International Journal of Computer Science and Informatics

Volume 4 | Issue 1 Article 8

July 2014

GENERATION OF TEST CASES USING ACTIVITY DIAGRAM

RANJITA KUMARI SWAIN
Rourkela Institute of Mgt. Studies, Rourkela, ranjita762001@yahoo.com

VIKAS PANTHI
Dept. of Comp. Sc. and Engg., National Institute of Technology, Rourkela, vpanthi@gmail.com

PRAFULLA KUMAR BEHERA
Dept. of Comp. Sc., Utkal University, Bhubaneswar, pbehera@hotmail.com

GENERATION OF TEST CASES USING ACTIVITY DIAGRAM

RANJITA KUMARI SWAIN[1], VIKAS PANTHI[2], PRAFULLA KUMAR BEHERA[3]

[1]Rourkela Institute of Mgt. Studies, Rourkela
[2]Dept. of Comp. Sc. and Engg., National Institute of Technology, Rourkela
[3]Dept. of Comp. Sc., Utkal University, Bhubaneswar
E-mail: ranjita762001@yahoo.com, vpanthi@gmail.com, p behera@hotmail.com

Abstract- As Unified Modeling Language (UML) activity diagrams capture the key system behavior, the UML activity diagram is well suited for the system level testing of systems. In this paper, first an activity flow graph is derived from activity diagram. Then, all the required information is extracted from the activity flow graph (AFG). The activity flow graph (AFG) for the activity diagram is created by traversing the activity diagram from beginning to end, showing choices, conditions, concurrent executions, loop statements. From the graph different control flow sequence are identified by traversing the AFG by depth first traversal technique. Next, an algorithm is proposed to generate all activity paths. Finally, test cases are generated using activity path coverage criteria. Here, a case study on Soft drink Vending Machine (SVM) has been presented to illustrate our approach.

Keywords*- Test generation technique, Test sequence gener-ation, Activity diagram, Test Case Generation, Test Coverage, Activity flow graph.*

I. INTRODUCTION

The complexity of system testing can possibly be attributed to the fact that it involves testing a fully integrated system that may be large and complex. Not surprisingly, system testing of typical systems often overwhelms manual test design efforts. Therefore, with continually increasing system sizes, the issue of automatic design of system test cases is assuming prime importance [35]. A properly generated test suite may not only locate the errors in a software system, but also help in reducing the high cost associated with software testing [24]. Unified Modeling Language (UML) is a de-facto standard for modeling analysis and design artifacts. Using UML, software designers can capture different views of a system. The different views that can be modeled using UML are: user, structural, behavioral, implementation and environmental views. An activity diagram is used to depict all possible flows of executions in a use case. Possibly, UML activity diagram is the only design artifact which is good at describing the flow of control in an object-oriented system. Due to this reason, among several UML diagrams activity diagrams are treated as one of the most important design artifact.

UML activity diagram [9] describes the sequential or concurrent control flow between activities. Activity diagram can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. UML activity diagram is a semi-formal specification of the system. As UML activity diagram captures the key system behavior, so it is well suited for the system level testing of systems [11]. What these modelling elements in the activity diagram represent are different aspects of system information, which are essential information of the system and must be preserved from design to implementation of the SUT(System Under Test) [21]. UML becomes more and more pervasively applied in the industry, but there are relatively few practical approaches and tools that support deriving test cases from models in analysis and design phases. Adequate system testing of such software requires satisfactory coverage of system states and transitions. Activity diagram is an important diagram among 13 diagrams supported by UML 2.0 [30]. It is used for business modeling, control and object flow modeling. 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. Testing activities consist of designing test cases that are sequences of inputs, executing the program with test cases, and examining the results produced by this execution. Testing can be carried out earlier in the development process so that the developer will be able to find the inconsistencies and ambiguities in the specification and hence will be able to improve the specification before the progam is written [18]. Even though UML models are intended to help reduce the complexity of a problem, with the increase in product sizes and complexities, UML models themselves become large and complex involving thousands of interactions across hundreds of objects [23]. The important part of quality control in the software life-cycle is testing. As the complexity and size of software increase, the time and effort required to do sufficient testing grow. Manual testing is time-consuming and error- prone. So, there is a pressing need to automate the testing process. The testing process can be divided into three parts: test case generation, test execution, and test evaluation. The latter two parts are relatively easy to automate provided that the criteria for passing the tests are available. However, to determine which tests are required to achieve a certain level of confidence is not trivial [25]. To generate test data form high level design notations has several advantages over code-based test case design [3]. Testing based on design models has the advantage that the test cases remain valid even when the code changes a little bit. Design models can be used as the basis for test case generation, which significantly reduces the costs of testing [34].

The process of generating test cases from design will help to discover problems early in the development process and thus it saves time and resources during development of the system. However, selection of test cases from UML models is one of the most challenging tasks [28]. In UML, the behavior of a use case can be represented by using interaction, activity and state machine diagrams. Sequence diagrams capture the exchange of messages between objects during execution of a use case. It focuses on the order in which the messages are sent. Activity diagrams, on the other hand, focus upon control flow as well as the activity-based relationships among objects. These are very useful for visualizing the way several objects collaborate to get a job done.

These are very useful for describing the procedural flow of control through many objects. Model-based testing [10] has grown in importance. Models are specified to represent the relevant, desirable features of the system under consideration (SUC). These models are used as a basis for (automatically) generating test cases to be applied to the SUC. Typical models that are used for representing system behavior are unified modeling language, finite state machines, statecharts etc.[5]. With this motivation, we aim our work at deriving test sequence from activity diagram and maximizing state or node coverage. Also our method minimizes the size of test, time and cost, while preserving test coverage. 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 etc. than the existing work.

The rest of the paper is organized as follows. A brief discussion on basic definitions and concepts used in our methodology is given in Section II. Section III presents our proposed approach for test case generation. Section IV presents a case study to demonstrate the use of our methodology with the SVM (Softdrink Vending Machine) example. In Sectin V, the related work are described and compared them with the proposed approach. Finally, Section VI presents the conclusion and future work of this paper.

II. BASIC CONCEPTS AND DEFINITIONS

Here, in this section, we introduce a few definitions and notations that are refered to in our discussions. This section briefly describes UML activity diagrams. First, we formally define several aspects of activity diagrams which will be used in the test generation. Next, we define the coverage criteria of the UML activity diagrams. Activity diagram can be used to model the dynamic behavior of a group of objects. Activity diagrams emphasize the activities of the object or a group of objects, so it is the perfect one to describe the realization of the operation in the design phase and to describe the sequence of the activities among the involving objects in the control flow during the implementation of an operation. It also describes the relationship between the activity and the object in the message flow, the state change of object in the object flow at the time of execution of activity [25]. Use cases are often supplemented with activity diagrams if the control structure of the use case includes loops or branches. The use of activity diagrams allows defining a coverage criterion to ensure a particular degree of completeness of the test scenarios. This diagram is able to reflect all possible scenarios for one use case as shown in Fig. 5.

A. UML activity diagram modelling

Here, in this section, we discuss some of the fundamentals about activity diagram. An activity diagram is similar to traditional flowcharts, which

allows us to model a process as an activity as a collection of nodes connected by edges.

can be attached to any modeling element for the purpose of modeling its behavior, including Use cases, Classes, Interfaces, Collaborations.

It is a kind of directed graph. Tokens which indicate control or data values flow along the edges from the source node to the sink nodes driven by the actions and conditions. An activity diagram has two kinds of modeling elements: Activity nodes and Activity edges.

Activity nodes

More specially, there are three kinds of nodes in activity diagrams:

– Action nodes (AN): Action nodes consume all input data/control tokens when they are ready, generate new tokens and send them to output activity edges.

– Control nodes (CN): Control nodes route tokens through the graph. The control nodes include constructs to choose between alternative flows (decision/ merge), to split or merge the flow for concurrent processing (fork / join)

– Object nodes (ON): Object nodes provide and accept data tokens, and may act as buffers, collecting data tokens as they wait to move downstream.

Activity edges

– Control flow edge: It represents flow of control through the activity

– Object flow edge: It represents the flow of objects through the activity

In our paper, we mainly consider the control and data flow of activity diagrams that are relevant to test generation. From the definition of a UML activity diagram, it may be noted that the various types of activity nodes occurring in an activity diagram include action nodes, object nodes and control nodes [1]. Among these, action nodes specify the behavior that needs to be executed. An action begins execution by taking data from its incoming edges. An action node can have multiple incoming and multiple outgoing edges. When the execution of an action is complete, data is made available to all its successor nodes. Object nodes specify the values passing through activity diagram. Object nodes are denoted by rounded rectangle symbols with the name of the node written inside.

Definition 1. An activity diagram is a six-tuple D = (A, T, F, C, a_I, a_F), where

- A = $\{a_1, a_2, \ldots, a_n\}$ is a finite set of activity states.
- T = $\{t_1, t_2, \ldots, t_n\}$ is a finite set of completion transitions.
- C = $\{c_1, c_2, \ldots, c_n\}$ is a finite set of guard conditions, and c_i is in correspondence with t_i, Cond is a mapping from t_i to c_i so that Cond(t_i) = c_i.
- F $\subseteq$ {A $\times$ T} $\bigcup$ {T $\times$ A} is the flow relation between the activities and transitions.
- $a_I \in$ A is the initial state, and $a_F \in$ A is the final state. There is only one transition t $\in$ T such that (a_I , t) $\in$ F, and for any t/ $\in$ T, (t/, a_I) $\notin$ F and (a_F , t/) $\notin$ F.

Definition 2: A test scenario, $t_s \in T_S$, in an activity diagram, AD, can be defined as an execution path from the initial state to the final state consisting of activities and transitions. i.e. $\forall$ t_s, where $t_s \in T_S$, t_s = $a_0 \rightarrow t_0 \rightarrow a_0 \rightarrow t_1 \rightarrow \ldots\ldots \rightarrow t_n \rightarrow a_m$ where
$a_i \in$ A,
$t_i \in$ T, a_0 is the initial state, a_m is the final state.
T_S is the set of test scenarios.

UML activity diagram can also be described as a directed graph, G = {*A, E, in, F*}. Here, *in* denotes the initial node such that there is a path from in to all other nodes and F denotes a set of all final nodes. A is a set of nodes consisting of {AN, ON, CN} where AN is a set of action nodes, ON is a set of object nodes and CN = {DN $\bigcup$ MN $\bigcup$ FN $\bigcup$ JN} is a set of control nodes such that DN is a set of decision nodes, MN is a set of merge nodes, FN is a set of fork nodes and JN is a set of join nodes. E denotes a set of control edges such that E = {(x, y)|x, y $\in$ A}.

Definition 3: Let AD be an activity diagram. The current state CS of AD is a subset of A. For any transition t $\in$ T,

Pre_t denotes the preset of t, then Pre_t = { a | (a, t) $\in$ F}.

$Post_t$ denotes the postset of t, then $Post_t$ = { a | (t, a) $\in$ F}.

enabled(CS) denotes the set of completion transitions that are associated with the outgoing flow edges of CS, then, enabled(CS) = {t | $Pre_t \subsetneq$ (CS) }.

firable(CS) denotes the set of transitions that can be fired from CS , then firable(CS) = {t | t $\in$ enabled(CS) }

Definition 4: Let AD be an activity diagram. For a state CS of D, a concurrent transition Γ is a set of completion transitions $t_1, t_2, \ldots t_n \in$ firable(CS) where,

1. $\forall$ i, j(1 <= i < j <= n), $Pre_{t_i} \bigcap Pre_{t_j}$ = NULL
2. $\forall$ t $\in$ (enabled(CS) - {$t_1, t_2, \ldots t_n$ }), there exists i (1 <= i <= n) such that $Pre_t \bigcap Pre_{t_i} \neq$ NULL.

An UML activity diagram can also be described as a directed graph, G = { *A, E, in, F* }. Here, *in* denotes the initial node such that there is a path from *in* to all other nodes and F denotes a set of all final nodes. A is a set of nodes consisting of {AN, ON, CN}, where, AN is a set of action nodes, ON is a set of object nodes and CN = {DN $\bigcup$ MN $\bigcup$ FN $\bigcup$ JN} is a set of control nodes such that DN is a set of decision nodes, MN is a set of merge nodes, FN is a set of fork nodes and JN is a set of join nodes. E denotes a set of control edges such that E = {(x, y)|x, y $\in$ A}.

Definition 5 Activity Flow graph (AFG): It 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.

Definition 6 Control Flow Activity Mapping Table (CFAMT): It is a table which contains the information about the mapping of each activity of an activity diagram corresponding to a node of an activity graph.

B. *Test criteria based on activity diagram*

In this sub section, we present the existing test coverage criteria. Test coverage criteria is a set of rules that guide to decide appropriate elements to be covered to make test case design adequate. The test coverage criteria proposed in the literature based on activity diagrams, are as follows:

1) All Basic Path Coverage Criterion: Here, 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. A basic path is a complete path through an activity diagram where each loop is exercised either zero or one times. This ensures that all iterations in an activity diagram are exercised.

Let us consider an example shown in Fig. 1. In the activity graph of Fig. 1(a), we see that there are two basic paths;
(a)1 → 2 → 3 → 7 → 8 → 9 (loop is executed for zero time) and
(b) 1 → 2 → 3 → 4 → 5 → 6 → 3 → 7 → 8 → 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. 1(b), there is only one basic path i.e 1 → 2 → 3 → 4 →5 → 7 → 8 → 9 (loop is executed once). This basic path covers the false value of loop condition. The path 1 → 2 → 3 → 4 → 5 → 6 → 3 → 4 → 5 → 7 → 8 → 9 (loop is executed twice) is necessary to check whether loop actually executes for 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 the faults associated with truth value of a loop condition.

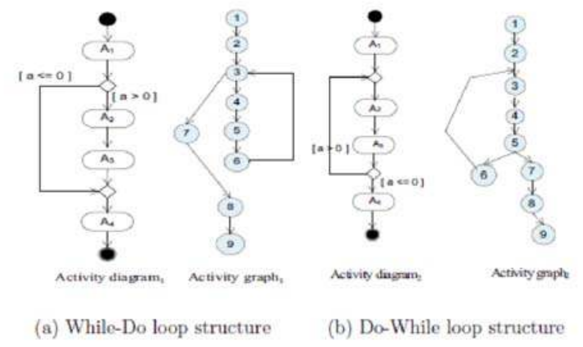

Fig. 1. activity diagram with loop structure

(2) *Simple path coverage criterion* A simple path is considered for activity diagrams that contain concurrent activities [25]. 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 testcase t ∈ T such that when the system is executed with a test case t, p_i is exercised. To understand the concept of simple path coverage criterion, we consider an activity diagram shown in Fig. 2. In the activity graph of Fig. 2, we see that there are eight partial order relations among activities in the activity diagram. They are:

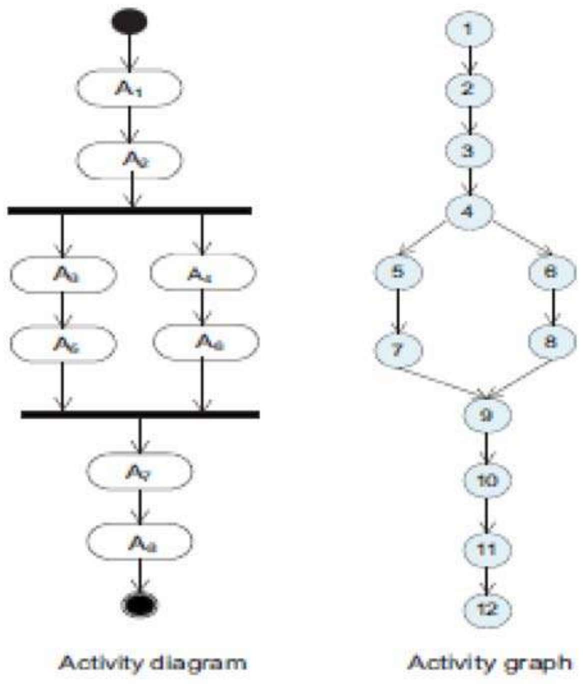

Fig. 2. activity diagram with concurrent activities

$A_1 < A_2$, $A_2 < A_3$, $A_2 < A_4$, $A_3 < A_5$, $A_4 < A_6$, $A_5 < A_7$, $A_6 < A_7$, $A_7 < A_8$. There are five basic paths which satisfy all these relations specified above and consist of same set of activities (see Fig. 2) given below.
P1 = 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12
P2 = 1 → 2 → 3 → 4 → 5 → 7 → 6 → 8 → 9 → 10 → 11 → 12
P3 = 1 → 2 → 3 → 4 → 6 → 5 → 7 → 8 → 9 → 10 → 11 → 12
P4 = 1 → 2 → 3 → 4 → 6 → 5 → 8 → 7 → 9 → 10 → 11 → 12
P5 = 1 → 2 → 3 → 4 → 6 → 8 → 5 → 7 → 9 → 10 → 11 → 12

(3) All Activity Path Coverage: Given a test set T and Activity Diagram AD, T must cause each possible activity path in AD to be taken at least once. An Activity Path is any sequence of activities from the initial activity into the terminal activity in the activity diagram. 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 ∈ T such that when the system is executed with a test case t, p_i is exercised.

(4)Transition Coverage : It requires that all transitions in the activity diagram be covered. For any test case $t \in t_s$, we can get the program execution trace (PET). If for the PET, there exists corresponding transition that is not marked in the activity diagram, we mark all the corresponding unmarked transitions for pet and record the test case *t*. The value of transition coverage is the ratio of the marked transitions to all transitions in the activity diagram.

C. Fault model
In this sub section, we present our fault model. Every test strategy targets to detect certain categories of faults called its fault model [7]. 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.
- Fault in loop: This fault occurs in either loop entry condition or loop terminating condition or increment operation or decrement operation.

III. OUR APPROACH TO GENERATE TEST CASES

In this section, we discuss our proposed approach to generate test cases from an activity diagram. We have named our appoach, Generating Test cases from Activity Diagram (Gen-TeAc). Our approach for generating test cases is schematically shown in Fig. 3. The proposed test case generation approach consists of the following steps, that are discussed below in more detail.

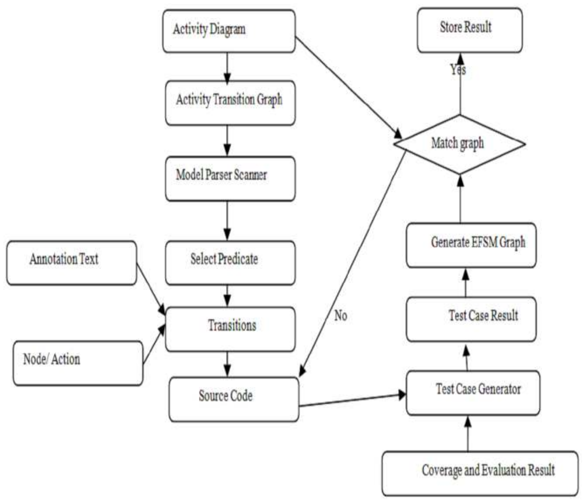

Fig. 3. Schematic diagram of our testcase geneartion process

- Constructing activity diagram for the particular usecase with necessary test information.
- Converting the activity diagram into an activity flow graph (AFG).
- Traversing the activity flow graph to extract all required information.
- Generating test cases from the activity graph.

We explain these steps in detail in the following subsections.

We also illustrate each step with a running example of Softrink Vending Machine.

A. Constructing the activity diagram with necessary test information Here, we describe guidelines for modeling necessary test information into an activity diagram followed by an example. In our technique, we employ UML models that are used to represent the requirements of a system for developing test scenarios. Each use case can be represented with one or more activity diagrams. The usecase of SVM is shown in Fig. 4.

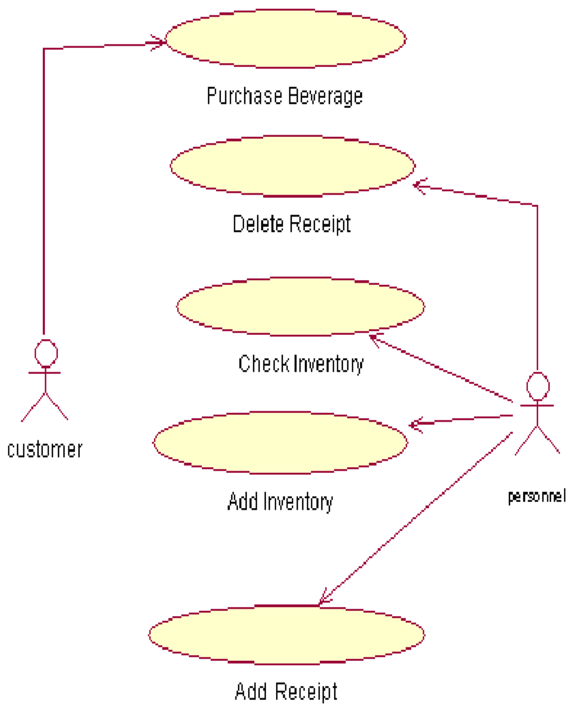

Fig. 4. Usecase diagram of Softdrink vending machine

Activity diagrams represent the scenarios related to a use case(example, Fig. 5). A scenario is a complete "path" through the activity diagram. Users of the system can traverse many paths to execute the functionality specified in the use case. The main scenario (basic path) is the one beginning from the start node, traversing through all the intermediate nodes without any error made, upto the end node. Alternate scenarios (alternate paths) are the cases when there is wrong entry of input or a condition is not satisfied. In this subsection, below, we describe guidelines for modeling necessary test information into an activity diagram.

- For an activity A_i that changes the state of an object OBJ_i from state S_a to state S_b, we show state S_a of object OBJ_i along with OBJ_i at input pin of the activity A_i [30], [13] and state S_b of the object OBJ_i along with OBJ_i at output pin of A_i.

- For an activity A_i that creates an object OBJ_i during execution [30], [13], we show that object OBJ_i at output pin of the activity A_i.

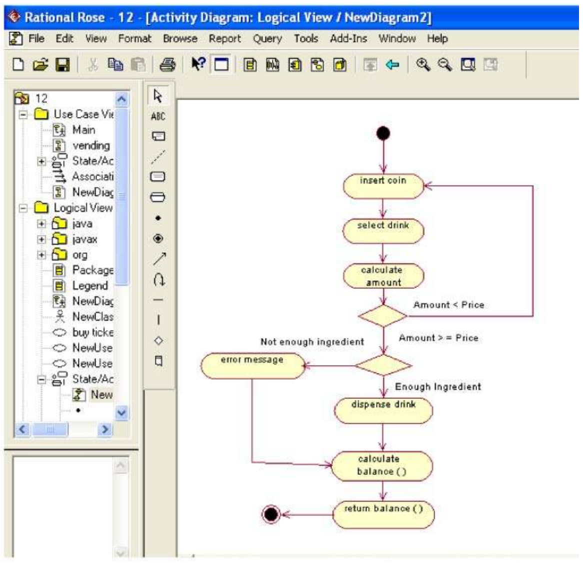

Fig. 5. A simple activity diagram of purchase bevarage usecase of SVM

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

B. Converting activity diagram into activity flow graph In this sub section, we explain about the conversion procedure of an activity diagram into an activity graph. We convert the activity diagram into a graph, called activity flow graph (AFG) using the following steps: The AFG for activity diagram is created by traversing the activity diagram from beginning to end, showing choices, conditons, concurrent executions, loop statements.

- For each conditional statement create an entry into the Control Flow Activity Mapping Table (CFAMT). Then traversing the CFAMT, create nodes in the AFG.
- The loop statements are transformed into conditional statements, listed in the CFAMT.
- For each concurrent execution statements an entry is made into the CFAMT for each execution path and in turn is represented by different execution paths in AFG.

C. Traversing the AFG to extract all required information from AFG

In this subsection we extract all necessary information's like nodes, edges, conditional statements etc. from the AFG by traversing it. An AFG may be treated as an activity transition graph, where each activity is considered as a node. From the AFG different control flow sequence are identified by traversing the AFG using depth first traversal algorithm. During traversal, we look for conditional predicates on each of the transition.

In this step there is a verification process to ensure that all required information are completed, like activity information, input, output and conditions. If all the required information are not available, then the process returns false. Again it allows re-design the diagram and filling more information. Otherwise the process will go to next step. In this process we can derive a set of test case, test data and test sequence.

D. Generating test sequences from the AFG

From the AFG, different control flow sequences are identified by traversing the AFG using depth first traversal algorithm. We generate test sequences following the activity path coverage criterion. To do this, we obtain all activity paths from the node of type *Start* to a node of type *End* in the activity graph.

1) Generating test cases: To generate test cases that satisfy the activity path criteria, we first enumerate all possible paths from the start node to a final node in the AFG of activity diagram. Each path then is visited to generate test cases. During visit, we look for conditional predicates on each of the transitions for execution of corresponding flow and activity. For each conditional predicate, based on the activity path coverage criteria and the guard condition, we generate test cases.

We propose an algorithm *GenerateActivityPaths & Testcase*, to generate all activity paths. In this algorithm, we use the path enumeration algorithms. We traverse the activity graph by depth-first search. In case of no guard condition, *NULL* is used. To generate test cases that satisfy the activity path coverage criterion, we propose this algorithm. *GenerateActivityPaths & Testcase* starts by enumerating all basic paths in the AFG, from the start node to the final node. Each basic path then is visited to generate test cases. A basic path essentially corresponds to a scenario. Now, we present below our proposed algorithm to generate test cases satisfying the coverage criterion.

Algorithm: *GenerateActivityPaths & Testcase*
Input: AFG of Activity diagram
Output: Activity paths and Test suite (T)
Steps:

```
1. TP[ ] = identifyAllBasicPaths(AFG)
//Enumerate all paths P =TP[1],TP[2] ,...,TP[n] from the start node to  a final node in the AFG.
2. For each path TP[i] ∈ TP do
3.    current node (CN) = S (start node)
4.    preC = FindPreCondof (S)
5.    $t_i = \phi$
6.    while (CN ≠ FN of path P[i]) do
7.    $Activity_{CN} = Find - activity_{CN}$
8.     If C ≠ Guard
9.       t = { preC, I ($a_1$ ,$a_2$ ,...,$a_i$), O ($d_1$ , $d_2$,...,$d_m$ ), PostC } // preC = precondition of the method or activity, I ($a_1$ ,$a_2$ ,...,$a_i$) = set of input values for the method or activity in sendObject, O ($d_1$, $d_2$,...,$d_m$ ) = set of resultant values in the receiveObject when the method or activity is executed, postC = the postcondition of the method or activity.
10.    EndIf
11.    If C = Guard then
12.       $C_{Value} = \{C_1, C_2, \ldots C_p\}$
13.         t = {preC, I ($a_1$ ,$a_2$ ,...,$a_i$), O ($d_1$ ,$d_2$,...,$d_m$ ), $C_{Value}$, postC }
14.    EndIf
15.       $t_i = t_i \bigcup t$
16.    Endwhile
17.    Determine the final output $O_i$ and postCi for the finalnode of TP[i]
18.    t = {preCi , $I_i$ ,$O_i$ , postCi}
19.    $t_i = t_i \bigcup t$
20.    $T = T \bigcup t_i$
21. Endfor
22. Return (T)
23. Stop
```

IV. CASE STUDY

In this section, we explain the working of our approach with a case study. First, we provide an overview of the problem and the activity diagram of the design model. Then, we describe the process of the test case generation from the activity diagram.

A. The problem and the model of the solution

To illustrate the test generation process, we present here the Soft Vending machine case study. In this machine an user can insert coins into it, ask the machine to vend an item or to cancel the transaction which results in the machine returning all the coins inserted and not consumed. If an item is not available or a users credit is insufficient, or a selection is invalid, the machine prints an error message and doesnt dispense the item, but instead returns any accumulated coins. Fig. 5 shows an activity diagram that represents application Vending Machine with Dispenser, and their interactions. Dispenser provides an interface that is used by Vending Machine. Vending Machine uses the services provided by Dispenser to manage credits inserted into the vending machine, validate selections, and check for availability of items. A Softdrink Vending System (SVM) dispenses softdrinks to customers. Customers use the front panel to specify their type and number of drinks i.e. details of items. The machine displays the prices for the requested item. We have mentioned 3 different categories of drinks. Once the user selects soft drink type in the menu, the object enters into DisplayForCustomer state and displays pricelist , where the prices of different types of soft drink are displayed. The user can select the type of soft drink needed, as well as the number of softdrinks (N) required. The customers then deposits cash in the bin provided and presses ’accept cash’. The machine checks the cash, if it is more, the balance cash is paid out. The Softdrink Vending Machine (SVM) system has two actors: customer and maintenance personnel. The use cases of the system are: Purchase Beverage, Check Inventory, Add Inventory, Add Recipe, Edit Recipe and Delete Recipe. Each use case was elaborated using activity diagrams. The customer can only purchase beverage. The use cases Check Inventory, Add Inventory, Add Recipe, Edit Recipe and Delete Recipe are associated with the maintenace personnel. The Softdrink Vending Machine (SVM) dispense softdrinks to the customers on receiving money from them. To explain the case, we take as example, the ‘Buy Beverage use case of the Vending Machine. The activity diagram of ‘Buy Beverage use case of the Vending Machine object for various events of interest are shown in Fig. 5. As the user enters money (a) the object changes its state to OrderController. In the OrderController state where , it calculates how much balance (ReturnMoney) is to be returned to the user if any, where ReturnMoney =Amt-TotalMoney. If the balance is less than zero, the SVM object changes its state from OrderController to ChangeDispenser, as the money inserted is insufficient. If the balance is more than or equal to zero, the object goes to SoftDrinkDispenser state and delivers the requested number of soft drink. If the balance is zero, then once the soft drink is delivered the machine changes its state from SoftDrinkDispenser to idle. If the balance is more than zero, it enters the ChangeDispenser state, where the balance money is returned. Once the money is returned, the SVM object transits to IdleMachine state. If the balance(chng) is more than zero, the machine object enters into the return money state, where the change balance money is returned. Once the money is returned, the machine object again enters into idle state.

B. Activity Diagram for SVM and CFAMT Table

After constructing activity diagram, we construct the Control Flow Activity mapping Table (CFAMT) as shown in Table I. The CFAMT (Control Flow Activity Table) is created by analyzing the activity diagram. The corresponding AFG is created by analyzing the CFAMT, as shown in Fig.. 6. For each label in the CFAT a node is created in AFG. The sequence of control flow in the CFAT is maintained in the AFG. During the AFG construction, each activity in the activity diagram is represented by a node in the AFG. The timing ordering of the diagram is maintained in the system. The conditional message in the diagram is represented by a node followed by two outward edges. Whether the condition is true or false one of the edges is covered.

Conditions for purchase beverage activity are:

- **Pre Condition:**
  1) Vending Machine does have some item available.
  2) Money entered should be a valid amount. It should be more than or equal to the calculated amount.
- **Post Condition:**
  1) Purchase successful.
  2) Purchase unsuccessful(Apology message).
- **Main Scenario :**
  1) The user invokes the purchase item by entering into the main menu.
  2) The user enters money more than or equal to the calculated amount.
  3) The user successfully receives dispensed item from the SVM.
- **Alternate course of action**

The user can not receive the item. The system notifies the apology message to the user.

I

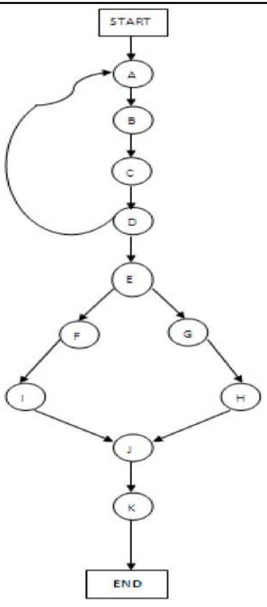

Fig. 6. Activity flow graph for the Activity diagram given in Fig. 5

C. Test Sequence Generation

From the AFG, given in Fig. 6, we identify five control flow sequence by traversing the AFG using depth first traversal algorithm. During traversal, we look for conditional predicates on each of the transitions. For each conditional predicate, we generate the test sequences by applying the *GenerateActivityPaths & Testcase* algorithm. In case of no guard condition NULL is used.

Applying the algorithm *GenerateActivityPaths & Testcase*, on the activity flow graph as shown in Fig. 6, we obtain the activity paths as given below.

1) AP_1 = A →B →C →D →A
2) AP_2 = A → B → C → D → E → F → I → J → K
3) AP_3 = A → B → C → D → E → G → H → J → k
4) AP_4 = A → B → C → D → A → B → C → D → E → F → I → J → K
5) AP_5 = A → B → C → D → A → B → C → D → E → G → H → J → k

D. Test Case generation

In this example, there is no such subordinate activity graph, so combination of activity paths is not required here. There-fore, we have total five 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.

TABLE I
CONTROL FLOW ACTIVITY MAPPING TABLE

Name of the node	Name of the activities	Predicate conditions
A	Insert coin	NULL
B	Select drink	NULL
C	Calculate amount	NULL
D	Amount < Price	C1 : Amount < Price
E	Amount >= *Price*	C2 : Amount >= Price
F	Item Available	D1 : Item Available
G	Item not Available	D2 : Item not Available
H	Error Message	NULL
I	Dispense drink	NULL
J	Calculate change	NULL
K	Return balance	NULL

Each branch condition in sequence of branch conditions corresponds to some input data specified in the textual description of the use case whose activity diagram is being considered. 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 CFAMT constructed for the activity graph.

V. RELATED WORK

Here, in this section, we discuss several existing research attempts in this area. These attempts have been reported on scenario coverage based system testing. A lot of studies have investigated the effect of test-set reduction on the size and fault finding capability of a test-set. As our work is related to test case generation using activity diagram, we survey only to those work related to UML activity diagrams. We present our survey in the followings: Earlier, Wong et al. addressed the question of the effect on fault detection of reducing the size of a test set while holding coverage constant [36], [37]. They randomly generated a large collection of test sets that achieved block and all-uses data flow coverage for each subject program. For each test set they created a minimal subset that preserved the coverage of the original set. Then, they compared the fault finding capability of the reduced test-set to that of the original set. Their data shows that test minimization keeping coverage constant results in little or no reduction in its fault detection effectiveness. This observation leads to the conclusion that test cases that do not contribute to additional coverage are likely to be ineffective in detecting additional faults. Again, to confirm the results in the Wong study, Rothermel et al. performed a similar experiment using seven sets of C programs with manually seeded faults [33]. For their experiment they used edge-coverage [14] adequate test suites containing redundant tests and compared the fault finding of the reduced sets to the full test sets. In this experiment, they found that [4] the fault-finding capability was significantly compromised when the test-sets were reduced and [6] there was little correlation between test-set size and fault finding capability. The results of the Rothermel study were also observed by Jones and Harrold in a similar experiment [17]. Hartmann et al. [16] proposed an approach for generating system level tests from activity diagrams. In their approach, activity diagrams were manually annotated prior to the test case generation. The annotations help to determine different variables and possible data choices for these variables. Test cases were then generated considering all paths in the annotated diagram. They made use of category partition method for generating test data corresponding to a test case. They also discussed test case execution for which test cases were converted into executable test scripts or test procedures. Offutt and Abdurazik [28], [29] proposed a technique for generating test cases from UML state diagrams. They have highlighted several useful test coverage criteria for UML state charts such as: (1) full predicate coverage, (2) transition coverage etc. Linzhang et al. [25] suggested an approach for generating test cases from activity diagrams and present UMLTGF, a prototype tool for supporting automation in test case generation. While traversing activity diagrams, they restricted that the loops be executed at most once and consider basic path coverage criterion for generating test cases. The approach relies on the assumption that any fork node can only have two outgoing edges and they generate two scenarios from such a fork structure. Kansomkeat and Rivepiboon [18] discussed a method for generating test sequences using UML state chart diagrams. They transformed the state chart diagram into a flattened structure of states called testing flow graph (TFG). From the TFG, they listed possible event sequences which they considered as test sequences. The testing criterion they used to guide the generation of test sequences was the coverage of the states and transitions of TFG. Kim et al. [19] proposed a method for generating test cases for class testing using UML state chart diagrams. They transformed state charts to extended FSMs (EFSMs) to derive test cases. In the resulting EFSMs, the hierarchical and concurrent structure of states were flattened and broadcast communications are eliminated. Then data flow is identified by transforming the EFSMs into flow graphs, to which conventional data flow analysis techniques were applied. Mingsong et al. [27] proposed an approach for generating test cases from activity diagrams. The approach made use of basic paths for handling loops. Based on a partial order relation that sequences the activities of an activity diagram, they define a simple path which selects a representative path for handling concurrency. These simple paths were then matched with the execution trace of the corresponding program by mapping functions to activities. In this way, they found out coverage of simple paths in an activity diagram.

In the experiment discussed in this paper, we attempt to highlight some additional issue. Our work is different in some respects. The test cases which are generated according to our approach, not only detect the synchronization faults but also identify possible location of the faults, which eventually reduces faults correction time and testing effort. We achive almost 100% activity path coverage, basic path, transition and action coverage.

VI. CONCLUSION AND FUTURE WORK

In this paper, first we derived activity transition graph (AFG), from activity diagram. All required information were extracted from the graph. Next, the graph was traversed to select the predicates. Then, test cases were generated usng activity path coverage criteria. Here, we first enumerated all possible paths from the start node to a final node in the AFG of activity diagram. Each path was then visited to generate test cases. During visit, we looked for conditional predicates on each of the transitions for execution of corresponding flow and activity. For each conditional predicate, based on the activity path coverage criteria and the guard condition, we generated test cases. In this way, this paper introduced an efficient test generation technique to optimize test coverage by minimizing time and cost.

In our opinion, there is an unacceptable loss in terms of test-suite quality. Thus, we advocate research into testcase prioritization techniques and experimental studies to determine if such techniques can more reliably lessen the burden of the testing effort by running a subset of an ordered test-suite as opposed to a reduced testsuite, without loss in fault finding capability. Further research is planned to extend the model for considering time constraints to handle more complicated applications. It might be more useful to include sequence diagrams illustrating each activity in a use case so that detailed test oracles can be derived to refine scenario generation phase further.

REFERENCES

[1] UML: UML 2.0 superstructure-final adopted specification. Object Management Group. http://www.omg.org/docs/ad/03-08-02.pdf, (2003).

[2] OMG. unified modeling language specification, version 2.0, object management group, www.omg.org, August 2005.

[3] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In In 3rd International Conference on the UML, pages 383 – 395, 2000.

[4] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models, In User Interfaces for Theorem Provers, 1998.

[5] F. Belli and A. Hollmann. Test generation and minimization with basic statecharts. ACM, SAC-08, pages 718 – 723, March 2008.

[6] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with lustre and pvs. In In Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7), 1999.

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

[8] M. R. Blaha and J. R. Rumbaugh. Object-Oriented Modeling and Design with UML. Pearson, second edition.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 2001.

[10] M. Broy. Model-based testing of reactive systems. Advanced Lectures, Springer., June 2005.

[11] M. Chen, P. Mishra, and D. Kalita. Coverage-driven automatic test generation for uml activity diagrams. In In Proceedings of the 18th ACM Great Lakes symposium on VLSI, pages 139 – 142, 2006.

[12] T. S. Chow. Testing software design modeled by finite-state machines. IEEE TSE, 4(3):178 – 187, 1978.

[13] B. P. Douglass. Real Time UML: Advances in The UML for Real-Time Systems. Addison Wesley, third edition, February. 2004.

[14] P. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In Proceedings of the symposium on Testing, analysis, and verification, 1991.

[15] D. Harel. Statecharts: A visual formulation for complex systems. Sci. Comp. Prog., 8:231 – 274, 1987.

[16] J. Hartmann, M. Viera, H. Foster, and A. Ruder. A uml based approach to system testing. Journal, Innovations in Systems and Software Engineering, pages 12–24, 2005. Springer, London.

[17] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Transactions on Software Emgineering, 29(3):195 – 209, March 2003.

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

[19] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha et al. Test cases generation from UML state diagram, Software Testing Verification and Reliability, 187 – 192, 1999.

[20] N. Kosindrecha and J. Daengdej. A test generation method based on state diagram. journal of Theoritical and Applied Information Technology, pages 28 – 44, 2005 – 2010.

[21] P. Kruchten. The Rational Unified Process -An Introduction. Addison-Wesley, 2nd edition, 2000. Reading, MA.

[22] R. Lai. A survey of communication protocol testing. Journal of Systems and Software, 62(1):21 – 46, 2002.

[23] J. T. Lallchandani and R. Mall. Integrated state-based dynamic slicing technique for UML models. In IET Software, Vol. 4, Issue 1, pages 55 – 78, 2010.

[24] H. Li and L. C. Peng. Software test data generation using ant colony optimization. In Proceedings of World Academy of Science, Engineeing and Technology, January 2005.

[25] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Ju, L. Xuandong, and Z. Guoliang. Generating test cases from UML activity diagram based on gray-box method. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC04), pages 284 – 291, 2004.

[26] R. Mall. Fundamentals of Software Engineering. Prentice Hall, 3rd edition, 2009.

[27] C. Mingsong, Q. Xiaokang, and L. Xuandong. Automatic test case generation for uml activity diagrams. In In Proceedings of the 2006 International workshop on Automation of software test, pages 2 – 8. Shanghai, China, 2006.

[28] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In Proceedings of 2nd International Conference. UML, Lecture Notes in Computer Science, pages 416 – 429, 1999.

[29] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann et al. Generating test data from state-based specifications. Software Testing Verification Reliability., 13:25 – 53, 2003.

[30] D. Pilone and N. Pitman. UML 2.0 in a Nutshell. NY. O'Reilly, USA, June 2005.

[31] M. Priestley. Practical Object-Oriented Design with UML. Tata McGraw-Hill, second edition.

[32] G. Reinelt. In the traveling salesman: Computational solutions for tsp applications. Springer Berlin / Heidelberg, 840, 1994.

[33] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In Proceedings of the International Conference on Software Maintenance, pages 34 – 43, November 1998.

[34] P. Samuel and R. Mall. Boundary value testing based on UML models. In In Proceedings of the 14th Asian Test Symposium (ATS), 2005.

[35] M. Sharma and R. Mall. Automatic generation of test specifications for coverage of system state transitions. Information and Software Technology, (51):418 – 432, 2009.

[36] W. Wong, J. Horgan, S. London, and A. Mathur. Effect of test set minimization on fault detection effectiveness. Software Practice and Experience, 28(4):347 – 369, April 1998.

[37] W. Wong, J. Horgan, A. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In Proceedings of the 21st Annual International Computer Software and Applications Conference, pages 522 – 528, August 1997.

◈◈◈