ADBT Frame Work as a Testing Technique: An Improvement in Comparison with Traditional Model Based Testing

[1]Mohammed Akour, [2]Bouchaib Falah, [3]Karima Kaddouri
[1]Computer Information Systems Department, Yarmouk University
[2,3]School of Science and Engineering, Al Akhawayn University

***Abstract*—Software testing is an embedded activity in all software development life cycle phases. Due to the difficulties and high costs of software testing, many testing techniques have been developed with the common goal of testing software in the most optimal and cost-effective manner. Model-based testing (MBT) is used to direct testing activities such as test verification and selection. MBT is employed to encapsulate and understand the behavior of the system under test, which supports and helps software engineers to validate the system with various likely actions. The widespread usage of models has influenced the usage of MBT in the testing process, especially with UML. In this research, we proposed an improved model based testing strategy, which involves and uses four different diagrams in the testing process. This paper also discusses and explains the activities in the proposed model with the finite state model (FSM). The comparisons have been done with traditional model based testings in terms of test case generation and result.**

Keywords—Activity Diagram; Black Box Testing; Finite State Machine; Model-Based Testing; Software Testing; Test Suite; Test Case; Use Case Diagram

I. Introduction

Software testing is an important, if not the most important, activity in the software development cycle of any system without exception. It is an intellectually challenging activity aimed at evaluating the capability of a program or system to determine whether or not it meets requirements [1].

Software testing is defined as the validation and verification of the proposed system or product to ensure that it conforms to the agreed-upon requirements, that it is functioning as expected by both the developer team and the stakeholders, and that it satisfies the latter. As software testing can get very difficult or costly to perform, software testing engineers are always developing new or refining existing testing techniques tools, always having in mind the objective of developing the optimal approach that would ideally be cost-effective and efficient at the job simultaneously. In other words, the ideal testing methodology is one with maximum coverage and minimal cost. Since testing occurs right after the development phase, it is an on-going process which might take place earlier in the software development cycle. Consequently, there are two essential methods of software testing:

- Black box (also called functional testing) to be specific, which are used to test and validate the requirements and design of the system before moving on to the implementation.
- White-box testing (also called structural testing and glass box testing) is testing that takes into account the internal mechanism of a system or component.

Each method encompasses a lot of different testing techniques. One of the most important techniques of the black box testing method is called Model-Based Testing (MBT).

Our suggested approach aims to overcome some of the traditional MBT challenges. The idea behind this approach is to base the testing process on abstract representations of the system just as MBT does, but -unlike MBT- manually generating corresponding test cases. Our approach will test the software by testing its design. Based on the system requirements, this new testing technique which was inspired from the traditional MBT paradigm will use three UML diagrams instead of the FSM diagram. Detailed description is covered in the next sections.

The rest of the paper is organized as follows: Section II describes the traditional MBT. Section III explains our proposed approach. Section IV presents some related works. Section V presents a comparison between ADBT and MBT. Finally, we conclude the paper in section VI.

II. Traditional MBT

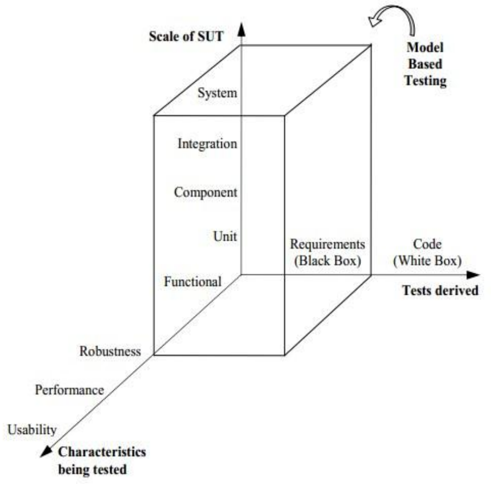

Fig. 1. MBT Context [3]

Model-Based testing refers to the black box testing technique where test cases are automatically generated based on a model, which represents the behavior of the system under test (SUT), and on the system's requirements and specification [2]. To clearly understand the scope of Model-Based Testing, Figure 1 illustrates the MBT context.

In the software development cycle, it is often required to model the system to be implemented and design an abstract view of its functionalities. There are many models available for testers to represent and model abstract depictions of systems. Some of which are UML diagrams, Markov chains, grammars, state charts, and finite state machines [4].

In traditional model-based testing, the model used to generate test cases is the finite state machine diagram (FSM) [4]. Finite state machines or finite state automata are mathematical models of computation. They are used at both hardware and software levels [5]. An FSM is the description of a finite set of states of a particular machine and the transitions between those states [5]. The events responsible for a transition from one state to another state are triggers [5]. In other terms, an FSM is a diagram which represents the set of system's states and the triggering events or conditions responsible for switching between states [5].

Figure 2 shows an example of a simple phone system. The SUT is a phone with a set of states [6]. Those states are represented as nodes, while the actions the user performs are represented as edges (triggering transitions) [6]. For instance, a possible system input to be tested could be: Pick Up. After getting the model of the SUT, the next step is to use a test case generator to generate test cases [6]. An example of a produced test case from this phone system model could be the following sequence of actions and states: <Ringing ->Hang Up -> On Hook ->Pick Up>.

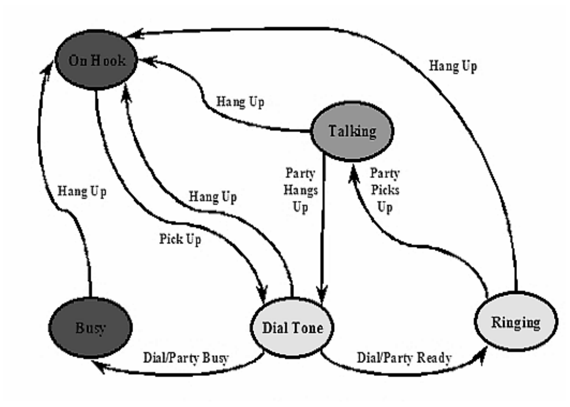

Fig. 2. Example of FSM (Phone System) [5]

The test case generator then proceeds to generate test cases for the given model, which could be based on some specific coverage criteria, such as a particular set of requirements [4]. The result of this process is an abstract test suite which needs to be concretized and converted into an executable set of test cases [4].

Test scripts or test drivers perform this operation and map each abstract model test case to an executable one using what is called an adaptor code, developed in C, Java, C#, or any other application language [4].The executable test suite is then run, and the final results are reported and analyzed. In the presence of faults, the failure is traced back; the model might be modified if deemed necessary and the testing process is repeated [4]. Figure 3 presents the process of MBT.

Model-based testing has many benefits [3]. Among its advantages is the fact that it fills the gap between the abstract and concrete levels of the system (enhances traceability between each executable test case and its corresponding part in the model and vice versa) thanks to the scripting tools [3].It also provides efficient fault detection and improves test quality, since it generates a set of non-repetitive test cases for a given SUT [3].

Some problems with traditional MBT are summarized as follows.

While having noticeable advantages, MBT also has its set of drawbacks and limitations [3].

Useless test cases

Not all the test cases generated automatically are useful for the testing process of the SUT. Traditional MBT produces a huge set of test cases, not all of them possible or beneficial for testing. This problem leads to additional cost and time to filter and get through all the test cases to choose the valid ones [3].

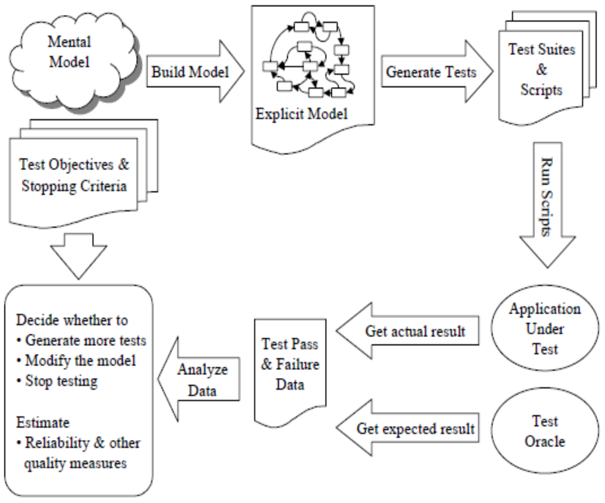

Fig. 3. MBT process [3]

Skills limitations

The test engineer has to demonstrate mastery of different skills such as high level of knowledge about computational system models or expertise in automation tools and scripts. This effort requires additional training costs.

FSM problem

The most issue with using finite state machines is state space explosion. For complex systems or large programs, the number of states in the FSM can grow uncontrollably and might exceed the given computational capabilities. This problem affects test coverage quality and efficiency and test case generation.

Failed tests issue

When a failure occurs, it can either be due to the system under test, the model, the test case generator, or the adaptor code used for conversion. Those many possible origins for failure increase the difficulty to trace back a failed test along with being extremely time-consuming [3].

III. PROPOSED ACTIVITY DIAGRAM BASED TESTING TECHNIQUE (ADBT)

The suggested testing approach aims to overcome these challenges by manually generating test cases instead of using a program for the task. The idea behind this approach is to base the testing process on abstract representations of the system just as MBT does, but – unlike MBT – manually generating corresponding test cases.

Our approach will test the software through testing its design. Based on the system requirements, this new inspired testing technique, from the traditional MBT paradigm, will use three UML diagrams instead of the FSM diagram: Use Case, Class and Activity diagrams. From the use case diagram, we derive the corresponding activity diagrams. The activity diagrams will present a set of numbered steps. Each path/scenario in the activity diagram of a single use case corresponds to a test case.

The test cases are then set up in test case tables divided into “Steps” and “Input/output”. “Steps” corresponds to the activity diagram numbered steps while Inputs are test points and Outputs are expected results from the system. The class diagram is needed to get the system’s variables (attributes of classes) used while setting up the test cases table.

The ADBT Steps are presented as follow:

1) Retrieve Use Case diagram from requirements: After getting the agreed upon set of specifications, the system designer will model the use case diagram for the system which represents the functional requirements.

2) Derive Class Diagram

3) Develop Activity Diagrams from Use Case Diagram: Each use case scenario in the use case diagram corresponds to an activity diagram.

4) Set test case for each Activity Diagram path: Each path from start to end in a given activity diagram is a test case.

5) Check test case results: In the case of error, the models used are checked for consistency problems or faults and use cases are generated again. In an optimal workflow setting, the design team will derive the necessary diagrams for the test engineers who will directly use them and only perform the last step of ADBT.

IV. CASE STUDY

As a case study, Online Movie Tickets Purchase scenario is employed and discussed.

Let us consider the following system requirements:

- The customer should be able to search for a movie by title, with the output being its price, its time and its days of screening. In case it is unavailable, the user gets an error message and is asked to re-enter its title.
- The customer should be able to buy tickets for an available movie. He has to enter his/her credit card credentials. In case there is an error, the payment process is canceled, and the user has to re-enter the payment information.
- The customer should be able to get a ticket receipt.
- The customer should be able to display the list of all movies.

Step 1: Use Case Diagram

This system results in a properly simple use case diagram with four use case scenarios. Figure 4 shows the use case diagram for online movie tickets purchase system.

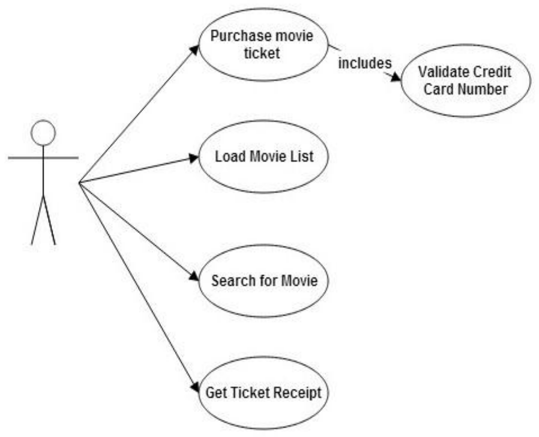

Fig. 4. Use Case Diagram

Step 2: Class Diagram

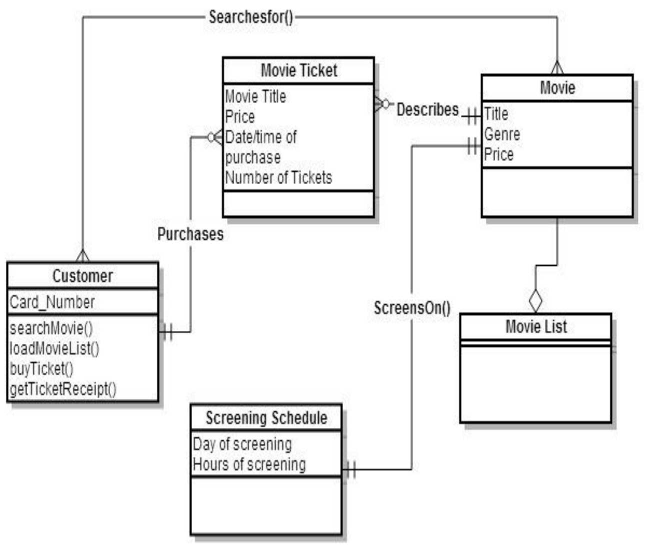

Fig. 5. Class Diagram

A tentative class diagram for this system could be as specified in Figure 5. The class diagram is vital for the I/O flow as it provides explicit information about the different system variables or attributes which we might need to test. Figure 5 shows the class diagram for online movie tickets purchase system.

Step 3: Activity Diagrams

According to the previous use case diagram, there are four use case scenarios. However, the last two requirements (get ticket receipt and load movie list) are impossible to test in the design phase and can be represented with an activity diagram. Thus, we will consider the two first requirements (search for the movie and pay movie ticket) for this particular example as shown in Figures 6 and 7.

Activity Diagram#1: Search for Movie

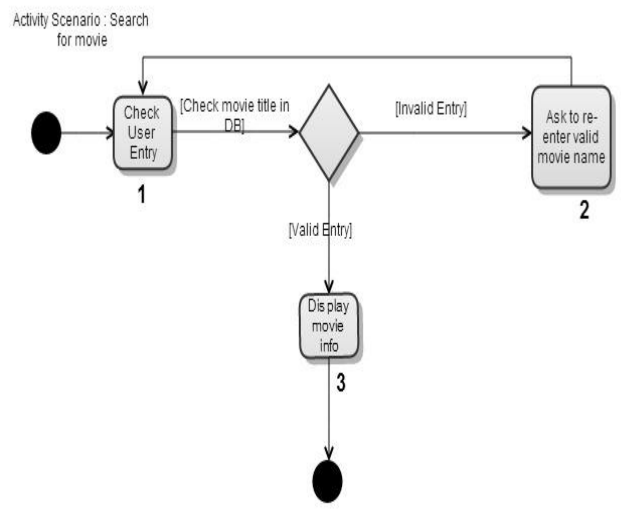
Activity Scenario : Search for movie
Check User Entry
1
[Check movie title in DB]
[Invalid Entry]
Ask to re-enter valid movie name
2
[Valid Entry]
Display movie info
3

Fig. 6. Activity Diagram 1

Activity Diagram#2: Pay Movie Ticket

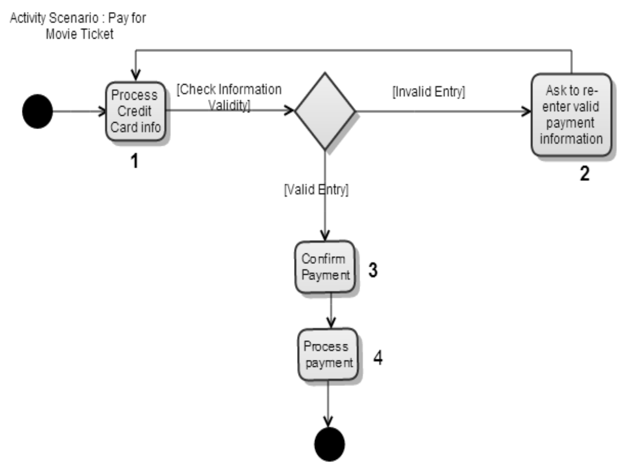
Activity Scenario : Pay for Movie Ticket
Process Credit Card info
1
[Check Information Validity]
[Invalid Entry]
Ask to re-enter valid payment information
2
[Valid Entry]
Confirm Payment
3
Process payment
4

Fig. 7. Activity Diagram 2

Step 4: Test Cases

Since each path in each activity diagram results in test cases, we have four test cases in total.

Test case#1

TABLE I. TEST CASE 1

Step	I/O
1	Input: "HY7LO."
2	Output:"Not Available-Please Enter Valid Title"

Test case#2

TABLE II. TEST CASE 2

Step	I/O
1	Input:"Hunger Games"
3	Output:"Hunger Games, price: 30dhs, MWF 2h-4h."

Test case#3

TABLE III. TEST CASE 3

Step	I/O
1	Input: "zzz zzz zzz"
2	Output: "Invalid Payment Information-Try again."

Test case#4

TABLE IV. TEST CASE 4

Step	I/O
1	Input:"1234569921"
3	Output:"Your Payment has been confirmed."
4	(no I/O, only processing)

Step 5: Checking Test Cases Results

There are no errors in the test cases; the I/O flow is correct and expected, which indicates that our models are consistent.

V. COMPARISON OF ADBT WITH MBT

A. *Test case generation:*

The state machine diagram for the previous online purchase ticket was implemented, resulting in some number of 11 states. For state-based test case coverage, each state has to be visited at least once, meaning that we will generate at least 11 test cases in addition to other relevant or irrelevant paths/test cases (since we have automatic random path selection). ADBT for this simple example results in 4 test cases.

B. SUT Model:

The ADBT approach uses the Activity Diagram for testing; however, it needs to first design the Use Case and Class as well. MBT only uses FSM. However, and as suggested earlier, the test engineer does not necessarily design all three models as it could be and usually is part of another team's work.

C. Test case results:

The test case results for the four test cases are clear and significant and easily allow checking for possible model errors. Concerning MBT, generating and running test cases is beyond this project's scope due to the complexity of the process (needs automation tools, coding test scripts, etc…). However, due to the number of generated test cases, we can assume that it will be cumbersome to filter through all of them, choose the relevant ones for execution, and proceed to trace back errors found since it could have – as previously mentioned – many origins (code, test script, model, test case generation, etc…).

VI. Related Work

Hemmati, *et al.* [7] propose an alternative technique to MBT, which is supposed to overcome one of MBT's most important issues, that is, the enormous number of generated test cases which impact negatively on both time and cost.

Their approach implements a smart test case selection technique based on genetic algorithms which choose a test suite from the large pool of generated test cases to be executed based on resources and maximum fault coverage criteria, thus extending traditional MBT into a more time/cost saving version [7].

Arnold, *et al.* propose a scenario-based approach to traditional MBT [8]. Tests are executed automatically and exist within the scope of a large pool of states in MBT, which makes it harder to trace them back directly to the SUT.

This new approach makes test execution semi-automatic and introduces scenario-based test cases which are much more relevant and closer to the SUT because the set of these applicable states is manually selected.

The approaches presented in [9, 10] utilized a model based for software security testing and software test selection perceptively. In [10], authors implemented model-based approach to tracking vital items in test models and its corresponding item in structure model. When any modification occurs in the component model of the software under test, the component model identifies and conveys changes that should be performed to update the corresponding test model.

Mohacsi *et al.* [11] adopted a model-based test (MBT) approach for systematic test design and generation of their case study. They believed in that MBT assured modularity and abstraction, moreover, it leads to decrease the required effort for test maintenance. Their model based testing is build based on activity diagrams. One of the main lessons learned from their case study is the reduction of the test effort, especially the effort for test maintenance.

Yanjun, *et al.* [12] proposed new model-based testing process in order to improve structural coverage in functional testing. They concentrate on integrating three main parts, specification-based test generation tool, a model-checker and an environment for model test execution to enhance structural coverage rate. Their MBT process facilitates capturing suspicious code branches that require analysis to determine whether they are truly unreachable or a bug is occurring in a condition guarding this branch. Moreover, Model checking allows extending the functional test set by test cases derived from uncovered branches.

Amalfitano, *et al.* [13] proposed and implemented a new fully automatic technique to test GUI-based Android apps. Their technique is composed of 3 main steps namely, observation, extraction, and abstraction of the run-time state of GUI widgets. The abstraction is employed to develop a scalable state-machine model that, together with event-based test coverage criteria provide a way to automatically generate test cases. They performed their technique on 4 open-source software applications. The results showed that the test cases generated were useful at detecting serious and relevant bugs in the apps.

VII. Conclusion

Software Testing is and will remain the most important, but also the trickiest and most challenging activity in the software development cycle. There is an abundance of testing techniques in the literature, and one of them is a black box testing technique called Model-Based Testing.

This paper presents an improvement on testing technique to overcome some of the traditional MBT challenges. Our approach is based on Activity Diagrams.

Following are the advantages of using ADBT.

- No more useless or irrelevant test case generation problems.
- The FSM diagram state explosion is resolved since we changed the model.
- Errors can be easily traced back to the model in case of failure.
- The tester does not need extra training skills to conduct the testing process, since the three UML diagrams can be provided by the design team and made available for the tester to only generate and execute tests.

This new technique has been demonstrated as having some benefits.

It is more costly and less composite than traditional MBT, it allows for easily testing the consistency of the software design and checking if it conforms to what is expected from the customer, and finally it provides an easy and systematic way of generating test cases. As future works, we intend to conduct more rigorous validation to make our result well proven.

References

[1] B.Falah, K.Magel, O.ElAriss, "*A Complexity Based Regression Test Selection Strategy*," Computer Science & Engineering: An International Journal (CSEIJ), Vol.2, No.5, October 2012

[2] M. Utting, A. Pretschner, and B. Legeard.Taxonomy of Model-Based Testing.In *Working Paper Series*, vol. 04, April 2006.

[3] Y. Malik. *Model Based Testing: An Evaluation*. Master Thesis. Reading University: Blekinge Institute of Technology, May 2010.

[4] K. El-Far and A. James. Model-based Software Testing. In *Encyclopedia on Software Engineering*, J.J. Marciniak (ed). Wiley, 2001.

[5] T. Coquand. *Finite State Machines. In Automata*, pp 471-545, University of Gothenburg Press, September 2010.

[6] A. Chandran. *Model-Based Testing: Executable State Diagrams*. In proceedings of the STEP-AUTO 2011 Conference for International Testing (ISQT' 11), pp 10-17, 2011.

[7] H. Hemmati, L. Briand, A. Arcuri, and S. Ali.*An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study*. Simula Research Laboratory.Technical Report, 2010.

[8] D. Arnold, J.P. Corriveau, and W. Shi. *A Scenario-Driven Approach to Model-Based Testing*.2010. http://people.scs.carleton.ca/~jeanpier/VF_test_generation.pdf

[9] Bouchaib Falah, Mohammed Akour, Samia Oukemeni, An Alternative Threat Model-based Approach for Security Testing. International Journal of Secure Software Engineering (IJSSE), IGI, Vol. 6 issue 3 (2015): 50-64.

[10] Ahmad Saifan, Mohammed Akour, Iyad Alazzam, Feras Hanandeh, Regression Test-Selection Technique Using Component Model Based Modification: Code to Test Traceability, IJACSA, 2016

[11] Mohacsi, Stefan, Michael Felderer, and Armin Beer. "A Case Study on the Efficiency of Model-Based Testing at the European Space Agency." 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015.

[12] Sun, Yanjun, Gérard Memmi, and Sylvie Vignes. "A Model-Based Testing Process for Enhancing Structural Coverage in Functional Testing." Complex Systems Design & Management Asia. Springer International Publishing, 2016. 171-180.

[13] Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. M. (2015). MobiGUITAR: Automated Model-Based Testing of Mobile Apps.Software, IEEE, 32(5), 53-59