Future Generation of Communication and Internet of Things (FGCIOT)

Journal homepage: http://fgciot.semnaniau.ac.ir/

PAPER TYPE? (Research paper.)

A Framework for Model-based Testing

Arash Sabbaghi [1]

[1] *Department of Computer Engineering, Semnan Branch,Islamic Azad University, Semnan, Iran*

a.sabbaghi@semnaniau.ac.ir

Article Info

Article History:
Received
Revised
Accepted

Keywords:
Software testing, Model-based testing, Automatic test case generation, Test models.

*Corresponding Author's Email Address:
a.sabbaghi@semnaniau.ac.ir

Abstract

Model-based testing (MBT) has attracted a lot of attention and has been extensively applied in different areas such as probabilistic systems, mobile systems, concurrent systems, real-time systems, software product lines, etc. However, MBT approaches have some limitations and challenges that are mostly related to the incompleteness, high level of abstraction, complexity, and also the informal nature of input models. In the literature, there are different studies addressing these problems. In this paper, we represent a framework for model-based test case generation approaches according to the aforementioned challenges. In this regard, firstly, we categorize different coverage criteria used in MBT, and then show that MBT approaches have three main steps: preprocessing, test scenario generation, and concrete test case generation. Finally, for each step, we represent its necessity and classify the proposed methods.

Introduction

Test case generation is one of the most expensive parts of software testing [1], which includes a series of operations to generate a test set that satisfies a certain coverage criterion. There are different techniques for automated test case generation, such as symbolic and concolic execution [2], random testing [3], combinatorial testing [4], search-based testing [5], and model-based testing (MBT) [6]. To generate test cases, these techniques utilize different software artifacts, which lead to testing different aspects of the system under test (SUT). Among these artifacts, SUT models, which are utilized in MBT, represent a significant opportunity for software testing. One of the greatest achievements of MBT is the earlier detection of faults. Meanwhile, it allows the improvement of specifications and design. Therefore, MBT approaches have attracted a lot of attention and have been extensively used in different areas such as probabilistic systems [7], mobile Systems [8], concurrent systems [9, 10], real-time systems [11], and software product lines [12]. But, on the other hand, they face some limitations and challenges.

In MBT, the quality of input models has a direct impact on the quality of the test suite, and subsequently, on the effectiveness of the whole testing process. Incompleteness, high level of abstraction, complexity, and also the informal nature are the hallmarks of many input models, which adversely affect their usefulness in MBT. In case of incompleteness and high level of abstraction, input models should be augmented with necessary information that may not be readily available in the design artifacts [13]. Also, according to the informal nature and structural complexity of most input models, it is often needed to convert them into intermediate forms in order to formalize them and make them more practical for automatic test case generation process. After all, it should be noted that some models may not have a good representation of repetition, recursion, or conditional sequences, omitting many control and interface details [14]. Therefore, some preprocessing should be accomplished to make test model ready for test case generation.

Doi:

In this paper, we represent a framework for model-based test case generation approaches. In this regard, firstly, we categorize different coverage criteria used in MBT. Then we introduce the general process of MBT. We show that MBT approaches have three main steps: preprocessing, test scenario generation, and concrete test case generation. The preprocessing phase includes model augmentation and transformation, whose goal is to prepare models for test scenario generation. In this phase, according to the desired coverage criterion, some test sequences are extracted from the prepared models. These test sequences are finally concretized to be ready for use. We also classify and discuss the proposed methods for each step in full detail.
The rest of the paper is organized as follows: in section 2, we categorize the different coverage criteria used in MBT. Section 3 introduces the general process of MBT and represents each step, including model augmentation, model transformation, test scenario generation, and concrete test case generation in further detail. Finally, section 4 is dedicated to the conclusion and future works.

I. Coverage Criteria In MBT

The effectiveness of a test is based on how well the information provided by a model is covered and exercised. Test coverage criteria [15] are a set of rules which impose test requirements, direct the test case generation process toward covering appropriate elements, and give testers an adequacy criterion to measure the completeness of testing. The different coverage criteria used in MBT can be divided into five categories.

A.Structural Coverage Criteria:

These criteria target to cover building blocks of the test models and are widely used in MBT. Transition coverage criterion (covering all transitions in activity diagram [16], EFSM [17], etc.), path coverage criterion (covering all paths in the model [18]), state coverage criterion (covering all states in the state-based models [19]), each message on link criterion in communication diagram [20], and class attribute criterion in class diagrams are some examples of this category.

B.Predicate-oriented Coverage Criteria:

The goal of these criteria is to test whether the predicates in the models and in the implementation are formulated correctly. For example, the full predicate coverage [21, 22] enforces the tester to provide inputs derived from each clause in each predicate on each transition.

C.Stochastic Coverage Criteria:

In some models like Markov usage models, the stochastic criteria are used based on their nature and the probabilities of actions [23].

D.Data Flow Coverage Criteria:

This criterion uses the data flow relations to guide the test selection process and considers the assignment (definition) and usage (use) of variables along paths [24]. For example, all-du-Paths coverage demands that every simple definition-use path from the assignments of a variable to its usages, without reassignment, examines at least once. To perform data flow testing, test models should be augmented with definitions and uses of data variables.

E.Custom Coverage Criteria:

Some researchers tried to find certain problems in the SUT and defined some custom criteria which is dependent on their application. For example, authors in [25] tried to generate test cases from UML sequence diagrams for detecting deadlocks during the design phase. They first converted the sequence diagram into a wait for graph, and then by identifying the possible deadlock points, traversed the graph using all deadlocks as coverage criterion to generate deadlock paths. Sun et al. [9] proposed an approach to generate test cases for concurrent programs using activity diagrams. They proposed three coverage criteria as weak concurrency coverage, moderate concurrency coverage, and strong concurrency coverage. For example, for weak concurrency coverage, test cases are generated to cover only one feasible sequence of parallel activities between a pair of fork and join activity, without considering the interleaving of activities between parallel activities.

II. Process of Model Based Testing

In this section, we discuss the inputs, outputs, and operations that are usually performed in MBT. For each operation, we represent its necessity and classify the proposed methods.

The general architecture of model-based test case generation is depicted in Figure 1. The input of the process is test models, and the output is a set of test cases. The main operations are model augmentation, model transformation, test scenario generation, and concrete test case generation. The solid transitions show the general workflow of MBT, and transitions with dashed lines are shortcuts. For example, there is a dashed transition between Test Models and Intermediate Form, which shows that it is possible to build an intermediate form from test models without augmentation (e.g., [19, 26]).

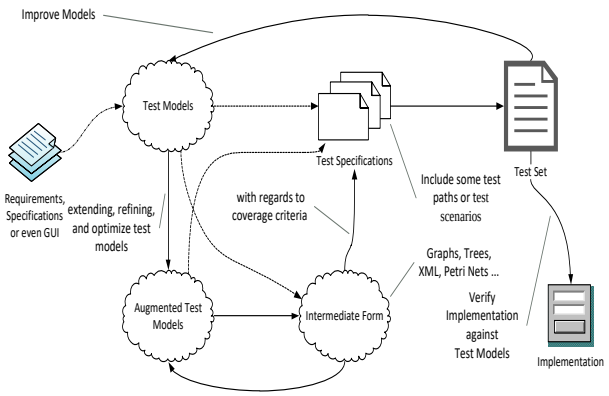

Figure1. : General architecture of automated model-based testing

Model augmentation and transformation, are some preprocessing on test models to prepare them for extracting test cases. In fact, many test models may be incomplete, complex, and informal, which causes problems for their further analysis. Therefore, their improvement leads to more successful testing. After preparing test models, it is turn to generate test paths, test sequences, or generally test scenarios. Finally, test scenarios are analyzed to generate concrete test cases. SUT is then executed with the generated test cases, and the results are used to verify implementation against test models. Notice that model-based generated test cases can also be used to verify models and improve the design by finding ambiguities and inconsistencies between requirements, specifications, and design [15, 16, 27-29]. For example, authors in [27] addressed the problem of inconsistencies between the class diagram, sequence diagram, and OCL constraints, or Eshuis [29] proposed an approach to verify the consistency of a UML activity diagram against a set of class diagrams.
Notice that some papers cover all steps of the presented architecture [30, 31], while others focus on some special steps like test scenario generation and don't present any way for automatically generating concrete test cases [26, 32]. In the following, we explain the model augmentation, model transformation, test scenario generation, and concrete test case generation steps in further detail.

III. Model Augmentation

Augmentation is mainly performed to enrich test models by extending, refining, and optimizing them. This can be done by updating or adding some test information into test models. Model augmentation increases the accuracy of test models by decreasing their level of abstraction and can help in generating oracles [33], concretizing test scenarios, and reducing the number of test cases that result in facilitating the efficiency of the testing process. On the other hand, extracting required information and the way of incorporating them into test models may pose a threat to automation. In the following, we mention some model augmentations that have been carried out in the literature.

Authors in [30] annotated use cases with pre- and post-conditions, which are used to infer the correct partial ordering of functionalities that the system should offer. Chow et al. [28] annotated FSMs with inputs and the expected outputs. Nayak et al. [34] enriched sequence diagrams with attributes and constraint information derived from class diagrams and Object Constrained Language (OCL) constraints, because in sequence diagram the parameters of messages, i.e. method calls, lacks the constraint and type information. In [13], the nodes of the sequence diagram graph are augmented, and the information needed for augmentation are mined from use case templates, class diagrams, and data dictionary expressed in the form of OCL. This augmentation helped them incorporating oracle information and the data needed for concretizing test scenarios into the test model. Vieira et al. [35] augmented activities in the activity diagram with custom annotations in the form of stereotypes, which represent different information such as the definition and usage of test variables. Test variables are obtained by considering all data objects relevant for corresponding use case diagrams. Their achievements were recognizing feasible and infeasible paths, helping to concretize test scenarios, and reaching the desired functional coverage. Kurth et al. [36] augmented activity diagrams with constraints in the form of OCL and then converted it to an 'A Mathematical Programming Language' (AMPL) program using symbolic execution [2, 37]. The AMPL program is also augmented with an objective function to generate boundary test data. Eshuis et al. [29] considered forks and joins as transitions rather than nodes in the activity diagram and extended it by adding some extra nodes, to be prepared for model transformation. Authors in [18, 38] enhanced the generated intermediate graphs from UML behavioral diagrams by adding weights to edges and nodes, respectively, in order to be able to prioritize test paths. Wang et al. [39] proposed to generate test cases for system testing of timing requirements from timed automata using the information provided by use case

specifications. They augment the set of user-provided timed automata to capture timing requirements by identifying their dependencies with functional scenarios. The augmented automata were issued to UPPAAL model checker for symbolic reachability analysis and generating test cases. Murthy et al. [40] augmented UML statechart diagrams by adding test statements to the state transitions or state nodes. Test statements are incorporated into test scenarios and helped to generate test scripts.

Gebizli et al. [41] proposed to refine system models based on the experience and domain knowledge of test engineers by collecting and analyzing their test execution traces. In this way, the experience gained from exploratory testing is used to improve test models. Note that in their approach, collection and analysis of execution traces are performed automatically, and the model refining phase is performed manually. Also, authors in [42] successively refined Markov usage models by updating transition probabilities to direct test case generation process toward different parts of the test model.

Another advantage of model augmentation is that it enables the testers to incorporate non-functional and quality-of-service test information into models in order to make them ready for generating test cases to test the system from different points of view. For example, Ryser et al. [43] annotated statechart diagrams with performance and timing constraints to allow for performance testing. In [44], the authors extended sequence diagrams to annotate timing constraints for messages. These annotations constraint the time between occurrences of events and specify which events have to happen before the annotated timestamp and which one has to happen at least after the specified time.

IV. Model Transformation

Test models based on their types are usually transformed into an intermediate form to be formalized and prepared for test path or test scenario generation [26]. This transformation is done to resolve the nonstructural problems of test models and to facilitate scenario representation, which results in an easier generation of test sequences with respect to the given coverage criteria. Intermediate forms used in the literature are usually XML [31], table [45-48], tree [20, 49, 50], graph [19, 26], Petri nets [44, 51], other test models [52], different types of automata such as timed automata [53] or descriptive languages like input language of NuSMV [16, 29].

XML is a generic and standard language that can be easily parsed, and many tools have the capability of translating input models into XML. The XML parser extracts necessary information and elements from test models and stores them in an appropriate form to be used for generating subsequent intermediate forms [20, 31, 54] or for later use in the next steps. Also, the XML Metadata Interchange (XMI) which is a standard for exchanging metadata information via XML between UML modeling tools defined by Object Management Group (OMG) is widely used for UML models [9, 21, 45, 55]. Using this standard allows reusing UML test models, which are drawn in different testing tools.

Tables are widely used to store information that is needed for test scenario generation and subsequent model transformations. This information can be the objects and messages presented in the sequence diagram [26], the domain information of input variables [34], or can be the information of activities extracted from activity diagrams [32, 46, 48].

Tree and graph are the most widely used structures as intermediate forms in the literature because of their structural similarities to test models and also their harmony with test scenario generation techniques. Sun et al. [50] proposed to transform activities in activity diagram specifications into an equivalent tree via a set of transformation rules in order to resolve the non-structural problem of activity diagrams. Test cases are then extracted using generated trees; definitely, it is more convenient to generate test scenarios from a normalized tree than a non-structural form. Authors in [49] mapped class and object diagrams into tree structures. The tree structure enabled them to apply the genetic algorithm's tree crossover due to generating new test scenarios. Sarma et al. [13] translated sequence diagrams into a graph structure called sequence diagram graph (SDG). Each node in the SDG is mapped to an interaction between two objects through a message. In other words, SDG nodes represent method calls. This transformation helped them having better augmentation by adding the required test information such as arguments in the methods and expected outputs into SDG nodes. Authors in [25] represent an equivalent wait for graph

for sequence diagram, which helps to identify the deadlock points in concurrent systems by detecting cycles using the Tarjan algorithm. Also, several studies (e.g., [26, 54]) attempt to gather information provided by different test models such as use case, sequence, collaboration, and activity diagrams by integrating them to carry out more effective, in-depth, and broader testing. In order to pave the way for the integration, they first transformed these inherently different input models into standard intermediate graphs or trees.

Petri nets, a well-known mathematical modeling language, are another useful intermediate form with a rich and strict theory and a wide range of available supporting tools. Petri nets have many variations such as colored, hierarchical, event-driven, or timed-arc, which can be used in MBT for test model formalization depending on the application. Petri nets are well-suited for expressing concurrency and parallelism and are well-known for their capabilities to deal with the interaction fragments properties of sequence diagrams [56]. Authors in [57] suggested formalizing sequence diagrams by translating them into colored Petri nets. By selecting this variant, colors can be used to distinguish between object types. Sieverding et al. [44] chose timed-arc Petri nets to formalize sequence diagrams annotated with timing information to better incorporate timing constraints into the model. Authors in [51] proposed formalizing requirement specifications written in Restricted-form of Natural Language (RNL) by automatically translating them into executable Petri net models. Then the reachability tree is extracted and traversed to generate test scenarios.

Timed automaton is a state-based model which can precisely specify timing constraints. Mücke et al. [53] automatically transformed the UML state diagram into timed automata, which is a suitable formal model as input to the UPPAAL model checker. Before translation, the state diagram is flattened to remove hierarchy structures because timed automata lack hierarchy. The generated test cases are used to validate the real-time behavior of state diagrams.

Note that some studies used multiple intermediate forms [31, 52]. For example, Kim et al. [52] proposed to generate test cases based on control-flow and data-flow in UML state diagrams. They first transformed the state diagrams into EFSMs for flattening the hierarchical and concurrent structure of states and also eliminating the broadcast communication. Control-flow in UML state diagrams is identified in terms of paths in the resulting EFSMs. Then to generate test cases based on data-flow, EFSMs were converted into flow graphs in order to be able to apply conventional data-flow analysis techniques.

It should be mentioned that model transformation is a complex and costly operation in which the excessive and misplaced use of intermediate forms may prolong the test case generation process and pose a threat to automation [32]. Also, a sort of traceability and synchronization [58] should exist between the original models and their corresponding intermediate forms, which are mostly neglected in the literature. This traceability enables the changes in the SUT models to be reflected immediately in their corresponding intermediate forms saving a lot of time and resources for regression testing and also for future test case generations.

V. Test Scenario Generation

After preparing test models, it is turn to generate test paths, test sequences, or generally test scenarios. Test scenarios contain a high-level description of input data, and may also include the expected output and the state of the SUT before and after execution of the test. Test scenarios are known as abstract test cases and are later used to extract concrete tests. As an example, a test scenario can be any path from the initial activity state to the final activity state consisting of activities and transitions guards in an activity diagram, or can be any sequence of message paths in a sequence diagram. Notice that the selected coverage criterion decides about the required test scenarios.

Test scenarios can be extracted from test models by traversing intermediate forms in a random manner using random-walk [23], or in a systematic manner using depth-first search (DFS) [19, 31, 34, 46], breadth-first search (BFS) [26, 30], post-order traversal [20], etc. There are also other approaches such as symbolic execution [36], model checking [16, 29], and search-based techniques [17, 18, 59].

With regard to the structure of test models in MBT, using traversal-based approaches are very common, easy to implement, and straightforward. The reason for choosing any of the aforementioned traversing methods is dependent on the application; for example,

Nebut et al. [30] chose breadth-first traversal of their proposed use case transition system in order to obtain small test objectives, i.e. sequences of use cases. This ensures that the size of the computed paths is minimal, with this motivation that small tests are more meaningful and understandable by humans than large ones. Samuel et al. [20] carried out post-order traversal of the tree representation of communication diagram (Communication Tree) for selecting conditional predicates, to reduce the number of execution steps in the function minimization process.

Symbolic execution is a promising technique for generating high coverage test suites and for finding deep errors in complex software systems [2, 37]. It explores SUT paths symbolically using symbolic values for variables and generates a path constraint for each path by accumulating predicates in branching nodes and updating its state in assignment statements .The generated path constraints are fed to a constraint solver to decide about their feasibility and generating concrete inputs. Symbolic execution is generally used in white box testing utilizing the program source code but is infiltrated to the MBT too. For example, the approach proposed by Kurth et al. [36] symbolically executes control-flow paths in the activity diagram and encodes them as AMPL programs. They used AMPL to formalize the execution of control flow paths in activity diagrams. A constraint solver is then used to generate concrete test cases. They also performed early infeasible path elimination by querying constraint solver at branching nodes.

Model checking is a well-known technique for verifying models and detecting errors in behavioral designs. In model checking, a desired set of formal properties of the system are specified by the user, expressed as temporal logic formulas, and is given to a model checker along with formalized test model. The properties can be test purposes or coverage criteria which are formulated as reachability formulas. The negated version of the properties is applied to the formal model using model checking to generate traces. These traces are then translated to generate test cases. Note that the generated test cases by model checking can be used to verify the implementation as well as verifying the specifications [16].

Search-based techniques are widely used in MBT, which apply metaheuristic search algorithms such as genetic algorithm (GA) [17, 18, 49], memetic algorithm [59], ant colony optimization [38], etc. to different model-based testing problems. These techniques can be used either directly [17, 18, 59] or indirectly [49] for generating and prioritizing test scenarios. Kalaji et al. [17] proposed generating test sequences from extended finite state machines (EFSMs) using genetic algorithm by defining a suitable fitness function that guides the search toward likely feasible transition paths. Their fitness function employed dataflow dependencies among transition guards of paths. It should be noted that in these techniques, solution encoding and defining proper fitness function is so crucial, and the success, efficiency, and convergence of the algorithm are directly dependent on it.

VI. Concrete Test Case Generation

Since test scenarios contain an abstract representation of test cases and are mostly incomplete, generally they cannot be directly used as executable test cases. So, based on the type and the complexity of test models, some works such as handling abstract information, concretizing input variables, and solving path constraints must be done manually [9, 10, 21, 32] or automatically [30, 34] on the test scenarios to make them concrete and executable. For automatically concretizing test scenarios, input models should be well-specified using detailed and additional information.

In order to concretize test scenarios, the first step is to replace abstract information. As an example of the presence of abstract information in the test scenarios, consider a simplified test scenario T for an Automated Teller Machine (ATM):

$$T\colon GetPin(PIN)[ValidPIN]GetAmount(amount)$$

$$[ValidAmount]DispenseCash(\ \ )$$

For the sake of simplicity, we omit some details such as message prompts to the user and also operations on the banking side. As can be seen, in this test scenario it is not known what an invalid PIN is and also the valid value for $amount$ is not specified. That is why we call it an abstract test case. If instead of $ValidAmount$ we had $Amount > 5000 \wedge Amount < Balance \wedge Amount\ mod\ 50 = 0$, it would be easier to concretize the $Amount$ variable automatically; otherwise, the tester must carefully read the specifications to exploit the validity conditions of Amount. The same is true

about $ValidPIN$ which may follow some domain rules and patterns.

After replacing abstract information, we are generally faced with some input variables and path constraints. Input variables can be method parameters, instance variables that set the states of classes, parameters of events and actions in the UML state diagram, etc. Note that there is zero- or one-to-many relation between test scenarios and concrete test cases, which means that for each test scenario, there can be more than one test case. Sometimes with regards to the coverage criteria, there is a need for more than one concretization for a test scenario, i.e., a path may need to be tested several times with different concrete data. For example, full predicate coverage requires that each clause in each predicate on each guard condition is tested independently [22] which means that all combinations of truth values of clauses should be concretized.

Concretizing test scenarios is mainly done by utilizing pre-defined information incorporated into test models [13, 43], constraint solving [17, 34, 36], search-based techniques like GA [17], category-partition method [10, 15, 32, 47], random [9], and function minimization technique [19, 20].

Annotating test models and adding suitable test information can help to concretize test scenarios. For example, in T, by choosing purely random values for PIN, it may be unlikely to obtain a valid value. But data annotations can specify its form and pattern. Also, data annotations can determine the type and the range of valid values for input variables. For example, authors in [55] used labels with textual stereotypes representing data flow for the edges of the activity diagram. These stereotypes contain the name and the type of variables. Also, some approaches utilize OCL constraints to specify the information needed for concretization (e.g., [13]).

There are many off-the-shelf constraint solvers that can be used to solve path constraints, but the existence of nonlinear arithmetic and floating-point computations makes a path constraint complex, and this may lead to difficulty in translating the constraints into the theory of the underlying solver. In other words, the efficiency and usability of this approach are based heavily on the theory supported by the constraint solver and its constraint solving abilities.

Search-based techniques can be used to extract input data by converting path predicates into fitness function. Since these techniques require the calculation of fitness functions in a large number of times, they should be computationally as simple as possible and effective in order to guarantee the efficiency of the approach.

It should be noted that, performing boundary value analysis along with these techniques increase the quality of test set. Domain boundaries are particularly fault prone [36], so choosing boundary test cases to execute a control-flow path are more valuable than arbitrary test cases. In this regard, some studies used boundary testing [9, 19, 20] to generate concrete test cases to ensure that boundaries are tested adequately.

Conclusion

Model-based testing has some limitations and challenges that are mostly related to the incompleteness, high level of abstraction, complexity, and also the informal nature of input models. In fact, in MBT, the quality of input models has a direct impact on the quality of the test suite, and subsequently, on the effectiveness of the whole testing process. In the literature, there are different studies addressing these problems.

In this paper, we represent a framework for model-based test case generation and show that MBT approaches have three main steps: preprocessing, test scenario generation, and concrete test case generation. We represent the necessity of each step and summarized the proposed methods. We also categorized different coverage criteria used in MBT.

References

1. Anand, S., et al., An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013. 86(8): p. 1978-2001.

2. Sabbaghi, A. and M.R. Keyvanpour, A Systematic Review of Search Strategies in Dynamic Symbolic Execution. Computer Standards & Interfaces, 2020: p. 103444.

3. Bidgoli, A.M. and H. Haghighi, Augmenting ant colony optimization with adaptive random testing to

cover prime paths. Journal of Systems and Software, 2020. 161: p. 110495.

4. Sabbaghi, A. and M. Keyvanpour, A novel approach for combinatorial test case generation using multi objective optimization, in Computer and Knowledge Engineering (ICCKE), 2017 7th International Conference on. 2017, IEEE.

5. Jatana, N. and B. Suri, An improved crow search algorithm for test data generation using search-based mutation testing. Neural Processing Letters, 2020. 52(1): p. 767-784.

6. Sabbaghi, A. and M.R. Keyvanpour. State-based models in model-based testing: A systematic review. in Knowledge-Based Engineering and Innovation (KBEI), 2017 IEEE 4th International Conference on. 2017. IEEE.

7. Gerhold, M. and M. Stoelinga, Model-based testing of probabilistic systems. Formal aspects of computing, 2018. 30(1): p. 77-106.

8. Gudmundsson, V., et al., Model-based Testing of Mobile Systems--An Empirical Study on QuizUp Android App. arXiv preprint arXiv:1606.00503, 2016.

9. Sun, C.a., et al., A transformation-based approach to testing concurrent programs using UML activity diagrams. Software: Practice and Experience, 2016. 46(4): p. 551-576.

10. Nayak, A. and D. Samanta, Synthesis of test scenarios using UML activity diagrams. Software & Systems Modeling, 2011. 10(1): p. 63-89.

11. Poncelet, C. and F. Jacquemard, Model-based testing for building reliable realtime interactive music systems. Science of Computer Programming, 2016. 132: p. 143-172.

12. Damiani, F., et al., A novel model-based testing approach for software product lines. Software & Systems Modeling, 2017. 16(4): p. 1223-1251.

13. Sarma, M., D. Kundu, and R. Mall. Automatic test case generation from UML sequence diagram. in 15th International Conference on Advanced Computing and Communications (ADCOM 2007). 2007. IEEE.

14. Cartaxo, E.G., F.G. Neto, and P.D. Machado. Test case generation by means of UML sequence diagrams and labeled transition systems. in 2007 IEEE International Conference on Systems, Man and Cybernetics. 2007. IEEE.

15. Andrews, A., et al., Test adequacy criteria for UML design models. Software Testing, Verification and Reliability, 2003. 13(2): p. 95-127.

16. Chen, M., P. Mishra, and D. Kalita. Coverage-driven automatic test generation for UML activity diagrams. in Proceedings of the 18th ACM Great Lakes symposium on VLSI. 2008.

17. Kalaji, A.S., R.M. Hierons, and S. Swift, An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. Information and Software Technology, 2011. 53(12): p. 1297-1318.

18. Sharma, C., S. Sabharwal, and R. Sibal, Applying genetic algorithm for prioritization of test case scenarios derived from UML diagrams. arXiv preprint arXiv:1410.4838, 2014.

19. Swain, R., et al., Automatic test case generation from UML state chart diagram. International Journal of Computer Applications, 2012. 42(7): p. 26-36.

20. Samuel, P., R. Mall, and P. Kanth, Automatic test case generation from UML communication diagrams. Information and software technology, 2007. 49(2): p. 158-171.

21. Ali, S., et al., A state-based approach to integration testing based on UML models. Information and Software Technology, 2007. 49(11): p. 1087-1106.

22. Offutt, J., et al., Generating test data from state-based specifications. Software testing, verification and reliability, 2003. 13(1): p. 25-53.

23. Prowell, S.J. Using markov chain usage models to test complex systems. in System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on. 2005. IEEE.

24. Prasanna, M., K. Chandran, and D.B. Suberi, Automatic test case generation for UML class diagram using data flow approach. Academia. Education, 2011.

25. Mallick, A., N. Panda, and A.A. Acharya, Generation of test cases from uml sequence diagram

and detecting deadlocks using loop detection algorithm. International Journal of Computer Science and Engineering, 2014. 2: p. 199-203.

26. Sumalatha, V.M. and G. Raju, Uml based automated test case generation technique using activity-sequence diagram. International Journal of Computer Science Applications, 2012. 1(9).

27. Pilskalns, O., et al., Testing UML designs. Information and Software Technology, 2007. 49(8): p. 892-912.

28. Chow, T.S., Testing software design modeled by finite-state machines. IEEE transactions on software engineering, 1978(3): p. 178-187.

29. Eshuis, R., Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering and Methodology (TOSEM), 2006. 15(1): p. 1-38.

30. Nebut, C., et al., Automatic test generation: A use case driven approach. IEEE Transactions on Software Engineering, 2006. 32(3): p. 140-155.

31. Boghdady, P.N., et al. An enhanced test case generation technique based on activity diagrams. in The 2011 International Conference on Computer Engineering & Systems. 2011. IEEE.

32. Linzhang, W., et al. Generating test cases from UML activity diagram based on gray-box method. in 11th Asia-Pacific software engineering conference. 2004. IEEE.

33. Sabbaghi, A., M.R. Keyvanpour, and S. Parsa, FCCI: A fuzzy expert system for identifying coincidental correct test cases. Journal of Systems and Software, 2020: p. 110635.

34. Nayak, A. and D. Samanta, Automatic test data synthesis using uml sequence diagrams. Journal of Object Technology, 2010. 9(2): p. 75-104.

35. Vieira, M., et al. Automation of GUI testing using a model-driven approach. in Proceedings of the 2006 international workshop on Automation of software test. 2006.

36. Kurth, F., S. Schupp, and S. Weißleder. Generating test data from a UML activity using the AMPL interface for constraint solvers. in International Conference on Tests and Proofs. 2014. Springer.

37. Sabbaghi, A., H.R. Kanan, and M.R. Keyvanpour, FSCT: A new fuzzy search strategy in concolic testing. Information and Software Technology, 2019. 107: p. 137-158.

38. Elghondakly, R., S. Moussa, and N. Badr, An optimized approach for automated test case generation and validation for UML diagrams. Asian J Inf Technol, 2016. 15(21): p. 4276-4290.

39. Wang, C., F. Pastore, and L. Briand. System Testing of Timing Requirements based on Use Cases and Timed Automata. in Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on. 2017. IEEE.

40. Murthy, P., et al. Test ready UML statechart models. in Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools. 2006.

41. Gebizli, C.S. and H. Sözer. Improving models for model-based testing based on exploratory testing. in 2014 IEEE 38th International Computer Software and Applications Conference Workshops. 2014. IEEE.

42. Gebizli, C.S., H. Sözer, and A.Ö. Ercan. Successive refinement of models for model-based testing to increase system test effectiveness. in Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on. 2016. IEEE.

43. Ryser, J. and M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. in Proc. 12th International Conference on Software and Systems Engineering and their Applications. 1999.

44. Sieverding, S., C. Ellen, and P. Battram, Sequence diagram test case specification and virtual integration analysis using timed-arc Petri nets. arXiv preprint arXiv:1302.5170, 2013.

45. Briand, L. and Y. Labiche, A UML-based approach to system testing. Software and systems modeling, 2002. 1(1): p. 10-42.

46. Jena, A.K., S.K. Swain, and D.P. Mohapatra. A novel approach for test case generation from UML activity diagram. in 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT). 2014. IEEE.

47. Boghdady, P., et al. An enhanced technique for generating hybrid coverage test cases using activity diagrams. in 2012 8th International Conference on Informatics and Systems (INFOS). 2012. IEEE.

48. Pechtanun, K. and S. Kansomkeat. Generation test case from UML activity diagram based on AC grammar. in 2012 International Conference on Computer & Information Science (ICCIS). 2012. IEEE.

49. Prasanna, M. and K. Chandran, Automatic test case generation for UML object diagrams using genetic algorithm. Int. J. Advance. Soft Comput. Appl, 2009. 1(1): p. 19-32.

50. Sun, C.-a. A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications. in 2008 32nd Annual IEEE International Computer Software and Applications Conference. 2008. IEEE.

51. Sarmiento, E., et al., Test Scenario Generation from Natural Language Requirements Descriptions based on Petri-Nets. Electronic Notes in Theoretical Computer Science, 2016. 329: p. 123-148.

52. Kim, Y.G., et al., Test cases generation from UML state diagrams. IEE Proceedings-Software, 1999. 146(4): p. 187-192.

53. Mücke, T. and M. Huhn. Generation of optimized testsuites for UML statecharts with time. in IFIP international conference on testing of communicating systems. 2004. Springer.

54. Sarma, M. and R. Mall. Automatic test case generation from UML models. in 10th International Conference on Information Technology (ICIT 2007). 2007. IEEE.

55. Teixeira, F.A.D. and G.B. e Silva, EasyTest: An approach for automatic test cases generation from UML Activity Diagrams, in Information Technology-New Generations. 2018, Springer. p. 411-417.

56. Bouabana-Tebibel, T. and S.H. Rubin, An interleaving semantics for UML 2 interactions using Petri nets. Information Sciences, 2013. 232: p. 276-293.

57. Bowles, J. and D. Meedeniya. Formal transformation from sequence diagrams to coloured petri nets. in 2010 Asia Pacific Software Engineering Conference. 2010. IEEE.

58. Ding, Z., M. Jiang, and M. Zhou, Generating petri net-based behavioral models from textual use cases and application in railway networks. IEEE Transactions on Intelligent Transportation Systems, 2016. 17(12): p. 3330-3343.

59. Nejad, F.M., R. Akbari, and M.M. Dejam. Using memetic algorithms for test case prioritization in model based software testing. in 2016 1st Conference on Swarm Intelligence and Evolutionary Computation (CSIEC). 2016. IEEE.