COMPONENT TESTING USING FINITE AUTOMATA

Damini Verma
Student, M.Tech-Software Engineering,
Yamuna Nagar, Haryana-135001, India
daminiverma611@gmail.com

Karambir
Assistant Professor, CSE department,
U.I.E.T, Kurukshetra University,
Kurukshetra, Haryana-136119, India
bidhankarambir@rediffmail.com

Abstract

In Component-Based Software Engineering (CBSE), software systems are mainly constructed with reusable components, such as third-party components and in-house built components. Component Based Software Development (CBSD) is used for making the software applications quickly and rapidly. In Component Based Development (CBD), the software product is built by gathering different components of existing software from different vendors. This process reduces cost and time of the software product. But for a tester, many difficulties arise in testing phase because the tester has a limited access to source code of reusable component of the product. This concept is known as Black-Box Testing (BBT) of software components because Black box testing is used where source code of the component is not available. The additional information with the components can be used to facilitate testing. This paper has its focus on testing of an application using Finite Automata-based testing which covers two types of testing, viz. NFA-based testing and DFA-based testing. The working of the application is explained with the help of UML diagrams.

Keywords: Component Based Software Development (CBSD); Unified Modeling Language (UML); NFA; DFA; Software Testing.

1. Introduction

Component-Based Software Development (CBSD) has been regarded as a next generation technology for fast and cost effective system development. In this approach, a software system is developed by assembling appropriate components from a repository of components. In order to ensure that a Component-Based Software System (CBSS) can run properly and effectively, the qualities of the constituent components need to be assured. As a CBSS is a collection of new and reused components, the components can communicate with other components through their interfaces. CBSD approach builds software systems by assembling pre-existing components under well-defined architecture which brings high reusability and easy maintainability to the component, and also reduces its time-to-market. Therefore, the productivity of software systems is improved and the development cost is also reduced.

In order to test component-based software, it is necessary to first understand what is software testability? According to IEEE Standard, the term "testability" refers to “the degree to which a system/component aids in the creation of test criteria and the performance of tests determine whether those criteria have been met or not”.

This paper uses Unified Modeling Language (UML) to explain the working of an application. UML is a standard general purpose modeling language used in object oriented software engineering. By definition, “UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems”. UML gives us the understanding of static and dynamic nature of a system.

Finite automata are used to test the application. Finite Automata are considered a very useful model for pattern matching, lexical analysis and for verifying all kinds of systems that have a finite number of distinct states for secure exchange of information. According to automata theory, “Non deterministic Finite Automata (NFA) is a finite state machine where the automaton may jump into several possible next states from each state with a given input symbol.” On the other hand, “a Deterministic Finite Automata (DFA) is a finite state machine that accepts/rejects finite strings of symbols and produces a unique computation of the automaton for each input string.” Although, DFA and NFA have distinct definitions and distinct rules, NFA can be converted into equivalent DFA.

This paper is organized as follows. First section gives the introduction to the methods and techniques followed for the completion of this paper. Section 2 contains the related research papers used for the survey purpose of this paper. Section 3 contains the UML diagrams for this paper. Section 4 implements the NFA and DFA-based testing of application. Section 5 shows the testing results. The paper is concluded and the future work is presented in last two sections.

2. Related Work

This section covers a review of the work done in previous publications related to component testing and uses them for the research work required for this paper.

In the year 2003, Ye Wu et. al. [1] presented a paper which stated that component-based software engineering is used for software development. This approach uses reusable components as building blocks for constructing software. This improves the quality and productivity of software, but it requires frequent maintenance activities. The cost of maintenance of software takes as much as two-thirds of the total cost, and it is likely to be more for component-based software. This paper presented a UML-based technique that attempted to help resolve difficulties introduced by the implementation transparent characteristics of CBSS. This technique was also useful for other maintenance activities.

In the year 2008, Bin et. al. [2] proposed a methodology to test component configuration based on mutation. This methodology stated that CBSD raises the developing efficiency and quality of software effectively. The users of component configure the tested components. Because of unavailability of component’s source code, errors of component configuration are difficult to be tested. This paper proposed a framework of mutation based component configuration testing and also increased the quality and the degree of automation of component configuration testing.

In the year 2010, Jiang et. al. [3] proposed a method for adequate testing of black box components. The unavailability of source code of black box components makes it difficult to generate test data and to ensure test adequacy. In this paper, an extended component-interface specification model was proposed to understand, test and reuse the component. Then the function of different types of specification elements in testing was defined. Based on the syntactic and semantic specifications, the proposed test data generation method produced test suite meeting a certain mutation score, which was viewed as a kind of effective criteria for test adequacy. Finally, some experiments were carried and the results had shown that the different kinds of specification supported the testing of black-box components.

In the year 2010, Naseer et. al. [4], proposed that in a component-based system there arise many difficulties for a tester because the tester has a very limited access to the source code of the reusable component of the product. In Component Based Development, the software product is built by uniting different components of pre-existing software from different vendors. By using this process, cost and time of the software product was reduced. But in testing phase, tester faced many difficulties because the tester had a very little access to the source code of the reusable component of the product. The component meta-data was used to attach additional information with the components to facilitate testing. Black-box testing was used where code of the component was not available. Usually, a component had a hidden interface and a tester was not able to input the values in it unless its interface was not completed. In this paper, the issues in component based testing using meta-data approach for black-box testing was discussed when component’s interface not available. It also presented the methodology how meta-data was used in for black-box testing.

In the year 2011, Mohanty et. al. [5] proposed a model based prioritization technique for component based software retesting using UML state chart diagram. A prioritization technique does scheduling of the test cases execution so that the test cases with higher priority are executed first followed by test cases with lower priority. Test case prioritization aims at detecting fault as early as possible so that the debuggers can begin their work earlier. This paper proposed a new prioritization technique to prioritize the test cases to perform regression testing for CBSS. The components and state changes for a CBSS were represented by UML state chart diagrams which were then converted into Component Interaction Graph (CIG) to describe the inter-relation among components. The prioritization algorithm took this CIG as input along with the old test cases and generated a prioritized test suite that took into account the total number of state changes and total number of database access, both direct and indirect, encountered due to each test case. This algorithm was found to be very effective in maximizing the objective function and minimizing the cost of system retesting when applied to few Java projects.

3. Implementation Of Component-Based Testing

The working of the application is explained with the help of 5 UML diagrams:

3.1. *Use Case Diagram*

Use Case Diagram captures the dynamic behaviour of the system. Dynamic behaviour is the behaviour of the system when it is in running state. The following diagram shows the respective functions that actor (user) and administrator can perform on this application respectively.

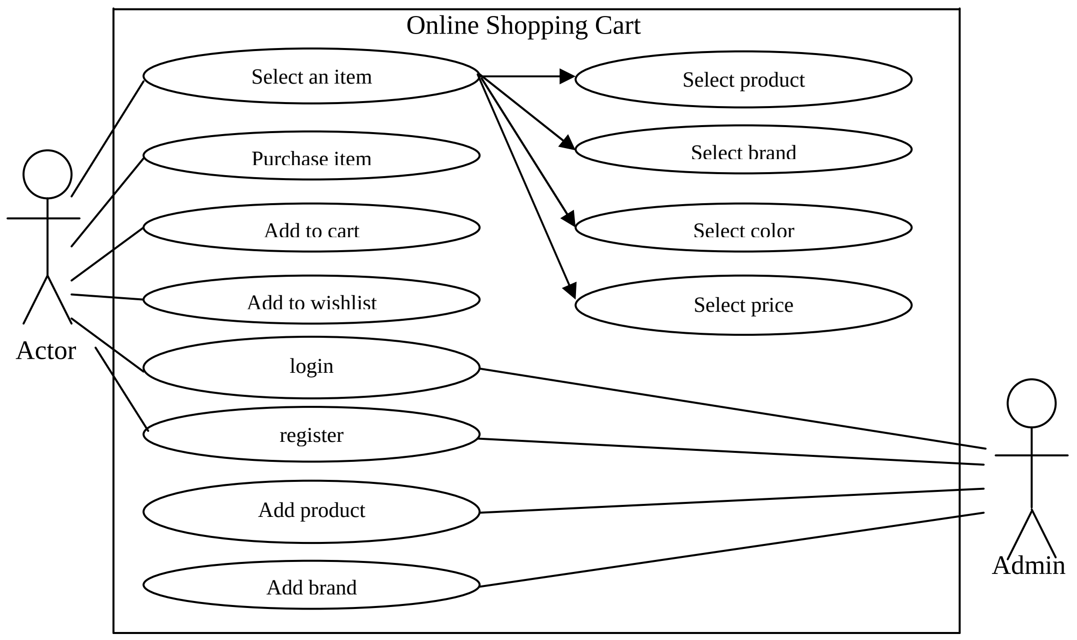

Figure 1. Use Case Diagram

3.2. *Sequence Diagram*

Sequence diagram shows the message sequence between the objects and the time sequence between the messages. Sequence diagram shows the method calls from one object to another and this depicts the actual scenario when the system is in running state. Sequence diagram is drawn for the different use cases. The following two are sequence diagrams for two different use cases: login a user and purchase item. These diagrams clearly show the message sequence between the objects of the application.

Use Case: Login a User

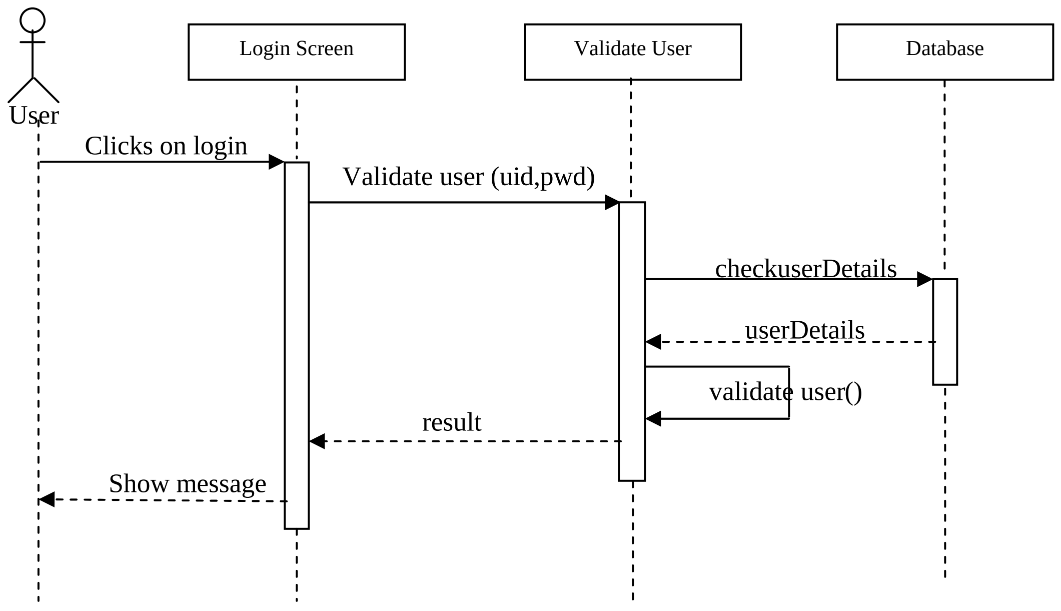

Figure 2. Sequence Diagram (Use Case: Login a User)

Use Case: Purchase Item

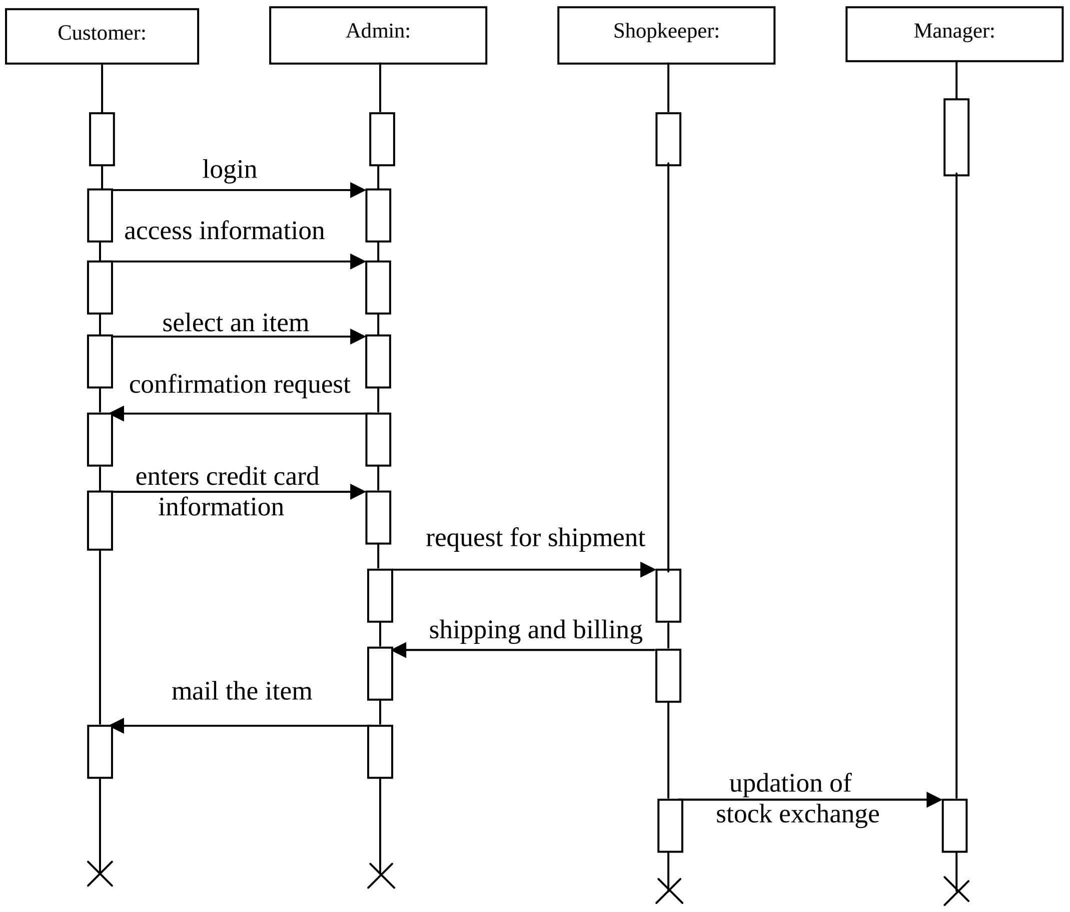

Figure 3. Sequence Diagram (Use Case: Purchase Item)

3.3. *Activity Diagram*

Activity diagram describes dynamic aspects of the system. It is just like a flow chart that represents the flow from one activity to another activity. The following diagram shows the different activities performed in this application. Customer, Admin, Billing represent different operations of this application while each node represents the activity. The flow from one activity to another is shown with the help of arrows. The sequence of activities between customer, admin and billing can be clearly understood from the diagram.

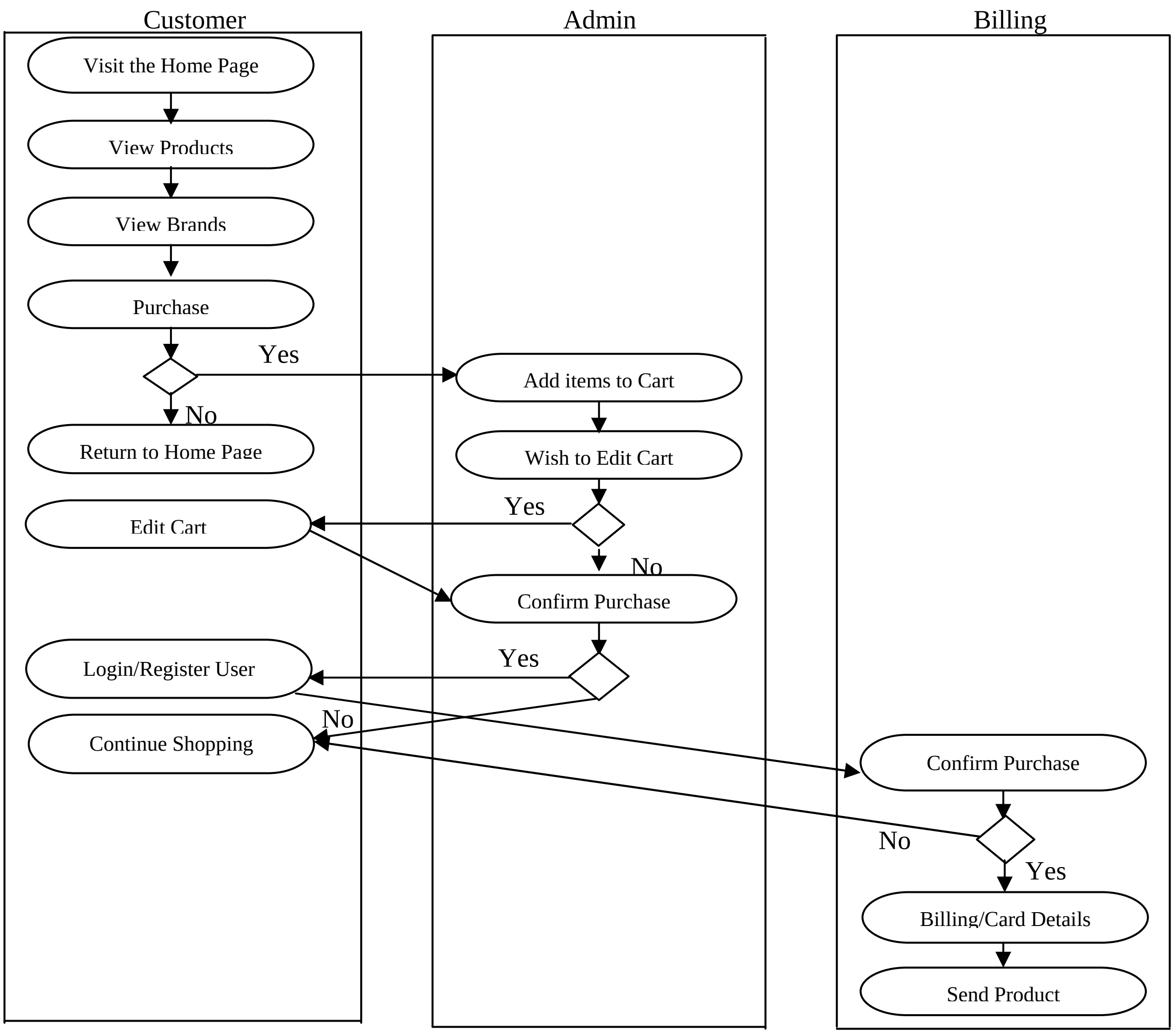

Figure 4. Activity Diagram

3.4. *Collaboration Diagram*

Collaboration diagram shows the object organization. In collaboration diagram, the method call sequence is indicated by a special numbering technique. The number indicates the sequence in which methods are called one after another. The method calls are similar to that of a sequence diagram. But the difference is that the sequence diagram does not describe the object organization where as collaboration diagram shows the object organization. Following is the collaboration diagram for this application which clearly shows the method calls between the objects.

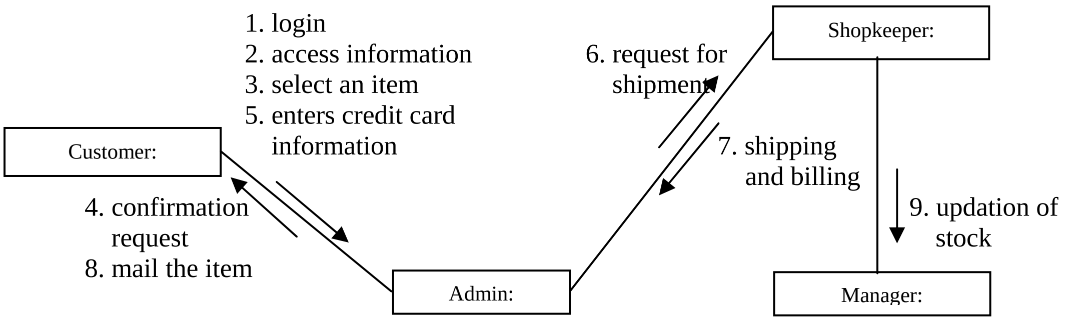

Figure 5. Collaboration diagram for purchase item (showing all steps)

3.5. *State Chart Diagram*

State chart diagram describes different states of a component in a system. The states are specific to a component/object of a system. A state chart diagram describes different states of an object and these states are controlled by external or internal events. State chart diagram is used to model lifetime of an object. Following is the state chart diagram that clearly shows different states in this application and the events that takes the control from one state to other.

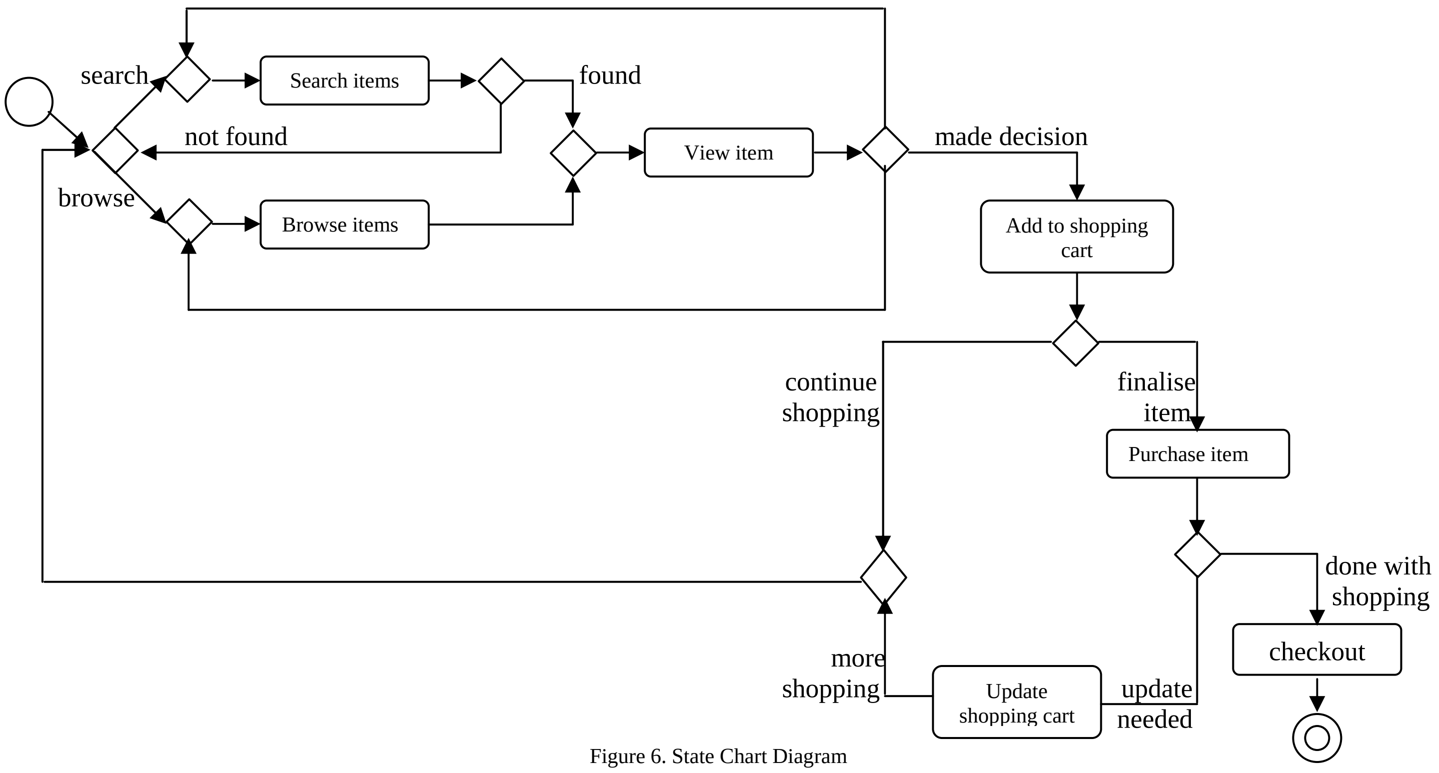

Figure 6. State Chart Diagram

4. Finite Automata-Based Testing

4.1. *NFA-based Testing*

The figure shown below i.e the NFA diagram of this application is used to test whether each and every path in the application is working correctly or not by supplying correct and incorrect inputs. For correct inputs, the path (transition) in the NFA from one state to next, is followed correctly and terminates at accepting state whereas for any incorrect input, the application does not move to next state, instead, it reaches to the same state by displaying the respective error message on the same state. This NFA-based testing proves that the application is capable of finding the error. Hence, every unique path in the NFA works correctly for correct input values and detects error corresponding to wrong input values.

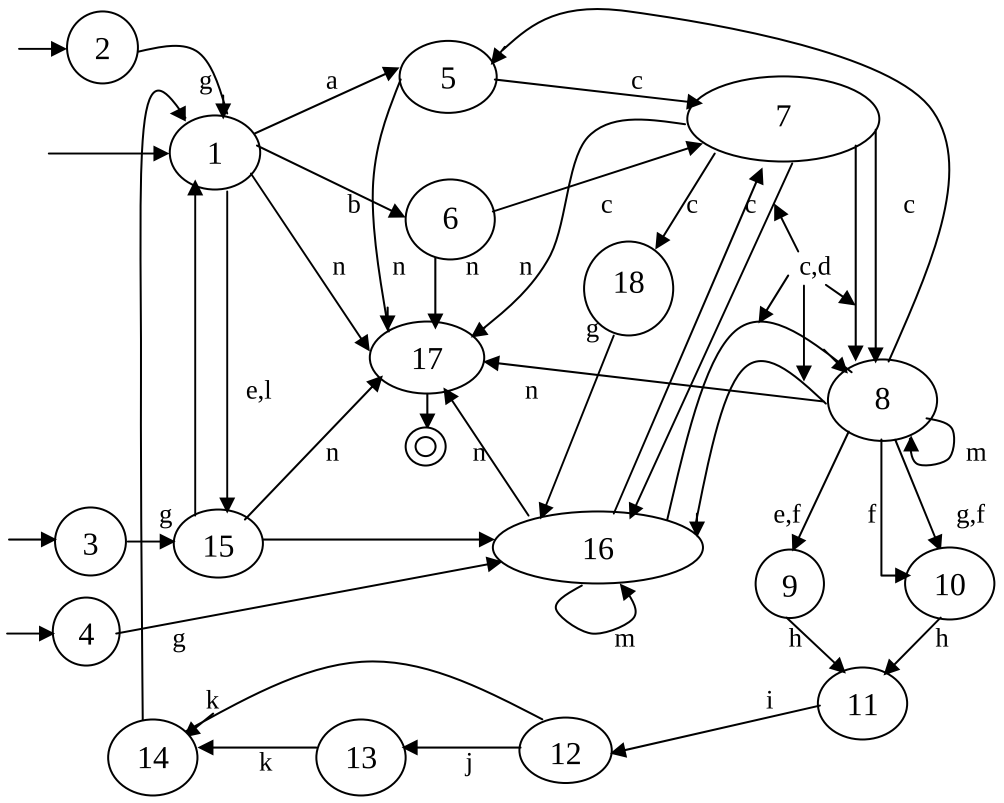

Figure 7. Testing of application through NFA

Table 1. State names and input symbols for NFA.

States in NFA	State Names	Inputs in NFA	Input Symbols
home	1	category	a
login	2	item details	b
account login	3	X (particular item)	c
wishlist login	4	size	d
browse items list	5	email	e
search items list	6	cart items	f
view item	7	login details	g
shopping cart	8	shipping address	h
enter shipping address	9	payment mode	i
existing/new shipping address	10	payment details	j
choose payment method	11	place order request	k
order summary	12	gender	l
purchase	13	remove item	m
order acknowledgement	14	logout request	n
my account	15		
my wishlist	16		
Logout Request (LR)	17		
login to move item	18		

4.2. *NFA-DFA Conversion Steps*

(1) Set of accepting states, F={17}

(2) Finite set of states, Q={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}

(3) Input symbols, ∑={a,b,c,d,e,f,g,h,i,j,k,l,m,n}

(4) Transition function, ∂ is represented by the following table:

i. Begin with Start State.

ii. See where each input takes you (may be a set of states or may be nowhere).

Table 2. Transition table to convert NFA into DFA.

∂	a	b	c	d	e	f	g	h	i	j	k	l	m	n	
1	5	6	-	-	15	-	-	-	-	-	-	15	-	17	-
2	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	15	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	16	-	-	-	-	-	-	-	-
5	-	-	7	-	-	-	-	-	-	-	-	-	-	17	-
6	-	-	7	-	-	-	-	-	-	-	-	-	-	17	-
7	-	-	8,16,18	8,16	-	-	-	-	-	-	-	-	-	17	-
8	-	-	16	16	9	9,10	10	-	-	-	-	-	8	17	5
9	-	-	-	-	-	-	-	11	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	11	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	12	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	13	14	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	14	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
15	-	-	-	-	-	-	-	-	-	-	-	-	-	17	1,16
16	-	-	7,8	8	-	-	-	-	-	-	-	-	16	17	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	16	-	-	-	-	-	-	-	-

(5) Now merge states on the basis of this transition table.

(6) Set of merged states={1,15},{1,14},{5,8},{8,16},{9,10},{1,2,3,4},{7,8,16,18}.

(7) Normalize transitions according to merged states to draw DFA.

(8) Draw DFA containing the set of merged states.

(9) The final states in DFA are a set of states containing final state of NFA.

4.3. *DFA-based Testing*

The figure below i.e, the DFA diagram of this application is also used to test whether each and every path in the application is working correctly or not by supplying correct and incorrect inputs. For correct inputs, the path (transition) from one state to next, is followed correctly and terminates at accepting state whereas for any incorrect input, the application does not move to next state, instead, it reaches to the same state by displaying the respective error message on the same state. This DFA-based testing proves that the application is capable of finding the error. Hence, every unique path in the DFA works correctly for correct input values and detects error corresponding to wrong input values.

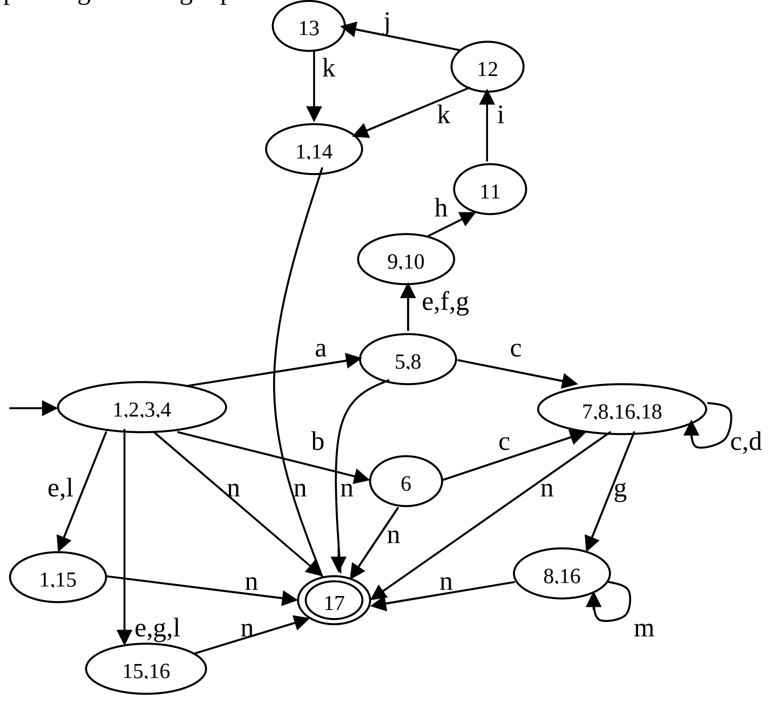

Figure 8. Testing of application through DFA

5. TESTING RESULTS AND ANALYSIS

On the basis the NFA and DFA diagrams, the workflow of the application is tested and the impact of error is recorded as well. Every unique path of the NFA and DFA is tested to find out any error on path and in case of occurrence of any error, that path is recorded and loaded into the error report for further concern.

The probability of any error is based on the fact that how critical the impact of that error is and how frequently the error prone page is traversed. Based on the impact of error and no. of page traversals, it can be concluded whether that error is negligible or critical. If the impact of error is high and page is traversed less no. of times, then the error is much critical and requires proper attention, therefore, it must be corrected as soon as possible. If the impact of error is not so high and the page is traversed many times, then the error is not so critical and does not require much attention, therefore, it can be corrected sometime later. Fault Tolerance of a particular state (or page) can be can be computes by taking the ratio of Impact of Error on Page and No. of traversals of corresponding page.

6. CONCLUSION

Today, most of the software systems are developed by using the existing code or the available components i.e Reusability. Reusability is achieved by performing some interfacing between different software components. The software reusability is presented either in terms of some code or in terms of component objects. This paper presents a new approach for testing the component-based software systems. The studies show that testing of component-based software systems is required for checking the reliability of complete system. Finite Automata-based testing is an easy way to test an application. The testing is based on the requirement of the system. If the system requirements are not met, then it shows failure otherwise success. The testing technique is quite simple, effective, and less complex and it does thorough testing of an application, but the process is time consuming because every single path (change of state) need to be tested solely which increases its time complexity.

7. FUTURE SCOPE

In future, the testing technique can be improvised to reduce the time complexity. It can be done by testing integrated paths (group of paths) instead of testing every single path, which might prove little complex but less time consuming.

References

[1] Ye Wu & Jeff Offutt (2003). “Maintaining Evolving Component-Based Software with UML”, proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR). ISBN: 0-7695-1902-4.

[2] Xia Bin & Pan Bin (2008). “Component Configuration Test Based on Mutation”, International Symposium on Intelligent Information Technology, pp. 906-909.

[3] Ying Jiang, Ying-Na Lia & Xiao-Dong Fu (2010). “The Support of Interface Specifications in Black-box Components Testing”, fifth International Conference on (FCST), pp. 305-311.

[4] Furqan Naseer, Shafiq ur Rehman & Khalid Hussain (2010). “Using meta-data technique for component based black box testing”, sixth International Conference on ICET, pp. 276-281.

[5] Sanjukta Mohanty, Arup Abhinna Acharya & Durga Prasad Mohapatra (2011). “A model based prioritization technique for component based software restesting using UML state-chart diagram”, 3rd International Conference on Electronics Computer technology, Vol. 2, pp. 364-368.