International Journal of Information Technology and Knowledge Management
July-December 2012, Volume 5, No. 2, pp. 239-243

A COMPONENT-BASED APPROACH FOR TEST CASE GENERATION

Shaveta Gupta[1] and Jimmy Singla[2]

ABSTRACT: Component based software engineering is a process that aims to design and construct software systems using reusable software components. Since the system is to be build with the ready to use components or prefabricated components the testing of these components is of utmost importance. Software testing is an important verification activity in Software Development Life Cycle which requires resources and man hours. Software testing is the process used to measure the quality of software. It consist of the dynamic verification of the behavior of a program on a finite set of the test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior. The testing effort can be divided into test case generation, test execution, test evaluation. To reduce the cost of testing process, efficient test case generation technique is required. So test case generation is an important part of software testing. As manual testing is error prone and time consuming activity therefore as a part of our research work we have tried to automate manual test generation activity. We have adopted two ways by which automated test data is generated, namely test case generation from UML activity diagram based on gray box method and automated test case generation using PETA Tool. These methods have been implemented and their results have been verified.

Keywords: platform, test-case, transition, configuration

1. INTRODUCTION

As there is an ever-growing need for techniques that could improve the software development process and to reduce the time to market and improve the quality of delivered software products, a more organized and focus approach to reuse called component based software engineering has emerged. In the traditional software engineering process, the typical activities involved in building a system are specification, implementation, testing and maintenance. Component-Based Software Engineering (CBSE) not only focuses on system specification and development, but also requires additional consideration for overall system context, individual components properties and component acquisition and integration process [12]. Components are the fundamental in the building of component based software system. Brown and WallnaTu describe a component as *"a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture [1]"*. Component Based Software development as the process for building a system which consists of the searching and identifying components based on preliminary assessment; Selecting components based on stakeholder requirements; integrating and assembling the selected components; and updating the system as components evolve over time with newer versions. With the help of component based software development we have following benefits:

1. Development time is reduced .As less time is required to buy a component than to design, code, test, debug and document it.
2. Flexibility is increased, as there are more choices of components from which to choose a component that meet the requirements.
3. Process risk is reduced. If a component exists, there is less uncertainty in cost associated with its reuse as compared with new development.
4. Quality is enhanced. As design and implementation faults are discovered and eliminated in the initial use, thus increasing the quality.
5. Low maintenance is there. Easy replacement of obsolete components with new enhanced ones.

Software organizations spend considerable portion of their budget in testing related activities. The customer before acceptance will validate a well-tested software system. The effectiveness of this verification and validation process depends upon the number of errors found and rectified before releasing the system. This in turn depends upon the quality of test cases generated. IEEE Standard 610 (1990) defines test case as *"A set of test inputs, execution conditions, and expected results developed for a Particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [3]"*. Test generation techniques can be white-box or black box. White-box testing techniques are used for unit testing and for the cases where the size of the code is small. For larger and complex codes white-box testing is not inappropriate, and black box testing techniques can be more useful. Black box

[1] Department of Computer Science, PGGC, Chandigarh, India, *E-mail: shaveta.83@rediffmail.com*
[2] Department of Computer Science and Engineering, NWIET, Dhudike, India, *E-mail: jimmysingla285@gmail.com*

testing usually requires generating test cases on the basis of a model which can represents the system, called the test model. Model based representation enhances the understanding of the system. It helps in making the test-generation process faster. The Unified Modeling Language is a very good tool for modeling of large and complex systems. It is widely used to improve the testing. As we know that manual testing is error prone and time consuming so we automate the test case generation process. Test automation allows for a continuous control of your software's quality. The common approach in automated testing primarily concentrates on the area of functional end-to-end tests (GUI testing) and the testing of single software modules. Modern software architectures like web services or service oriented architectures (SOA) connect these components by providing communication via message based protocols. PETA is an Eclipse-based platform for automated software quality assurance. PETA focuses on communication between software components on the message protocol layer, making it applicable in almost any environment. Quick results are achievable for quality engineers through the UML-oriented graphical editor even without programming skills. The PETA-Platform is therefore a highly productive and flexible solution that is likewise covering both functional and non-functional testing processes. With the help of automate test case generation:

- Test process is accelerated.
- Test coverage is increased.
- Unattended execution of large test sets
- Relief from tedious, routinely test activities
- Continuous quality control

Component-Based software Test Case generation: - Software testing is an important and integral part of the software development process. It is used to reveal bugs in a system, to assure that the system complies with its specification and to verify that the system behaves in the intended way. According to Bertoline, *"Software testing consist of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the specified expected behavior"*. Testing should systematically uncover different classes of errors in a minimum amount of time and with a minimum amount of effort. A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can also provide an indication of the software's reliability and quality. But, testing cannot show the absence of defect -- it can only show that software defects are present. Under testing we have test requirements these are the specific things that must be satisfied or covered during testing, for example, reaching statements are the requirements for statement coverage. Test specifications these are specific descriptions of test cases, often associated with test requirements or criteria, for example, for statement coverage, test specifications are the conditions necessary to reach a statement [7]. Testing Criterion it is a rule or collections of rules that impose test requirements on a set of test cases. Testing Technique it guides the tester through the testing process by including a testing criterion and a process for creating test case values. Under Software testing test cases are generated. According to Ron Patton (2001, p. 65). "Test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software [3]". A test or a test case is a general software artifact that includes test case input values, expected outputs for the test case, and any inputs that are necessary to put the software system into the state that is appropriate for the test input values. A test specification language is a language that can be used to describe all components of a test case.

Components for a test case are:

1. Test case values: It is essential part of a test case, which comes from the test requirements. It may be a command, user inputs, software functions or values for its parameters.
2. A Test Case Prefix Value: It includes all the inputs necessary to place the software system into appropriate state for running test case values.
3. Verify Values: Any inputs that is necessary to show the results.
4. Exit Commands: That terminates the execution of the software.
5. Expected Outputs: These are the outputs of the test case on correct version of the software.

2. TEST CASE GENERATION METHODOLOGIES

2.1 Real-Time Process Algebra Based Test Case Generation

A real-time system is a system where the behavior of the system depends not only on the input but also on the timing of the input. Such system can also have requirements on the timing of its outputs. In order to test a real-time system, we have to take into account not only what inputs to supply to the system, but also when to supply them. For correct behavior of a real-time system, a response should not only provide correct values, but the values should also be provided at the right time-points. e.g. a computer that distinguishes between single and double-clicks from an input device must measure the time between two consecutive clicks. First after waiting the maximum time bound for double-click; the computer can determine a first click as a single-click. If a second click arrives earlier, then the two

clicks should be interpreted as a double-click. Based on RTPA, a method of least completed set of tests (LCST) is developed, which reveals that the sufficient number of tests for given software is O (4^n) where n is the number of the input variables [14]. The Least Completed Set of Tests is defined as a set of sufficient boundary test cases for a given program, if and only if they are sufficient and efficient for the testing of the given program, i.e.:

$$LCST = X\ Ii$$
$$I = 0$$

2.2 UML Based Test Case Generation

The Unified Modeling Language (UML) is an industry standard modeling language with a rich graphical notation, and comprehensive set of diagrams and elements. It is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

The primary goals in the design of the UML

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, Patterns and components.
7. Integrate best practices.

UML diagrams represent three different views of a system model

- Functional requirements view: This view emphasizes the functional requirements of the system from the user's point of view. It includes use case diagrams.
- Static structural view: This view emphasizes the static structure of the system using objects, attributes, operations, and relationships. It includes class diagrams and composite structure diagrams.
- Dynamic behavior view: This view emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. It includes sequence diagrams, activity diagrams and state machine diagrams.

UML Diagrams for test case generation

- Use case diagrams describe the required functionality of a system in interaction with its environment (external actors) and define its border. These are the Behavior diagrams emphasize what must happen in the system being modeled. These diagrams define the required functionality from an external point of view and thus provide information relevant for black box testing. Use cases are there to capture user requirements.
- State diagrams are state charts, which describe the behavior of a class or an object. We have o focus on finding state diagram that depict behavior of all components using single diagram. Unit testing is performed under this. State diagrams are the basis for automated model checking.
- Class diagrams are used to capture the static structure of objects. It emphasizes what things must be in the system being modeled.
- Collaboration and sequence diagrams are used to capture dynamic interactions between objects and system. These are the interaction diagrams, a subset of behavior diagrams that emphasize the flow of control and data among the things in the system being modeled: A collaboration diagram consists of objects and associations that describe how the objects communicate. An interaction occurs when two or more objects are used together to accomplish one complete task.

UML-Based Test Case Generation Techniques

- UML-Based Statistical Test Case Generation: This method introduces an approach for generating system-level test cases based on use

case models that are refined by state diagrams. And state diagrams are transformed into usage graphs and then to usage models from which test cases are generated. This method supports iterative development processes by reducing testing efforts [10].

- Using Adaptive Agent to automatically generate test scenarios from the UML Activity Diagrams: This is an automated approach using adaptive agents to directly generate test scenarios from UML activity diagrams. An adaptive agent can effectively explore the UML activity diagrams and automatically generate test scenarios. With the help of this approach UML activity diagrams exported by UML tools are directly used to generate test scenarios and the whole generation process is fully automated, and the redundant exploration of the activity diagrams is highly avoidable due to use of adaptive agents, so resulting in improved efficiency in the generation of the test scenarios [5].
- Generating Test Cases from UML Activity Diagram based on Gray-Box Method: Black box testing method generates tests from system specifications in the user's viewpoint, it only validates whether the functions specified in the system requirement Specifications were implemented or not. It needs no information about how the system was implemented, and does not take into account the developing method and programming language adopted. On the contrary, white box testing method, which is based on the programmer's viewpoint, creates program flow charts from the code implementation by reverse engineering using software comprehension and analysis techniques. Gray box method combines the white box method and the black box method. It extends the logical coverage criteria of white box method and finds all the possible paths from the design model which describes the expected behavior of an operation. Then it generates test cases, which can satisfy the path conditions by black box method. It can find problems, which used to be ignored by both black and white method [13].
- Test Cases Generation from UML Activity Diagrams: UML activity diagram is a notation suitable for modeling a concurrent system in which multiple objects interact with each other. This method generates test cases from UML activity diagrams that minimize the number of test cases generated while deriving all practically useful test cases. Our method first builds an I/O explicit Activity Diagram from an ordinary UML activity diagram Input/output explicit Activity Diagram (IOAD), is an activity diagram that explicitly shows external inputs to and external outputs. It is Explicit in showing the external inputs and outputs in the sense that in IOAD no activity can be further decomposed into constituent activities or tasks, thereby exposing all possible external inputs and outputs. And then transforms it to a directed graph, from which test cases for the initial activity diagram are derived. Based on all-paths test coverage criterion, we will traverse all nodes without revisiting the same node [6].

Importance of taking UML to generate test cases

1. It generates test cases with high efficiency and quality.
2. It gives more detailed information for requirement engineering.
3. It provides encouragement for developers towards more in depth modeling and precise maintenance of models.
4. It supports iterative development processes by reducing testing efforts.

3. OBJECTIVES

Objectives include:

1. To identify the problem associated with manual test data generation
2. Automate the manual test case generation process as much as we can.

4. AUTOMATED GENERATION OF TEST CASES

4.1 Automatic Generation of Test Cases Using PETA Tool

PETA is Java/eclipse based platform for automated software testing. Under this various components of component based software systems like client/server and Service-Oriented-Architectures (SOA) may be simulated as well as tested in isolation and in their supposed interaction. For the generation of test cases :

(*a*) Firstly we should have to apply settings under this tools.

(*b*) Class diagrams age generated.

(*c*) Finally test cases are generated.

STEPS:

1. Apply settings

1.1. Creating the project. All information needed to define the tests and their test Execution organized in a PETA project that contains the different PETA documents. These documents are test case files, test suite files, template files, At least one configuration file.

1.2. Setting up Configuration (Client & Server) While the setups are only needed the test case execution, the definition of the actors is required during test development. Therefore, before the creation of test cases, a PETA Configuration has to be created first.

1.2.1 Create the Actors

1.2.2 Create the setup

1.2.3 Activating Configuration

1.3 Setting up Test Cases. It describes the (expected) message flow between the actors and the data of the exchanged messages.

1.3.1 Select Actors

1.3.2 Add Events to the Actors Events are used to define the actions that an actor performs during the test scenario. An event is triggered by the reception of a message from another actor or by the ending of a previous event of the same actor without sending an own message.

1.3.3 Add operations to the Events. operation is used to define the sending of a message to another actor

1.3.4 Adding assertions Assertions are used to define expected results, for example to validate the data of received messages.

2. Class diagram corresponding to settings.

3. Launch test cases.

4.2 Automatic Generation of Test Cases Using Activity Diagram

Activity Diagram emphasizes on the activities of the object, so it is perfect one to describe the realization of operation in design phase. Using Activity Diagram Test Cases are generated using following approach:

(*a*) Draw the Activity diagram representing the component.

(*b*) Input file is created corresponding the Activity Diagram which is in the form [initial state, transition, next state]

(*c*) Now store the Input File into appropriate data structure like we use Hash Map.

(*d*) Now create the final state transition table, add new states to handle fork join.

(*e*) Finally test scenarios are generated

STEPS:

STEP A: Store the UML Activity diagram into appropriate data structure which can represent fork – join relationship.

1. Draw the Activity Diagram corresponding to component.

2. Represent UML activity diagram as state table and write it into in to some file.

   [Each row is represented as initial_state, transition_edge , nextState]

3. Read the input file created in Step 2 and store the state relationship information into appropriate data structure. We have used multilevel Hash Map (stateTable) to represent state relationships.

   Hash Map is the data structure to store key value pairs. We are using multilevel Hash Map, where key to main Hash Map is current state and value is inner Hash Map having all the {transition_edge, nextState} pairs. That is inner Hash Map will contain all the outbound edges (as key) and destination states corresponding to each edge(as value).

   Eg. If there two transitions starting from state a.

   a, t1, b and

   a, t2, c

   This relation will be represented as [{a, [{t1,b}, {t2,c}]}

   …… ……]

[Note: In case of fork where transition *t* from current state a reaches multiple states say *p* and *q*, then nextState will be represented as *a*, *t*, *p*:*q*, Similarly join will have single edge connecting states say *x* and *y* to next state *z* will be represented as *x*:*y*, *t*, *z*

STEP B: Add new states to handle fork - join and create final state transition table

- Initialize Multilevel Hash Map stateTransition Table to store the final transition table with expanded states to represent fork - join.
- Create queue toBeProcessedStatesQueue , this is to store transient states while the program run. Insert initial state a_i into it.

- Create a Hash Map of visitedStates, to keep track of the states which are already processed.
- while (toBeProcessedStatesQueue.isEmpty() != true)
  1. currentState := get front element from toBeProcessedStatesQueue.
  2. Get from stateTable all the transitions that starts with currentState and insert into stateTransitionTable Hash Map.
  3. For each transition transition_edge, nextState that starts with currentState.
     - IF entry of nextState exists in stateTable & nextState is not final state
       - IF state not already visited ie. visitedStates Hash does not have entry for nextState, THEN put nextState into toBeProcessedStatesQueue.
       - ENDIF
     - ELSEIF nextState key is not present in stateTable Hash Map and nexState is composite state *Q* ie. multiple states separated by colon (:)

       (*a*) ForEach state s present in composite state *Q*

       - IF (stateTable.get(s) != null & !s.equals(finalState)) THEN createNewStates with all the transition edges t of s and replacing state s in composite state *Q* by the destination of *t*.

         [ie. if *Q* = *p*:*q* and there is a entry in state Table *p*, *t*5, *n*

         Then new state will be *p*:*q*, *t*5, *n*:*q*

         ENDIF

       (*b*) if(visitedStates.get(nextState) == null)

         toBeProcessedStatesQueue.enqueue (nextState);

         visitedStates.put(nextState,new Integer(1));

END WHILE LOOP

STEP C: Find various Test Scenario [Recursive]

[ASSUMPTION: we are not going to infinite loop, if cycle we visit a state at most twice.]

(*a*) findTestScenario(stateTransitionTable, currentState, pathDiscoveredTillNow, finalState)

1. if (currentState == finalState
   - PRINT TestScenario : " + pathDiscovered TillNow and return
2. For each transition transition_edge, nextState that starts with currentState.
   - String pathString = pathDiscovered TillNow + "-[" + transition + "]->" + nextState
   - if (stateTransitionTable.get(nextState) != null)
     - o IF nextState is visited < twice
     - o THEN findTestScenario(graphMap, nextState, pathString, finalState)
     - o ELSEIF (nextState = finalSTate) findTestScenario(graphMap, nextState, pathString, finalState);

MAIN :

Call STEP A to read the input and populate stateTable HashMap

Call STEP B to create new virtual states to deal with fork -join and generate state transition table Hash Map stateTransitionTable

Call STEP C to print the possible test scenarios.

REFERENCES

[1] Alan W. Brown, Kurt C. Wallnau, "The Current State of CBSE", *IEEE Transaction on Software Engineering*, September 1998, pp. 37-46.

[2] Aynur Abdurazik and Jeff Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", *Proceedings of George Mason University, Fairfax VA 22030, USA The Third International Conference on the Unified Modeling Language (UML'00)*, York, UK, October 2000, pp. 383-395

[3] Cem Kaner, J.D., Ph. D, "What is a Good Test Case", *Proceedings of Florida Institute of Technology Department of Computer Sciences STAR East*, May 2003.

[4] D.M. Cohen, S.R. Dalal, A. Kajla, G.C. Patton, "The Automatic Efficient Test Generator (AETG) System", *Proceedings of Bellcore Morristown*, NJ Piscataway, NJ.

[5] Dong Xu, Huaizhong Li, Chiou Peng Lam, "Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams", *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)* 0-7695-2465-6/05 © 2005 IEEE.

[6] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, "Test Cases Generation from UML Activity Diagrams", *Proceedings of Information and Communications University Korea* 0-7695-2909-7/07 2007 IEEE.

[7] Jeff Offutt and Aynur Abdurazik, "Generating Tests from UML Specifications", *Proceedings of George Mason University*, Fairfax VA22030, USA.

[8] Jim Q. Ning, "Component-Based Software Engineering", *Proceedings of U.S National Institute of Standards and Technology's Advance Technology Program on Component Based Software*, Document Number 0-8186-7940-9/97, 1997, pp. 34-43.

[9] Jon Edvardsson, "A Survey on Automatic Test Data Generation", *In Proceedings of the Second Conference on Computer Science and Engineering in Linkoping*, pp. 21-28. ECSEL, October 1999.

[10] Mattias Riebisch, Ilka Philippow, Marco Gotze, "UML-Based Statistical Test Case Generation", *Proceedings of I1menau Technical University*, Max-Planck-Ring 14, D-98684 I1menau, Germany.

[11] M. Prasanna1 S.N. Sivanandam2 R.Venkatesan3 R.Sundarrajan4, "A Survey On Automatic Test Case Generation", *Department of Computer Science and Engineering PSG College Of Technology*, Coimbatore 641 004.

[12] S. Mahmood, R. Lai and Y.S. Kim, "Survey of Component-based Software Development", *Proceedings of the Institute of Engineering and Technology 2007*, pp. 57-66.

[13] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong and Zheng Guoliang, "Generating Test Cases from UML Activity Diagram based on Gray-Box Method", *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*1530-1362/04 IEEE.

[14] Yizheng Yao and Yingxu Wang, "A New Approach to Test Case Generation based on Real-Time Process Algebra (RTPA)", *Theretical and Empirical Software Engineering Research Center Dept. of. Electrical & Computer Engineering*.