JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008
Vol. 8, No. 3, May–June 2009

A Novel Approach to Generate Test Cases from UML Activity Diagrams

Debasish Kundu and **Debasis Samanta**
School of Information Technology
Indian Institute of Technology, Kharagpur
Kharagpur, West Bengal, India
{dkundu,dsamanta}@sit.iitkgp.ernet.in

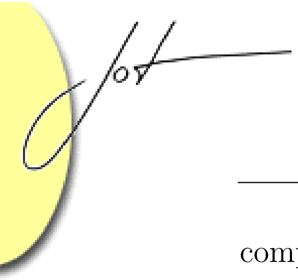
Model-based test case generation is gaining acceptance to the software practitioners. Advantages of this are the early detection of faults, reducing software development time etc. In recent times, researchers have considered different UML diagrams for generating test cases. Few work on the test case generation using activity diagrams is reported in literatures. However, the existing work consider activity diagrams in method scope and mainly follow UML $1.x$ for modeling. In this paper, we present an approach of generating test cases from activity diagrams using UML 2.0 syntax and with use case scope. We consider a test coverage criterion, called *activity path coverage* criterion. The test cases generated using our approach are capable of detecting more faults like synchronization faults, loop faults unlike the existing approaches.

1 INTRODUCTION

Model-driven software development is a new software development paradigm [5]. Its advantages are the increased productivity with support for visualizing domains like business domain, problem domain, solution domain and generation of implementation artifacts. In the model-driven software development, practitioners also use the design model for testing software- specially object-oriented programs. Three main reasons for using design model in object-oriented program testing are: (1) traditional software testing techniques consider only static view of code which is not sufficient for testing dynamic behavior of object-oriented system [2], (2) use of code to test an object-oriented system is complex and tedious task. In contrast, models help software testers to understand systems 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 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.

Activity diagram is an important diagram among 13 diagrams supported by UML 2.0 [12]. It is used for business modeling, control and object flow modeling,

Debasish Kundu, Debasis Samanta: "A Novel Approach to Generate Test Cases from UML Activity Diagrams", in *Journal of Object Technology*, vol. 8, no. 3, May–June 2009, pp. 65–83, http://www.jot.fm/issues/issue_2009_05/article1/



complex operation modeling etc. Main advantage of this model is its simplicity and ease of understanding the flow of logic of the system. However, finding test information from activity diagram is a formidable task. Reasons are attributed as follows: (a) activity diagram presents concepts at a higher abstraction level compared to other diagrams like sequence diagrams, class diagrams and hence, activity diagram contains less information compared to others, (b) presence of loop and concurrent activities in the activity diagram results in path explosion, and practically, it is not feasible to consider all execution paths for testing. To address these, few work have been reported in the literatures [10, 11]. Existing work [10, 11] consider activity diagram in method scope and use UML 1.0 syntax. But, UML 2.0 introduces major revision for activity diagrams from its preceding UML 1.x version. Therefore, there is a need for improvement of testing quality by considering activity diagrams using UML 2.0 and with higher level scope, that is, use case level.

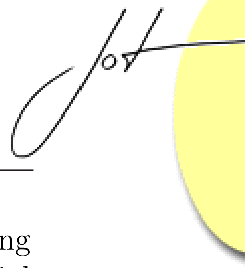
In this work, we propose an approach for generating test cases using UML 2.0 activity diagrams. In our approach, we consider a coverage criterion called *activity path* coverage criterion. Generated test suite following *activity path* coverage criterion aims to cover more faults like synchronization faults, faults in a loop than the existing work.

The rest of the paper is organized as follows. Section 2 reviews the use of activity diagrams for software testing. Section 3 presents our proposed methodology with an illustration. Comparison of our work with previous work is discussed in Section 4. Finally, Section 5 concludes the paper.

2 RELATED WORK

In this section, we review the existing work where activity diagrams are used for generating test cases.

L. C. Briand et al. [3] propose the *TOTEM* system for system level testing. In their approach, all possible invocation of use case sequences are captured in the form of an activity diagram. L. C. Briand et al. [3] consider sequence diagrams to represent use case scenarios. Further, they propose to derive various test information, test requirements, test cases, test oracles from the detailed design description embedded in UML diagrams, and expressed in Object Constraint Language (OCL). In another work, J. Hartmann et al. [8] describe an approach of system testing using UML activity diagrams. Their approach takes the textual description of a use case as input, and converts it into an activity diagram semi-automatically to capture test cases. They also add test requirements to the test cases with the help of stereotypes. Test data (set of executable test scripts) are then generated applying category partition method. In an approach on scenario-based testing, Xiaoqing BAI et al. [1] consider a hierarchy of activity diagrams where top level activity diagrams capture use case dependencies, and low level activity diagrams represent behavior of the use cases. Xiaoqing BAI et al. [1] first eliminate the hierarchy structure of the activity diagrams, and subsequently, they convert them into a flattened system



level activity diagram. Finally, it is converted into an activity graph by replacing conditional branches into its equivalent execution paths and concurrency into serial sequences. This activity graph is a graphic representation of the execution called thin-thread tree. A thin thread is basically a usage scenario in a software system from the end user's point of view. Thin threads are further processed to generate test cases.

Activity diagrams are also used for *gray-box testing* and checking consistency between code and design [10, 11]. Wang Linzhang et al. [10] propose an approach of *gray-box testing* using activity diagrams. In *gray-box testing* approach, test cases are generated from high level design models, which represent the expected structure and behavior of software under testing. Wang Linzhang et al. [10] consider an activity diagram to model an operation by representing a method of a class to an activity and a class to a swim lane. Test scenarios are generated from this activity diagram following *basic path* coverage criterion, which tells that a loop is to be executed at most once. *Basic path* coverage criterion helps to avoid path explosion in the presence of a loop. Test scenarios are further processed to derive *gray-box* test cases. Chen Mingsong et al. [11] present an idea to obtain the reduced test suite for an implementation using activity diagrams. Chen Mingsong et al. [11] consider the random generation of test cases for Java programs. Running the programs with applying the test cases, Chen Mingsong et al. [11] obtain the program execution traces. Finally, reduced test suite is obtained by comparing the *simple paths* with program execution traces. *Simple path* coverage criterion helps to avoid the path explosion due to the presence of loop and concurrency.

3 PROPOSED METHODOLOGY

In this section, we discuss our proposed approach to generate test cases from an activity diagram. Our approach consists of the following three steps.

1. Augmenting the activity diagram with necessary test information.

2. Converting the activity diagram into an activity graph.

3. Generating test cases from the activity graph.

We describe these three steps in detail in the following sub sections. We also illustrate each step with a running example of *registration cancellation* use case of *conference management system.*

Augmenting activity diagram with necessary test information

In this sub section, we describe guidelines for modeling necessary test information into an activity diagram followed by an example.
Guidelines

1. For an activity A_i that changes the state of an object OB_i from state S_a to state S_b, we show state S_a of object OB_i along with OB_i at input pin of the activity A_i [4, 12] and state S_b of the object OB_i along with OB_i at output pin of A_i.

2. For an activity A_i that creates an object OB_i during execution [4, 12], we show that object OB_i at output pin of the activity A_i.

3. We replace a loop, decision block or fork-join block in any thread originated from a fork by an activity with higher abstraction level.

Note that we are not considering the details of each activity such as what are actions encapsulated in an activity and what are the input, output parameters of each action in order to preserve simplicity of activity diagrams.

Example 1

Let us now consider an example of *registration cancellation* use case. We model the activity diagram of this use case following aforementioned guidelines, which is shown in Fig. 1.

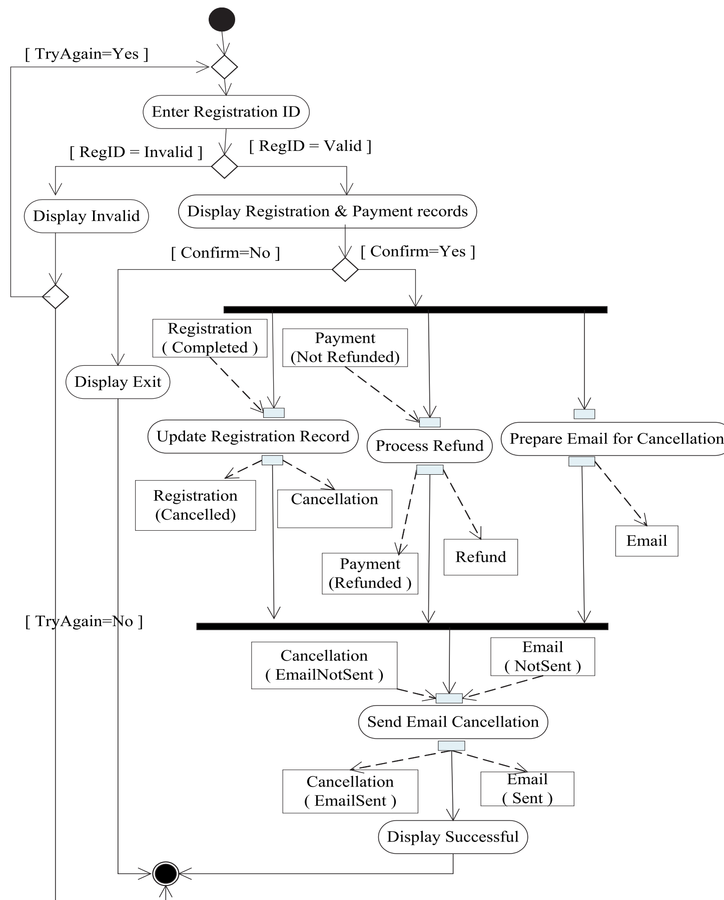
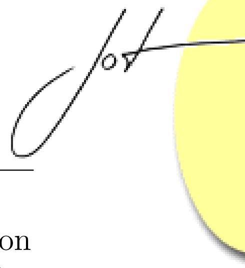


Figure 1: An activity diagram of *Registration Cancellation* use case



We see in Fig. 1 that at first *Enter Registration ID* activity asks to enter registration ID. If registration ID is valid, *Display Registration and Payment Records* activity displays detail information about registration such as workshop, tutorial, schedule etc. and payment details. If registration ID is invalid, *Display Invalid* activity shows invalid message and prompts to try again or not. If 'no' option is selected, use case ends immediately else *Enter Registration ID* activity asks again to enter registration ID. In case registration ID is found to be valid, it asks to confirm it. If not confirmed, execution of use case ends otherwise it begins concurrent execution of activities namely, *Update Registration Record*, *Process Refund*, *Prepare Email for Cancellation*. Here, *Update Registration record* activity updates *registration* object and changes its state from *complete* to *cancelled*. During execution of this activity, new *cancellation* object is also created. Another concurrent activity *Porcess Refund* involves in the refund of payment and causes to change of state of *Payment* object from *NotRefunded* to *Refunded*. New *Refund* object is also created during execution of this activity. *Prepare Email for Cancellation* activity prepares email for cancellation based on the information contained in *Registration* and *Payment* objects. As this activity does not cause any change of state of *Registration* and *Payment* object, they are not shown in input/output pin of that activity. Final result of this activity is creation of *Email* object. After execution of all three concurrent activities are over, *Send Email Cancellation* activity starts execution. It takes the input *Email* object created by *Prepare Email for Cancellation* activity and *Cancellation* object created by *Update Registration record* activity. This activity changes the state of *Email* object from *NotSent* to *Sent* and state of *Cancellation* object from *EmailNotSent* to *EmailSent*. At end, *Display Successfully* shows successful message. Note that in this example, no loop or decision or fork-join block is present in any of three threads of the activity diagram.

Converting activity diagram into activity graph

In this sub section, we discuss about the conversion procedure of an activity diagram into an activity graph.

An activity graph is a directed graph where each node in the activity graph represents a construct (initial node, flow final node, decision node, guard condition, fork node, join node, merge node etc.), and each edge of the activity graph represents the flow in the activity diagram. Note that an activity graph encapsulates constructs of an activity diagram in a systematic way suitable for further automation.

We propose a set of rules for mapping constructs of an activity diagram into nodes of an activity graph, which are shown in Table 1. We may note that there are ten different types of nodes in activity graph: S(*start node*), E(*flow final/activity final*), A(*activity*), O(*object*), OS(*object state*), M(*merge*), F(*fork*), J(*join*), D(*decision*), C(*condition*).

Example 2
Applying the mapping rules as specified in Table 1 on the running example of activity

Table 1: Mapping rules.

No	Constructs of Activity Diagram	Node of Activity Graph
1	Initial Node	Node of type S with no incoming edge
2	Activity Final Node	Node of type E with no outgoing edge
3	Flow Final Node	Node of type E with no outgoing edge
4	Decision Node	Node of type D
5	Guard Condition associated decision node	Node of type C and associated with condition string. Its parent node is of type D
6	Merge Node	Node of type M and having single outgoing edge
7	Fork Node	Node of type F with single incoming edge
8	Guard condition associated with fork node	Node of type C and its parent node is of type F
9	Join Node	Node of type J and will have one outgoing edge.
10	An object '*OB*' at input/output pin of an activity '*AC*'	Node of type O and associated object name is '*OB*'. Its parent node will be of type '*A*' and associated activity name '*AC*'. If same object '*OB*' is in both input and output pin of the activity '*AC*', then only one node is to be used.
11	Object state '*S*' of an object '*OB*'	Node of type OS. If '*OB*' is at input of an activity, then this node is left child of node of type O and associated object name is '*OB*' otherwise this node is right child of parent node associated object name with '*OB*'.
12	Activity Node	Node of type A. Its associated string is activity name.

diagram of '*Registration Cancellation*' use case, we obtain a set of nodes of the activity graph as shown in Fig. 2. To form edges, we consider one-to-one mapping from an edge of the activity diagram into an edge between two nodes in the activity graph (see Fig. 2). For easy references in our subsequent discussions, we label the

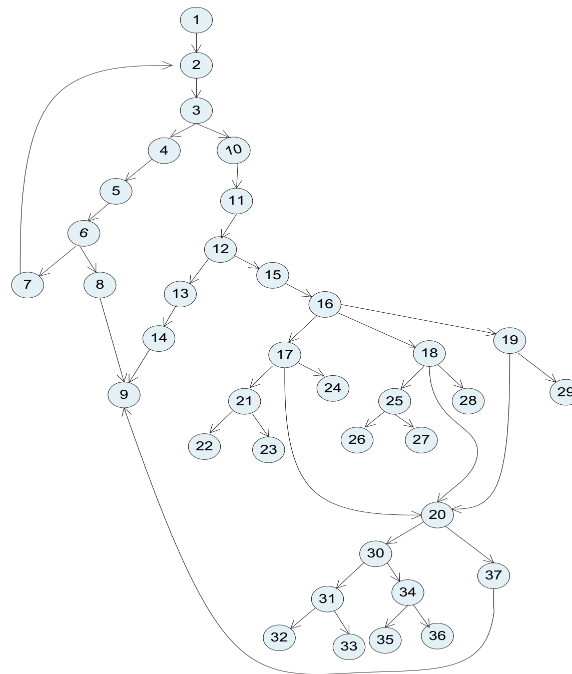


Figure 2: Activity graph obtained from activity diagram of *Registration Cancellation* use case

nodes of the activity graph as shown in Fig. 2 and store detail information of each node in the activity graph in a data structure, called *Node Description Table* (NDT) (see Table 2). Similarly, we also obtain subordinate activity graph for each high level

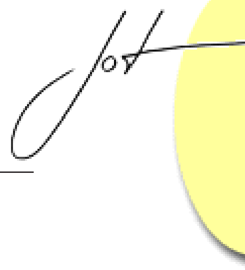


Table 2: NDT for activity graph

Node Index	Type of Node	Associated String(Activity name/ branch condition/ object name / object state)
1	S	
2	A	Enter Registration ID
3	D	
4	C	RegID=NotValid
5	A	Display Invalid
6	D	
7	C	TryAgain=Yes
8	C	TryAgain=No
9	E	
10	C	RegID=Valid
11	A	Display Registration and Payment Record
12	D	
13	C	Confirm=No
14	A	Display Exit
15	C	Confirm=Yes
16	F	
17	A	Update Registration Record
18	A	Process Refund
19	A	Prepare Email for Cancellation
20	J	
21	O	Registration
22	OS	Completed
23	OS	Cancelled
24	O	Cancellation
25	O	Payment
26	OS	Not Refunded
27	OS	Refunded
28	O	Refund
29	O	Email
30	A	Send Email Cancellation
31	O	Cancellation
32	OS	EmailNotSent
33	OS	EmailSent
34	O	Email
35	OS	NotSent
36	OS	Sent
37	A	Display Successfully

activity which replaces the decision block / fork-join block /loop block in a thread as discussed in the first sub section of Section 3.

Generating test cases from activity graph

In this sub section, we first present fault model. We then discuss the existing test coverage criteria, and our proposed test coverage criterion. Subsequently, we present our approach of generating test cases from an activity graph following the proposed test coverage criterion.
Fault model
Every test strategy targets to detect certain categories of faults called its fault model [2]. Our test case generation scheme is based on the following fault model.

- *Fault in decision*: This fault occurs in a decision of an activity diagram. For example, a fault in the decision which decides validity of a registration may display registration information and payment information of some registrant

for invalid registration ID, whereas for valid registration ID, it may display invalid account.

- *Fault in loop*: This fault occurs in either loop entry condition or loop terminating condition or increment operation or decrement operation. For an example in *Registration Cancellation* use case, if a fault exists in the terminating condition of the loop then it may happen that after giving input for the first time $TryAgain = Yes$, loop is executed for its 2nd iteration and say, at the end of 2nd iteration, after giving $TryAgain = No$, loop is not exiting rather it executes for its 3rd iteration etc.

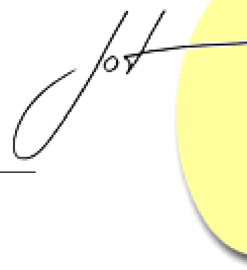
- *Synchronization fault*: This faults occurs when some activity begins execution before completion of execution of group of all preceded activities. In the *Registration Cancellation* example, if *Send Email Cancellation* activity starts execution before the completion of all concurrent activities *Update Registration Record* and *Prepare Email for Cancellation*, failure may arise as because objects *Email* and *Cancellation* which are created by preceded concurrent activities may not be available to *Send Email Cancellation.* Main reason is that activities are not executed in timely manner, that is, *Send Email Calcellation* activity is not synchronized with concurrent activities.

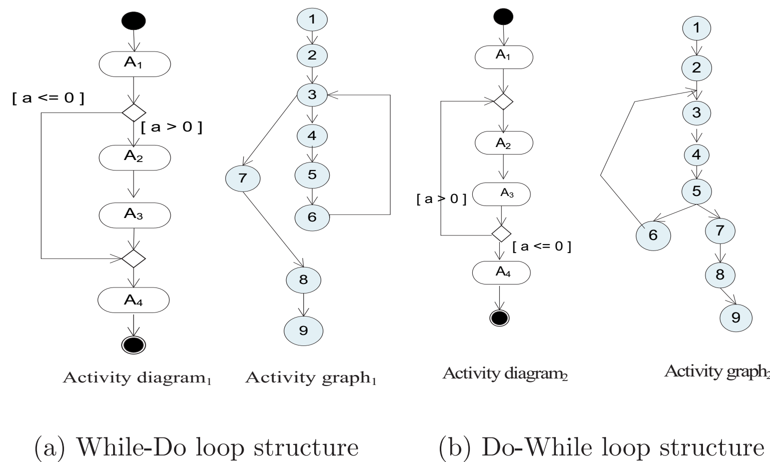
Test coverage criteria
Test coverage criteria [9] is a set of rules that guide to decide appropriate elements to be covered to make test case design adequate. We discuss the existing test case coverage criteria followed by our proposed criterion, which we consider as an improved test coverage criterion.

a. **Basic path coverage criterion**
First, we define *basic path* in activity graph. A *basic path* is a sequence of activities where an activity in that path occurs exactly once [10, 11]. Note that a *basic path* considers a loop to be executed at most once.

Given a set of *basic paths* P_B obtained from an activity graph and a set of test cases T, for each *basic path* $p_i \in P_B$, there must be at least one test case $t \in T$ such that when system is executed with the test case t, p_i is exercised.

To understand concept of *basic path* coverage criterion, let us consider an example shown in Fig. 3. In the activity graph of Fig. 3(a), we see that there are two basic paths; (a)$1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (loop is executed for zero time) and (b) $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (loop is executed once). These two basic paths cover the false and true value of loop condition. On the other hand, in the activity graph of Fig. 3(b), there is only one basic path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (loop is executed once). This basic path covers the false value of loop condition. The path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9$ (loop is executed twice) is necessary to check whether loop actually executes for





(a) While-Do loop structure (b) Do-While loop structure

Figure 3: Examples of activity diagram with loop structure

the true value of loop condition. But, it is not a basic path because in this path, activities A_2, A_3 occur more than once which violates the properties of basic path. This example reveals that with *basic path* coverage criterion, it may not be possible to detect fault associated with truth value of a loop condition.

b. **Simple path coverage criterion**

A *simple path* is considered for activity diagrams that contain concurrent activities [11]. It is a representative path from a set of basic paths where each basic path has same set of activities, and activities of each basic path satisfy identical set of partial order relations among them. Note that partial order relation between two activities A_i and A_j, denoted as $A_i < A_j$ signifies that activity A_i has occurred before activity A_j.

Given a set of *simple paths* P_S for an activity graph which contains concurrent activities and a set of test cases T, for each *simple path* $p_i \in P_S$ there must be a test case $t \in T$ such that when system is executed with a test case t, p_i is exercised.

To understand concept of *simple path* coverage criterion, we consider an activity diagram shown in Fig. 4. In the activity graph of Fig. 4, we see that there are eight partial order relations among activities in activity diagram : $A_1 < A_2$, $A_2 < A_3$, $A_2 < A_4$, $A_3 < A_5$, $A_4 < A_6$, $A_5 < A_7$, $A_6 < A_7$, $A_7 < A_8$. There are six basic paths which satisfy all these relations specified above and consist of same set of activities (see Fig. 4) given below.

$$
\begin{aligned}
P_1 &= 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \\
P_2 &= 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \\
P_3 &= 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \\
P_4 &= 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 8 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \\
P_5 &= 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12
\end{aligned}
$$

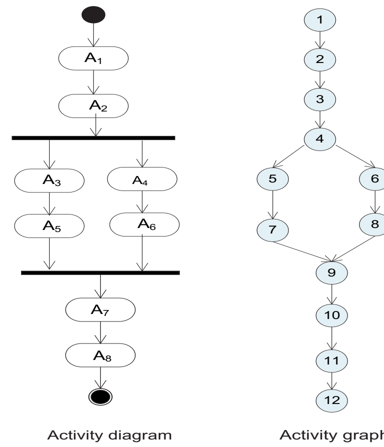


Figure 4: Example of activity diagram with concurrent activities

$$P_6 \quad = \quad 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 7 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$$

One representative path from the set of basic paths $P_1, P_2, P_3, P_4, P_5, P_6$ is considered as a simple path. It may be noted that presence of loop and decision among concurrent activities results in path explosion and hence, it is infeasible to consider all paths due to limited capability of resource and time. Simple path is one remedy in reducing number of paths to be tested. Question that still remains unanswered in the work [11] is how to select simple path from a set of basic paths so that generating redundant basic paths can be avoided in the test case generation process.

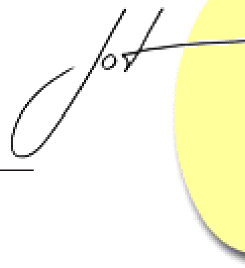
c. **Activity path coverage criterion**

We propose a test coverage criterion, called *activity path* coverage criterion. We aim to use this coverage criterion for both loop testing and concurrency among activities of activity diagrams. Before describing our new coverage criterion, we mention about *activity path* and types of *activity path*s, namely (i) *non-concurrent activity path*, and (ii)*concurrent activity path*.

First, we consider a *precedence relation* as given below.
Definition 1: A *precedence relation*, denoted as '$\prec$', over a set of activities S_A in the activity diagram is defined as follows.

1. If an activity $A_i \in S_A$ precedes a fork F and $A_j \in S_A$ is the first activity that exists in any thread originated from the fork F, then $A_i \prec A_j$.
2. If an activity $A_j \in S_A$ follows next to a join J and $A_k \in S_A$ is the last activity in any thread joining with the join J, then $A_k \prec A_j$.
3. If $A_i \in S_A$ and $A_j \in S_A$ are two consecutive concurrent activities in a thread originated from a fork F where A_i exists before A_j in the thread, then $A_i \prec A_j$.

An *activity path* is a path in an activity graph that considers a loop at most two times and maintains precedence relations between concurrent and non-concurrent



activities.

Non-concurrent activity path is a sequence of non-concurrent activities (that is, activities which are not executed in parallel) from the *start activity* to an *end activity* in an activity graph, where each activity in the sequence has at most one occurrence except those activities that exist within a loop. Each activity in a loop may have at most two occurrences in the sequence.

Concurrent activity path is a special case of *non-concurrent activity path*, which consists of both non-concurrent and concurrent activities satisfying *precedence relation* among them.

Next, we define the *activity path* coverage criterion:
Given a set of *activity paths* P_A for an activity graph and a set of test cases T, for each *activity path* $p_i \in P_A$ there must be a test case $t \in T$ such that when system is executed with a test case t, p_i is exercised.

We now discuss which type of activity path should be considered to cover what kind of faults. We use *non-concurrent activity path* to cover faults in loop and branch condition. To detect a fault in a loop, general approach [6] is to (i) execute loop zero time, (ii) execute loop once, (iii) execute loop two times, (iv) execute loop for n times, (v) execute loop for $n+1$ times, and choose a suitable value of n. Its main motivation is to test whether increment or decrement operator of the loop as well as the condition (whose value may change subject to the user input) specified in the loop entry point (*while-do loop structure*) or exit point (*do-while loop structure*) is error free or not. Here, we choose $n = 1$, that is to execute loop zero time, one time, and two times because it ensures validity of the loop condition as well as proper working of the increment/decrement operator of the loop but still avoids the path explosion. We refer it as *minimal loop testing*. Note that for *do-while* loop structure, loop condition is tested at the end of the loop. Thus, we require to test the *do-while loop structure* two times - first time for one iteration and second time for two iterations.

In our example of activity graph shown in Fig 3(a) (*while-do structure*), we see that *non concurrent activity path*s are $1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$, $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$ and, $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$ whereas *non concurrent activity path*s in the activity graph of Fig(b) for *do-while structure* are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9$ and $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9$. In both cases, non-concurrent activity paths cover both true and false value of loop condition which ensures *minimal loop testing*. Main difference between *non-concurrent activity path* and *basic path* [10] is that *basic path* only focuses on avoidance of path explosion taking the loop execution at most once but does not consider *minimal loop testing* whereas our *non-concurrent activity path* considers both.

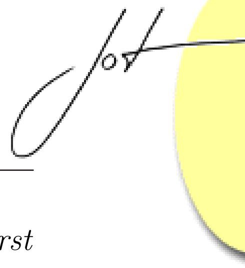
On the other hand, we consider *concurrent activity path* for an activity diagram

that contains concurrent activities. For a complex and large system, it is common to have explosion of *concurrent activity path*s because there would be large number of threads and every thread on an average would have large number of concurrent activities. Depending on runtime environmental condition, execution thread of the system would follow one *concurrent activity path*, but which *concurrent activity path* would be followed, can not be known before execution of the system. For effective testing with limited resource and time, we aim to test only relative sequence of the concurrent and non concurrent activities that is, set of *precedence relation*s exist among these activities. For this, we are to choose one representative *concurrent activity path* from a set of *concurrent activity paths* that have same set of activities and satisfy same set of *precedence relation*s. Now question is: which representative *concurrent activity path* from activity graph is to select and how? We propose to select the *concurrent activity path* such that sequence of all concurrent activities encapsulated in that path, correspond to *breadth-first search* traversal of them in the activity graph. It is so because it ensures all *precedence relation*s among the activities in the activity diagram be satisfied. Note that, we can avoid the generation of entire set of *concurrent activity paths* by finding representative *concurrent activity path* from activity graph. This will make the task of test case generation process easier and hence, reduce testing effort.

In the example shown in Fig. 4, we see that there are four concurrent activities A_3, A_4, A_5, A_6 satisfying the set of *precedence relation*s $S_p = \{A_2 \prec A_3, A_2 \prec A_4, A_3 \prec A_5, A_4 \prec A_6, A_5 \prec A_7, A_6 \prec A_7\}$. We find six *concurrent activity path*s, all of which include A_3, A_4, A_5, A_6 and satisfy S_p. Note that these six paths are same as $P_1, \ldots, P_6$ (mentioned earlier). Among them, only the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$ contains sequence of concurrent activities same as the *breadth-first search* traversal of them, so we choose it as representative *concurrent activity path*. Let us compare *concurrent activity path* with *simple path* [11] which is considered only for avoiding the path explosion due to presence of concurrent activities. Our proposed *concurrent activity path* considers both avoiding the path explosion due to presence of concurrent activities and loop testing. Another difference is that *concurrent activity path* considers *precedence relation* either between two concurrent activities or between a non-concurrent activity and a concurrent activity, but not between two non-concurrent activities. But, *simple path* considers *partial order relation* between any two kinds of activities irrespective of whether it is non-concurrent or concurrent.

Generating test cases

We generate test cases following the *activity path coverage* criterion. To do this, we obtain all *activity paths* from the node of type S to a node of type E in the activity graph. We propose an algorithm *GenerateActivityPaths* to generate all *activity paths*. In this algorithm, we use the path enumeration algorithms - *depth-first search* and *breadth-first search* traversals of graph. We traverse the activity graph by *depth-first search* except the portion of sub tree (which contains a set of nodes corresponding to all concurrent activities) rooted at a node of type F, whereas we



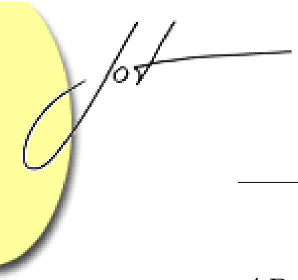
traverse that sub tree following *breadth-first search* traversal of graph. *Breadth-first search* traversal helps to avoid generating all possible *concurrent activity paths*, but to obtain only representative one. Note that these *activity paths* are not necessarily linearly independent paths [7] due to consideration of loop iteration more than once.

Algorithm1: GenerateActivityPaths

Input : An activity graph
Output: Set of *activity paths*

1 **Initialize**: LoopFlag=0; Visits of every node=0; Stack S and queue Q is used for keeping the track of visited nodes.
2 **Begin**
3 Traverse the activity graph using depth-first search (DFS). For each node visited during the traversal, count its no of visits and push the node into the stack S;
4 **if** *Visits of the current node = 2* **then**
5 Set LoopFlag=1;
6 **end**
7 **if** *LoopFlag=1 and visits of current node = 3* **then**
8 Backtrack to the node (of type 'D') which has at least a child node with its visits less than two, and then repeatedly pop from S until top of S is the node where recent backtrack has stopped;
9 **end**
10 **if** *Type of currently visited node* N_F=*'F'* **then**
11 Traverse the subtree rooted at node N_F using breadth-first search (BFS) and enqueue the nodes of type 'A' into Q until node of type 'J'/'E' is visited for out-order-degree (N_F) number of times; At end, enqueue the node of type 'J' found during recent breadth-first-search(BFS) into Q;
12 **while** *Q is Not Empty* **do**
13 Dequeue q from Q;
14 **if** *(type(q)!='J')* **then**
15 Push q into S ;
16 Explore all descendent d_q of node 'q' with taking 'q' as the root upto maximum depth=2 using depth-first-search (DFS) traversal and push d_q into S if type of d_q is other than 'A' and 'E' and 'J';
17 **end**
18 **end**
19 **end**
20 **if** *Type of currently visited node='E'* **then**
21 Copy current content of S into an array where top of S represents last node of *activity path*, sequence of elements in the array is an *activity path*; Backtrack to node of type='D' that has at least a child node whose visit count=0, and then repeatedly pop from S until top of S is the node where recent backtrack has stopped; visit its child node whose visit count = 0; If S is found to empty, stop else go to step 3 ;
22 **end**
23 **End**

Applying the algorithm *GenerateActivityPaths* on the activity graph as shown in Fig. 2, we obtain three *activity paths* as given below.



$AP_1 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9$

$AP_2 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 9.$

$AP_3 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 15 \rightarrow 16 \rightarrow 17 \rightarrow 21 \rightarrow 22 \rightarrow 23 \rightarrow 24 \rightarrow 18 \rightarrow 25 \rightarrow 26 \rightarrow 27 \rightarrow 28 \rightarrow 19 \rightarrow 29 \rightarrow 20 \rightarrow 30 \rightarrow 31 \rightarrow 32 \rightarrow 33 \rightarrow 34 \rightarrow 35 \rightarrow 36 \rightarrow 37 \rightarrow 9.$

It may be noted that algorithm *GenerateActivityPaths* generates a subset of all *activity paths*. We augment set of *activity paths* as derived following *GenerateActivityPaths* using Rule 1.

Rule 1 *: Let AP_i be an activity path in a set of activity paths obtained from an activity graph. Decompose AP_i into sequence of sub paths as $AP_i = P_1P_iP_mP_iP_n$ if possible where P_1, P_i, P_m, P_n are all sub paths of AP_i and $P_i \neq \phi$. If no such decomposition is possible, then no activity path can be derived from this AP_i, otherwise, set of activity paths derived from activity path AP_i*

$$\begin{aligned} P_{derived} &= \{P_1P_iP_n\} && \textit{if } P_m \neq \phi \\ &= \{P_1P_n, P_1P_iP_n\} && \textit{otherwise.} \end{aligned}$$

Note that $P_m = \phi$ for while-do loop structure and $P_m \neq \phi$ for do-while loop structure.

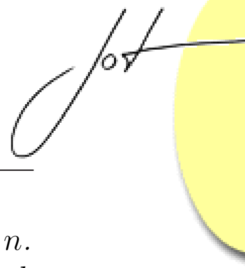
Applying this rule on each *activity path* AP_1, AP_2, AP_3, we obtain following derived *activity paths.*
$P_{derived}(from\ AP_1) = \{1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9\}$

$P_{derived}(from\ AP_2) = \{1 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 9\}$

$P_{derived}(from\ AP_3) = \{1 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 15 \rightarrow 16 \rightarrow 17 \rightarrow 21 \rightarrow 22 \rightarrow 23 \rightarrow 24 \rightarrow 18 \rightarrow 25 \rightarrow 26 \rightarrow 27 \rightarrow 28 \rightarrow 19 \rightarrow 29 \rightarrow 20 \rightarrow 30 \rightarrow 31 \rightarrow 32 \rightarrow 33 \rightarrow 34 \rightarrow 35 \rightarrow 36 \rightarrow 37 \rightarrow 9\}$

Following the above approach, we also obtain *activity paths* from subordinate activity graph which is developed for each high level activity (used to replace decision / loop / fork-join block in any thread). We then merge *activity paths* generated from main activity graph with *activity paths* generated from subordinate activity graphs using Rule 2.

Rule 2 *: Let G be a main activity graph with set of activity paths S_1 and i be a node of activity graph G corresponds to a high level activity for which there is a subordinate activity graph G_s with set of activity paths $S_2 = \{a \rightarrow b \rightarrow c \rightarrow d, a \rightarrow e \rightarrow f \rightarrow d\}$. If $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow i \rightarrow \ldots \rightarrow n \in S_1$ then replacing i in that activity path with each activity path of set S_2, obtain new merged activity paths as $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow$*



$a \rightarrow b \rightarrow c \rightarrow d \rightarrow \ldots \rightarrow n$ *and* $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow a \rightarrow e \rightarrow f \rightarrow d \rightarrow \ldots \rightarrow n$. *If cardinality of set* S_2 *is* n*, we then obtain* n *new merged activity paths for each activity path in* S_1*, which has a node* i*, corresponding to an subordinate activity, whose activity graph is* G_s*.*

For the running example, there is no such subordinate activity graph, so combination of *activity paths* is not required here. Therefore, we have total six *activity paths*, which we process for generating test cases. Test case in our approach consists of four components - *sequence of branch conditions*, *activity sequence*, *object state changes*, and *object created*. *Activity sequence*, *object state changes*, and *object created* constitute the expected system behavior. On the other hand, we consider the *sequence of branch conditions* as a source of test input. Each branch condition in *sequence of branch conditions* corresponds to the some input data specified in the textual description of the use case whose activity diagram is being considered. With the help of system analyst, we may find these input values corresponding to each branch condition.

As a part of test case generation, we obtain necessary values of all four components of a test case from the corresponding *activity path* itself. For this, we use *node description table* (NDT) constructed for the activity graph. Constituent parts of test case T_i are filled up after processing of corresponding *activity path* AP_i with help of following steps.

a. If type of the current node of AP_i is either 'S' or 'E' or 'D' or 'F' or 'J', it is ignored.

b. If type of the current node of AP_i is 'C', branch condition associated that node is the next branch condition in '*sequence of branch conditions*' part of T_i.

c. If type of the current node of AP_i is 'A', activity name associated with that node is the next activity in '*activity sequence*' part of T_i.

d. If type of the current node of AP_i is 'O' and type of the next node is 'OS', then the object name associated with the current node is the object in '*object state changes*' part of T_i.

e. If type of the current node of AP_i is 'OS' and type of the previous node is 'O' then the state associated with current node will be *old state* of the object associated with previous node.

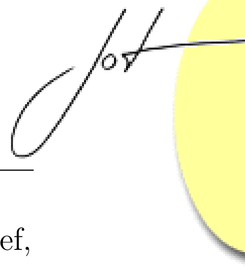
f. If type of both current node and previous node of AP_i are 'OS', then the state associated with current node will be *new changed state* of the object associated with precedent of previous node.

g. If type of current node of AP_i is 'O' and type of next node $\neq$ 'OS', then the object associated with this node will be the next in '*object created*' part of T_i.

Table 3 lists test cases which are obtained from six *activity paths*. Here, test cases 1, 4 cover fault in loop (minimal loop testing), whereas test case 2, 5 cover fault in decision. On the other hand, test case 3, 6 cover synchronization fault. We now discuss how test case 3, 6 detect synchronization fault. Let us consider an *activity path* AP_k containing two different nodes n_i and n_j which are of type O (means *object*) and associated with same object OB. Note that the object OB should be associated with two different activities A_i and A_j in AP_k. This is because, if object OB were associated with only one activity, then AP_k would not have two such nodes n_i and n_j (see also conversion of activity diagram into activity graph). Thus, it is evident that state of object OB is supposed to change as specified in the test case T_k (corresponding to AP_k) during execution of A_i and A_j. If state of OB does not change accordingly, there must be some faults in the implementation. There are two possible scenarios. In one scenario, A_i and A_j occur consecutively in the *activity path* AP_k. In that case, possible root cause for this fault may be that the activities A_i and A_j are not properly synchronized, which means synchronizing statements for these two activities have some faults. In another scenario, A_i and A_j do not occur consecutively in the *activity path* AP_k, and in that case A_i or A_j may be faulty. For example, suppose *Cancellation* object has not changed its state as expected during execution of the test case T_3. To find root cause for this fault, we find two activities *Update Registration Record* and *Send Email Cancellation* in the *activity sequence* of T_3, which are associated with the object *Cancellation* (see Table 2 and 3). Further, *Update Registration Record* is concurrent activity whereas *Send Email Cancellation* is a sequential activity and they occur consecutively. Therefore, we may infer that there are some faults exist in synchronizing statements for *Update*

Table 3: Test cases from activity graph

Test Case No	Sequence of Branch Conditions	Activity sequence	Object State changes [Object (Old state, New state)]	Object created
1	RegID = Invalid, TryAgain = Yes, RegID = Invalid, TryAgain = No	Enter Registration ID, Display InValid, Enter Registration ID, Display Invalid		
2	RegID = Invalid, TryAgain = Yes, RegID = Valid, Confirm = No	Enter Registration ID, Display InValid, Enter Registration ID, Display Registration and Payment Records, Display Exit		
3	RegID = Invalid, TryAgain = Yes, RegID = Valid, Confirm = Yes	Enter Registration ID, Display InValid, Enter Registration ID, Display Registration and Payment Records, Update Registration Record, Process Refund, Prepare Email for Cancellation, Send Email Cancellation, Display Successfully	Registration (Complete, Cancelled), Payment (NotRefunded, Refunded), Cancellation(EmailNotSent, EmailSent), Email (NotSent, Sent)	Cancellation, Refund, Email
4	RegID = Invalid, TryAgain = No	Enter Registration ID, Display InValid		
5	RegID = Valid, Confirm = No	Enter Registration ID, Display Registration and Payment Records, Display Exit		
6	RegID = Valid, Confirm = Yes	Enter Registration ID, Display Registration and Payment Records, Update Registration Record, Process Refund, Prepare Email for Cancellation, Send Email Cancellation, Display Successfully	Registration (Complete, Cancelled), Payment (NotRefunded, Refunded), Cancellation(EmailNotSent, EmailSent), Email (NotSent, Sent)	Cancellation, Refund, Email



Registration Record and *Send Email Cancellation* in the implementation. In brief, with the test case generated according to our approach we not only detect the synchronization faults but also identify possible location of the faults, which eventually reduces faults correction time and hence testing effort.

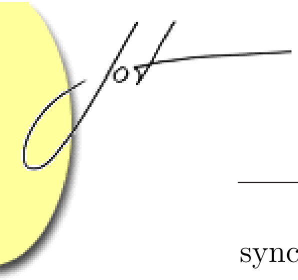
4 COMPARISON WITH PREVIOUS WORK

Our work is comparable to the work reported in [10, 11]. In the work reported [10], activity diagram is used in method scope where required input and output parameter are clearly shown in input and output action states. But in our work, we have considered activity diagrams at higher level of abstractions, that is in use case scope without capturing the details of individual activity. We have done this only to maintain the simplicity in the design. Moreover, the reported work [10] assumes that any fork node will have only two exit edges and concurrent activity states will not access the same object and only execute asynchronously as per the testing requirement. It is difficult to preserve these assumptions for real life critical applications. Our work addresses this issue. In another work [11], activity diagrams are not used for test case generation directly, but for obtaining the reduced set of test cases for an implementation whose activity diagrams are being considered. Another difference we would like to point out that both the reported work [10, 11] assume that each activity corresponds to a method of a class and each swim lane corresponds to a class and all activities in same swim lane correspond to the methods of same class, whereas an activity in our work corresponds to one or more methods of one or more classes.

Comparing our proposed *activity path* coverage criterion with the existing coverage criteria, we see that basic path [10] and simple path [11] do not ensure *minimal loop testing*, whereas *activity path* coverage criterion ensures it. To avoid path explosion, simple path is considered in [11] but how simple path (basic representative path) is selected from a set of basic paths is not discussed. Our proposed approach have addressed the problem of selection of representative path by considering the *breadth-first search* traversal of concurrent activities. None of the reported work [10, 11] has discussed how synchronization faults can be detected from the test cases generated from activity diagrams. This issue is also taken care in this work.

5 CONCLUSIONS

In this paper, we have presented an approach for generating test cases from activity diagram at use case scope. We have also proposed a test coverage criterion, called *activity path* coverage criterion. Our approach is significant due to the following reasons. First, our approach is capable to detect more faults like faults in loop,



synchronization faults than the existing approaches. Second, test case generated in our approach may help to identify location of a fault in the implementation, thus reducing testing effort. Third, our model-based test case generation approach inspires developer to improve design quality, find faults in the implementation early, and reduce software development time. Fourth, it is possible to build an automatic tool following our approach. This automatic tool will reduce cost of software development and improve quality of the software.

In the present submission, we have focused only activity diagram of a single use case at a time. However, activity diagrams of multiple use cases which are related to each other by various relationships such as, include, extend, generalization / specialization can be considered, which we plan to take up in our next work.

REFERENCES

[1] X. Bai, C. P. Lam, and H. Li. An approach to generate the thin-threads from the UML diagrams. In *28th Annual International Computer Software and Applications Conference (COMPSAC04)*, pp. 546-552, 2004.

[2] R. V. Binder. *Testing Object-Oriented Systems Models, Patterns, and Tools.* Addison Wesley, Reading, Massachusetts, October 1999.

[3] L. Briand and Y. Labiche. A UML-based approach to system testing. In *4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pp. 194-208, 2001.

[4] B. P. Douglass. *Real Time UML: Advances in The UML for Real-Time Systems.* Addison Wesley, Third Edition, February 2004.

[5] 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.

[6] J. Z. Gao, H.-S. J. Tsao, and Y. Wu. *Testing and Quality Assurance for Component-Based Software.* Artech House Publishers, 2003.

[7] Z. Guangmei, C. Rui, L. Xiaowei, and H. Congying. The automatic generation of basis set of path for path testing. In *14th Asian Test Symposium (ATS 05)*, pp. 46-51, 2005.

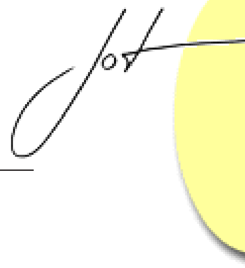
[8] J. Hartmann, M. Vieira, H. Foster, and A. Ruder. A UML-based approach to system testing. *Innovations in Systems and Software Engineering*, 1(1):12-24, April 2005.

[9] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366-427, December 1997.

[10] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating test cases from UML activity diagram based on gray-box method. In *11th Asia-Pacific Software Engineering Conference (APSEC04)*, pp. 284-291, 2004.

[11] C. Mingsong, Q. Xiaokang, and L. Xuandong. Automatic test case generation for UML activity diagrams. In *2006 international workshop on Automation of software test*, pp. 2-8, 2006.

[12] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell.* O'Reilly, June 2005.



ABOUT THE AUTHORS

Debasish Kundu received his B. Tech. in Computer Science and Technology from Kalyani University. He received MS in Information Technology from Indian Institute of Technology Kharagpur, India. Currently, he is pursuing Ph.D. in the field of Software Testing at Indian Institute of Technology Kharagpur, India. He has published 10 research papers and technical reports in peer-reviewed journals and high quality conference proceedings. He is an IEEE graduate student member. He can be reached at d.kundu.iitkgp@gmail.com. See also http://sit.iitkgp.ernet.in/∼dkundu.

Debasis Samanta received his B. Tech. in Computer Science and Engineering from Calcutta University, M. Tech. in Computer Science and Engineering from Jadavpur University, Ph.D. in Computer Science and Engineering from Indian Institute of Technology, Kharagpur. He is currently an Assistant Professor in School of Information Technology, Indian Institute of Technology, Kharagpur. He has more than 15 years of experience in teaching and published more than 50 research papers in peer-reviewed journals and high quality conference proceedings. He also has authored two books in the field of Computer Science and Engineering. He is a senior IEEE member and presently Chair of IEEE Kharagpur Section, India Council. He can be reached at dsamanta@sit.iitkgp.ernet.in. See also http://facweb.iitkgp.ernet.in/∼dsamanta.