: Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR 12, Ohio State University, 2004.

[34] A. Nayak and D. Samanta, “Automatic Test Data Synthesis using UML Sequence Diagrams”. Journal of Object Technology, Vol. 09, No. 2, pp. 75-104, March-April 2010.

[35] P. L. Navarro, D. S. Ruiz, and G. M. Perez, A Proposal for Automatic Testing of GUIs Based on Annotated Use CasesAdvances in Software Engineering, Vol. 2010, Article ID 671284, doi: 10.1155/2010/671284, 2010.

[36] M. Prasanna and K.R. Chandran, “Automatic Test Case Generation for UML Object diagrams using Genetic Algorithm”, Int. J. Advance. Soft Comput. Appl., Vol. 1, No. 1, pp. 19-32, July 2009.