International Journal Of Computer Science And Applications Vol. 1, No. 1, June 2008
ISSN 0974-1003

A Novel Approach for Test Case Generation Using Activity Diagram

Pragyan Nanda
Department of Computer Science and Engineering, National Institute of Technology, Rourkela
n.pragyan@gmail.com

Baikuntha Narayan Biswal
Department of Computer Science and Engineering, National Institute of Technology, Rourkela
baikunthanarayan@gmail.com

Durga Prassad Mohapatra
Department of Computer Science and Engineering, National Institute of Technology, Rourkela
durga@nitrkl.ac.in

ABSTRACT
Testing is an important part of quality assurance in the software development life cycle. As the complexity and size of software grow, more and more time and man power are required for testing the software. Manual testing is very much labor-intensive and error-prone. So there is a pressing need to develop the automatic testing strategy. Test case generation is the most important part of the testing efforts. Test cases can be designed based on source code but this makes test case generation difficult for testing at cluster level. Therefore, it is required to generate test cases automatically from the design documents. Also this approach holds an added advantage of obtaining test cases early in the software development life cycle (SDLC), there by making test planning more effective. Our approach first constructs the activity diagram for the given problem and then randomly generates initial test cases, for a program under testing (PUT). Then, by running the program with the generated test cases, we can get the corresponding program execution traces (PET). Next, we compare these traces with the constructed activity diagram according to the specific coverage criteria. We use a rule based frame work to generate a reduced test case set, which meets the test adequacy criteria. Advantage of our approach is that it achieves maximum path coverage.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and debugging-Testing tools.

General Terms
Reliability

Keywords
UML Activity diagram, Test case, Program under testing, program execution traces, Software testing, Rule based frame work.

1 INTRODUCTION
Testing remains, the most important part of quality assurance in the practice of software development. Quality of the end product and effective reuse of software depend to a large extent on testing. Developers therefore spend considerable amount of time and effort to achieve through testing. It is well known that software testing is a time-consuming, error-prone and costly process [3] [9]. Therefore, techniques that support the automation of software testing will result in significant cost and time savings for the software industry. Automatic generation of the test cases is essential for the automation of software testing. Once the test cases are generated automatically, a software product can even be tested fully automatically through a test execution module to realize an integrated automated test environment.

Generally test cases are designed from program source code [4]. This makes test case generation difficult especially for testing at cluster levels. Further this approach proves to be inadequate in component–based software development, where the source code may not be available with the developers. Therefore generation of test cases automatically from the *software design documents* are essential rather, than code or code-based specifications. It also holds the added advantages of allowing test cases to be available early in the *software development life cycle (SDLC)* there by making test planning more effective. Again, design oriented approach tests the generated test data which is independent of any particular implementation of the design.

In this paper, we use UML activity diagrams as design specifications and consider the automatic approach to test case generation by extending [1]. Classification of UML diagrams, depending on whether they are intended to describe the structural or behavior aspects of systems. UML activity diagrams [11, 12] describe the sequential or concurrent control flow of activities. They can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. Our approach first constructs the activity diagram for the given problem and then randomly generates the initial test cases for a program under testing (PUT) [6]. The approach is to develop a technique that will automatically generate test cases with maximal path coverage.

The rest of the paper is organized as follows. Section 2 illustrates the UML activity diagram. Various test adequacy criteria are described in section 3. Section 4 presents our approach to generate test cases from the activity diagram using a rule based frame work. Section 5 presents the Future work and conclusion.

ISSN 0974-1003

2 ACTIVITY DIAGRAM

UML provides a number of diagrams to describe particular aspects of software artifacts. These diagrams can be classified depending on whether they are intended to describe structural or behavioral aspects of systems. Activity diagrams also describe the sequence of activities among the objects involved in the control flow during implementation. Activity diagrams are similar to procedural flow charts. But the major difference between them is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities. Before presenting the detailed procedure to generate test cases using UML *activity diagram*, we need to define the *activity diagram*.

Definition. An activity diagram is a eight-tuple ACD = (A, B, F, J, K, T, C, a_0), where

- . A = $\{a_1, a_2, \ldots, a_n\}$ is a finite set of activity states.
- . B = $\{b_1, b_2, ..., b_m\}$ is a finite set of branches.
- . F = $\{f_1, f_2, \ldots, f_q\}$ a finite set of forks.
- . J = $\{j_1, j_2, ..., j_r\}$ a finite set of joins.
- . K = $\{k_1, k_2, ..., k_p\}$ a finite set of final states and end flows.
- . T = $\{t_1, t_2, ..., t_s\}$ a finite set of transitions and $t_s \in T$
- . C = $\{c_1, c_2, ..., c_v\}$ is a finite set of guard conditions.
- . a_0 is the only initial state and $a_0 \in A$

The above descriptions are shown in figure.1.

3 TEST ADEQUACY CRITERIA FOR ACTIVITY DIAGRAMS

Problem specification is the key factor to get the result accurate, which is very much important. Therefore, there is a pressing need for specification of test adequacy criteria, before going to follow the software testing procedure. The adequacy criteria of activity diagrams are based on the matching between the paths of activity diagrams and program execution traces of the implementation codes.

The description about test adequacy is given in [5, 6] as a measurement function. Suppose the *p* is a program, and *tcs* be the test cases set. The test adequacy criteria, to generate test cases for an activity diagram are given below:

- *Activity coverage:* According to this, all activity states in the activity diagram should be covered. For any $t \in tcs$, we can get the program execution trace *pet*. If there exists any function in *pet* whose corresponding activity is not *marked* in the activity diagram, we mark all the corresponding unmarked activities of *pet* and record the test case *t*. So, the value of *activity coverage* is the ratio of the marked activities to all activities in the activity diagram.
- *Transition coverage:* All transitions in the activity diagram must be covered. For any $t \in tcs$, we can get the program execution trace *pet*. If there exists any function in *pet* whose corresponding transition is not *marked* in the activity diagram, we mark all the corresponding unmarked transitions of *pet* and record the test case *t*. So, the value of *transition coverage* is the ratio of the marked transitions to all transitions in the activity diagram.
- *Path coverage:* All paths in the activity diagram must be covered. For any $t \in tcs$, we can obtain the program execution trace *pet*. If there exists any function in *pet* whose corresponding path is not *marked* in the activity diagram, we mark all the corresponding unmarked path of *pet* and record the test case *t*. So, the value of *path coverage* is the ratio of the marked paths to all paths in the activity diagram.

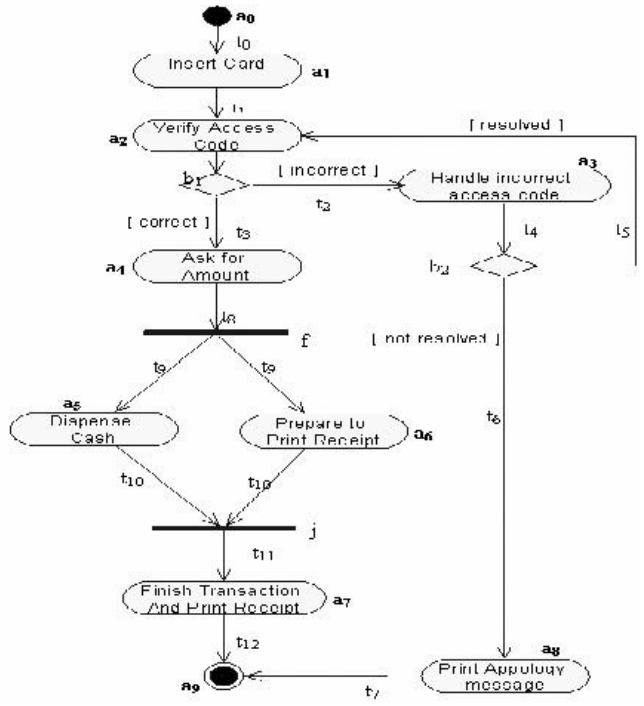

Figure.1 Activity diagram

4 HEURISTIC METHOD

In this section we discuss our work to generate test cases automatically from UML activity diagram. First, we construct the activity diagram for the given problem. Though, we are using UML activity diagram to generate the test cases, but not directly. An indirect approach is being used for automatic generation of test cases. Next, we use a randomly generated test case [8] as the initial test case is to get the *program execution traces* for a *program under testing (PUT).* Then by applying a. *"heuristic rule"* we get the best test case. At last, by comparing the execution traces with the constructed activity diagram satisfying some specification criteria, we get the reduced test cases which meet the test adequacy criteria.

4.1 Frame work

The schematic outline of the automatic test case generation strategy is described in figure.2.

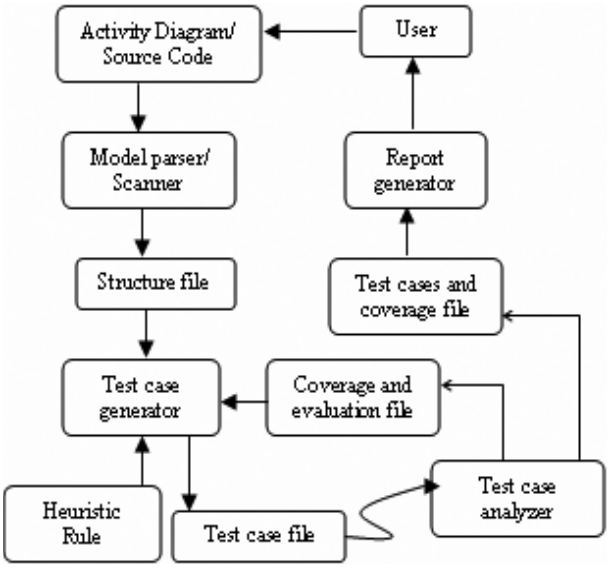

Figure 2: System model

4.1.1 MODEL PARSER/ SCANNER

The purpose of the model parser is to keep the path traversal details of the activity diagram.

4.1.2 TEST CASE GENERATOR

The test case generator test case generator produces new test cases that would cover the target branches/conditions in the code from the structure file and determines what conditions/branches should be targeted for new case generation

4.1.3 TEST CASE ANALYZER

Test case analyzer evaluates by running each test case in the program and maintains a track of condition and branch coverage. If the test case satisfies the coverage criteria it generates a report otherwise the analysis result is used by test case generator for further test case generation.

4.1.4 REPORT GENERATOR:

The report generator prints the result which includes the generated test cases, condition and branch coverage and percentage of path coverage.

4.2 Paths in the Activity diagram

The selection of path coverage in test case generation is a very complex task. When a path in the activity diagram is matched, we delete this path from the path coverage set. Hence the matching process for activity diagram will terminate when the path coverage set is empty. The algorithm for simple path searching is given in [13]. The complexity in path selection is due to the presence of synchronization, concurrency and loops. Our approach only considers the paths for selecting the program execution traces, which satisfies the semantics of the synchronization such as the *join* and *fork* in the activity diagram. Loops in an activity diagram may result in a path with infinite activities. From figure 1, we derived the following paths:

$$start <a_0>, \quad <a_1>,$$
$$< a_1 > \quad < a_2 >,$$
$$< a_2 > \quad <a_3 >,$$
$$< a_2 > \quad <a_4 >,$$
$$< a_3> \quad <a_8 >,$$
$$<a_8 > \quad <a_9 >,$$
$$<a_4 > \quad <a_5, a_6 >,$$
$$< a_5, a_6 > \quad <a_7 >,$$
$$<a_7 > \quad <a_9 > end$$

We have considered simple path to avoid the complexity due to loops and concurrency, which is beyond the scope of the discussion.

4.3 Test case generation strategy

We use the *heuristic rule* to achieve the maximal branch coverage. A branch coverage analysis is required to get the *best test case* (BCASE). The path coverage analysis follows the path prefix strategy of Prather and Myers [7]. When a path is found, we should delete this path from the path coverage set. So the matching process is getting stopped, when the path coverage set is empty. The branch coverage status of the code is recorded in a coverage table. When a branch is covered by any test case, the corresponding entry in the table is marked with a "√". The target of the test case generation is to mark all entries in the table. Therefore, the partially covered transitions are the main targets for modification, to cover all paths. The uncovered conditions will not be targeted for new test case generation. Earlier test cases can be used as models for new cases, because, no test case model yet exists that can be used for modification.

The main problem arises to select a model test case when, more than one test case drives the same path. So it is very essential to identify the *goodness* of a test case. We define the *goodness* of a test case as

$$\frac{1}{n}\left\{\left|LHS(t_1) - RHS(t_1)\right| \div \left(2 * \max\left(\left|LHS(t_1)\right|, \left|RHS(t_1)\right|\right)\right)\right\} \quad (1)$$

where, t_1 is a test case, LHS (t_1) and RHS (t_1) represent the evaluated value of LHS and RHS, respectively, when t_1 is used as the input data and n is the number of branches covered. Here, we have considered a typical format of an IF-THEN statement where the *expression* (exp) can be expressed in the form of: LHS<op>RHS. The *goodness* of a test case, t_1, relative to a given condition can be calculated using the above formula. This measures the closeness between LHS and RHS [2]. When this measure is small, it is generally true that a slight modification of t_1 may change the truth value of *exp*, thus covering the other branch. The measurement of (1) provides the *goodness* of a test case which ranges from 0 to 1. A test case that yields the smallest measurement is considered to be the best test case of the condition under consideration. In the following, we present our algorithm to get the reduced test case, which is given in figure.3.

```
begin
BCASE = F;     //A Boolean variable.
Supply AD & RTC to TCG as an I/P; /* RTC is randomly generated test case, AD is activity diagram, TCG is test case generator. */
Execute PUT with RTC to give PET; /* PET is program execution traces. */
Apply heuristic rule to TCG to generate best test case;
While (Path ≠ empty  && BCASE ≠T) {
Run TCG};
end;
```

Figure.3: The Algorithm for Reduced Test Case Generation.

5 CONCLUSION AND FUTURE WORK

We have proposed an approach to generate the test cases for *object oriented programs* from the UML activity diagrams. We have used a heuristic rule to obtain the reduced test cases, which satisfy the test case adequacy criteria. In this paper we have considered only the path (simple) for automatic test case generation. Our approach achieves the maximum path coverage, which is an added advantage. Currently, we are working on developing test cases involving nested fork -joins and branch nested fork-joins. Also it sees very similarity for Model Driven Architecture (MDA) [10], which is our next prospective. To the best of our knowledge no other paper has discussed the use of heuristic rule for generating test cases from activity diagram.

6 ACKNOWLEDGMENTS

We would like to acknowledge the anonymous reviewers for their constructive comments that helped us to improve the quality of this paper.

7 REFERENCES

[1] A. Abdurazik, J. Offutt. 2000. Using UML collaboration diagrams for static checking and test generation, in: proceedings of the third International Conference on the UML, Lecture Notes in Computer Science, Springer-Verlag GmbH, York, UK, vol.93, 2000, pp. 383-395.

[2] A. Kleppe, J. Warmer, and W. Bast. 2003. MDA Explained: The Model Driven Architecture–practice and promise. Addison-Wesley, 2003

[3] C. Mingsong, Q. Xiaokang, and L. Xuandong. 2006. Automatic Test Case Generation for UML Activity Diagrams. AST'06, May 23, 2006, Shanghai, China

[4] C. Oriat. Jartege. 2005. A Tool for Random Generation of Unit Tests for Java Classes. In QoSA/SOQUA, pages 242–256, 2005.

[5] F. Basanieri, A. Bertolino, and E. Marchetti. 2001. CoWTeSt: A Cost Weighted Test Strategy, Proc. Escom-Scope 2001, London, 2001

[6] Grade Booch, James Rambaugh, Ivar Jacobson. 2001. The Unified Modeling Language User Guide, Addison-Wesley, 2001.

[7] H. Zhu, P. Hall, and J. May. 1997. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366–427, December 1997

[8] H. Zhu and X. He. A Methodology of Testing High-level Petri Nets.2002. Information and Software Technology, 44(8):473–489, 2002.

[9] K. H. Chang, W. Homer Carlisle, James H. Cross II, and David B. Brown. 1991 A heuristic approach for test case generation. ACM.

[10] L. Briand and Y. Labiche. 2002. A UML based approach to system testing. Software and Systems modeling, 1 (1), 2002.

[11] Object Management Group, UML specification 1.5, available at http://www.omg.org/uml, 2003.

[12] R.E. Prather and P. Myers, Jr. 1987. The Path Prefix Software Testing Strategy, IEEE Trans. on Software Engineering, Vol. SE-13, No. 7, July 1987

[13] W. H. Deason, D. B. Brown, K.-H.Chang, and J. H. Cross II. 1991. A Rule-based Software Test Data Generator, IEEE Trans. on Knowledge and Data Engineering, March 1991.