World Journal of Science and Technology 2012, 2(3):57-62
ISSN: 2231 – 2587
Available Online: www.worldjournalofscience.com

Proceedings of " *National Conference on Emerging Trends in Computer Technology (NCETCT-2012)*"
Held at R.C.Patel Institute of Technology, Shirpur, Dist. Dhule, Maharashtra,India.
April 21, 2012

Test case formation using UML activity diagram

Puneet Patel[1] and Nitin N. Patil[2]

[1]PG Student, Department of Computer, R.C.Patel Institute of Technology, Shirpur, Dist. Dhule,Maharastra,India.
[2]Head and Associate Professor, Department of Computer, R.C.Patel Institute of Technology, Shirpur, Dist. Dhule,Maharastra,India.

Abstract
The automated test cases generation is viewed as a guarantee to carry out effective and maintainable software testing. Model-based test case generation methodology becomes an obvious choice in software industries and is the focus of this work. Activity diagram 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. Here an approach for generating test cases using UML activity diagram. Also a coverage criterion called activity path coverage criterion is considered in this work. Generated test suite following activity path coverage criterion aims to cover more faults like synchronization faults, loop faults.

Keywords: UML Activity Diagram

INTRODUCTION

UML models are important source of information for test case design. UML activity diagrams describe the realization of the operation in design phase and also support description of parallel activities and synchronization aspects involved in different activities perfectly.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) 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.Software testing is an important method to assure software quality. Traditional testing method based on handwork is low efficiency and cause the increase of test cost and time. With the development of testing technology, it advances high demand for software testing automation. The automated test cases generation is viewed as a guarantee to carry out effective and maintainable software testing. Model-based test case generation methodology becomes an obvious choice in software industries and is the focus of this work. 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, different UML diagrams have been considered for generating test cases. In the past, there were several approaches for generating test cases from UML activity diagrams [1-5]. But most of them do not deal with concurrency problems and tend to propose only a conceptual idea for test generation.The method of the paper [1] generates test cases from UML activity diagrams systematically, which modifies Depth First Algorithm (DFS) for automated generation. The paper does not fully handle fork-join structures. The deficiency of [1] is that any fork node has only two exit edges; evidently, these assumptions limit the applicable scope of the proposed algorithm. Another problem in the paper is that *basic path* defined, *basic paths,* can be found by the DFS algorithm, while the detailed walkthrough of the proposed algorithm shows that some test scenarios are not generated, especially when the test scenarios are derived from the fork-join parts of the activity diagrams [4].The paper [2] defines the concept of the thin-thread tree, the condition tree, and the data-object tree, as well their relationship with the UML activity diagrams [5]. The previous works dealing with test scenario generations in activity diagrams did not consider data objects and input values. This paper proposes traversal algorithms for this [5].L. C. Briand et al. [6] propose the *TOTEM* system for system level testing.In their approach, all possible invocations of use case sequences are captured in the form of an activity diagram. L. C. Briand et al. [6] consider sequence diagrams to represent use case scenarios. Further, they propose to derive various test information, test requirements, test cases, and test oracles from the detailed design description embedded in UML diagrams, and expressed in Object Constraint Language (OCL). J. Hartmann et al. [8] describe an approach of system testing using UML activity diagrams. Their

*Corresponding Author

Puneet Patel
Department of Computer, R.C.Patel Institute of Technology, Shirpur, Dist. Dhule,Maharastra, India.

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 hierarchies 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] rst 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 [1]. Wang Linzhang et al. [1] 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. [1] 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. [7] present an idea to obtain the reduced test suite for an implementation using activity diagrams. Chen Mingsong et al. [7] consider the random generation of test cases for Java programs. Running the programs with applying the test cases, Chen Mingsong et al. [7] 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.

PROPOSED Work

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

Key points of work-

- The use of activity diagrams for software testing- Activity diagrams are particularly useful in describing workflow, and use case's flow of events.
- Augmenting the activity diagram with necessary test information- Modeling necessary test information into activity diagram
- converting the activity diagram into an activity graph- An activity graph is a directed graph where each node represents a construct and each edge represents the flow in activity diagram.
- Generating test cases from the activity graph- Approach of generating test cases from an activity graph by following the proposed test coverage criterion.
- Activity diagrams of related use cases are also considered- Activity diagram can model control as well as data flow. Relationships in use cases such as Extend, generalization are considered in test cases formation.

Use of UML Activity Diagram to generate Test Cases:

Activity diagram is an important diagram among 13 diagrams supported by UML. 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 is critical task because of the following reasons:

- (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. Here an approach is proposed for generating test cases using UML activity diagrams. In this 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, loop faults.

Control nodes: The following control nodes coordinate the flow in an activity

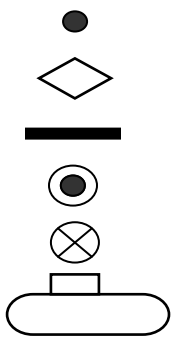

Fig 1. Control Nodes of Activity Diagram

Initial: The starting node of the activity.
Decision: A decision node has one incoming flow and multiple outgoing flows, of witch only one will be taken. Guard condition should be made to insure that only one flow can be taken.
Merge: A control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows, but to accept one among the several flows.

Fork: A control node that splits an incoming flow into multiple concurrent outgoing flows. Tokens arriving at a fork are duplicated across each outgoing flow.
Join: A join node is a control node that synchronizes multiple flows. If there is a token offered on all incoming flows, then a token is offered on the single outgoing flow.
Activity End:A node that stops all flow activity.
Flow Final: A node that terminates that flow.
Object Node: An object node represents the data participating in an activity diagram. It represents an instance of classifier as well as the corresponding data and behavior.
An object node is an abstract class, and is specialized by following object nodes:

- Pins
- Activity parameter node
- Central buffer
- Data stores

Augmenting the activity diagram with necessary test information

Guidelines for modeling necessary test information into an activity diagram are-

1. For an activity *Ai* that changes the state of an object *OBi* from state *Sa* to state *Sb*, we show state *Sa* of object *OBi* along with *OBi* at input pin of the activity *Ai* and state *Sb* of the object *OBi* along with *OBi* at output pin of *Ai*.
2. For an activity *Ai* that creates an object *OBi* during execution , we show that object *OBi* at output pin of the activity *Ai*.
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.

Example -

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

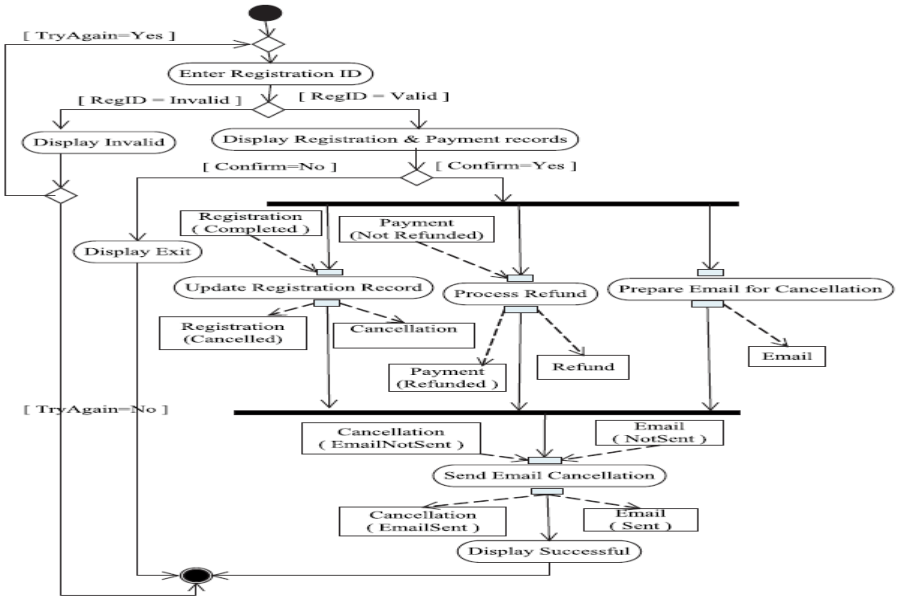

Fig 2. An activity diagram of Registration cancellation use case

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.

Converting activity diagram into 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 [10], 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*).Applying the mapping rules on activity diagram of '*Registration Cancellation*' use case, a set of nodes of the activity graph can be obtained as shown in Fig. 2. To form edges, consider one-to-one mapping from an edge of the activity diagram into an edge between two nodes in the activity graph. Detail information of each node in the activity graph can be stored in a data structure, called *Node Description Table* (NDT)[10].

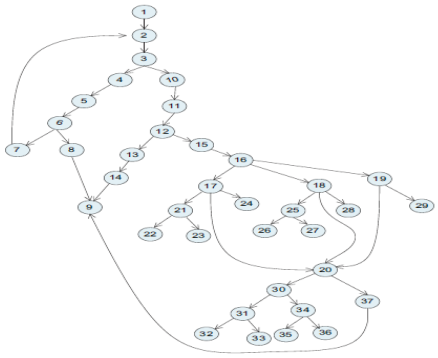

Fig 3. An activity graph obtained from an activity diagram of *Registration cancellation* use case

Generating test cases from activity graph
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 'Á', over a set of activities SA in the activity diagram is defined as follows.

1. If an activity *Ai 2 SA* precedes a fork *F* and *Aj 2 SA* is the first activity that exists in any thread originated from the fork *F*, then *Ai Á Aj* .
2. If an activity *Aj 2 SA* follows next to a join *J* and *Ak 2 SA* is the last activity in any thread joining with the join *J*, then *Ak Á Aj* .
3. If *Ai 2 SA* and *Aj 2 SA* are two consecutive concurrent activities in a thread originated from a fork *F* where *Ai* exists before *Aj* in the thread, then *Ai Á Aj* .

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.

***Activity path* coverage criterion definition:**

Given a set of *activity paths PA* for an activity graph and a set of test cases *T*, for each *activity path pi 2 PA* there must be a test case *t 2 T* such that when system is executed with a test case *t*, *pi* is exercised[10].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

1. execute loop zero time,
2. execute loop once,
3. execute loop two times,
4. execute loop for *n* times,
5. 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.

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

In our example of activity graph shown in Fig 3(a) (*while-do structure*), we see that *non concurrent activity path*s are

1 → 2 → 3 → 7 → 8 → 9, 1 → 2 → 3 →4 → 5 → 6 → 3 → 7 → 8 → 9 and, 1 → 2 → 3 → 4 → 5 → 6 → 3 → 4 →5 → 6 → 3 → 7 → 8 → 9 whereas *non concurrent activity path*s in the activity graph of Fig(b) for *do-while structure* are

1 → 2 → 3 → 4 → 5 → 7 → 8 → 9 and

1 → 2 → 3 → 4 → 5 → 6 → 3 → 4 → 5 → 7 → 8 → 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.This work considers *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 are satisfied. Note that, we can avoid the generation of entire set of *concurrent activity path*s 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 work reported, activity diagrams are considered at higher level of abstractions that is in use case scope without capturing the details of individual activity. This is only to maintain the simplicity in the design. The proposed work addresses this issue of fork considering multiple incoming flows. An activity in proposed work corresponds to one or more methods of one or more classes.The basic path and simple path do not ensure *minimal loop testing*, whereas *activity path* coverage criterion ensures it. To avoid path explosion, simple path is considered proposed approach have addressed the problem of selection of representative path by considering the *breadthFirst search* traversal of concurrent activities. Synchronization faults can be detected from the test cases generated from activity diagrams.

CONCLUSION

Activity diagrams are particularly useful in describing workflow, and use case's flow of events. This task is for modeling necessary test information into activity diagram, generating an activity graph where each node represents a construct and each edge represents the flow in activity diagram, generating test cases from an activity graph by following the proposed test coverage criterion.

The project work will go through detail study of

- UML Notations,
- Dynamic behavior of Use Cases
- Software testing basic concepts
- Code coverage testing
- Related representations

REFERENCES

[1] Wang. L., Yuan, J., Yu, X., , Hu, J., Li , X., Zheng G., "Generating Test Cases from UML Activity Diagram based on Gray-Box Method," National Natural Science Foundation of

China, 2005.

[2] Li, H., Lam, C. P., “Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams,” TestCom 2005, LNCS 3502, pp. 69 – 80, 2005.

[3] Chandler, R., Lam, C. P., Li, H., “An Automated Approach to Generating Usage Scenarios from UML Activity Diagrams,” Proceedings of the 12th Asia-Pacific Software Engineering Conference, 2005.

[4] Chen, M., Qiu, X., Li, X., “Automatic Test Case Generation for UML Activity Diagrams,” National Natural Science Foundation of China, AST’06, 2006.

[5] Xu, D., Li, H., Lam, C.P., “Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams,” Proceedings of the 12th Asia-Pacific Software Engineering Conference, 2005.

[6] 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.

[7] 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.

[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] Debasish Kundu and Debasis Samanta., “A Novel Approach to Generate Test Cases from UML Activity Diagrams.”, Journal of Object Technology Vol. 8, No. 3, May/June 2009.