CSIT Conference 2023, Yerevan, Armenia, September 25 - 30

Automated Model-Based Test Case Generation for UML Activity Diagrams using EFSM

Nazanin Shahbazi
Data Mining Laboratory, Department of Computer Engineering
Alzahra University
Tehran, Iran
e-mail: N.shahbazi@student.alzahra.ac.ir

Mohammad-Reza Keyvanpour
Department of Computer Engineering, Faculty of Engineering
Alzahra University
Tehran, Iran
e-mail: keyvanpour@alzahra.ac.ir

***Abstract*— Model-based testing (MBT) is a promising approach for generating test cases from system models, providing high levels of automation and effectiveness. The potential for automation in MBT is possible if the model is formal and machine-readable. A commonly employed formal modeling technique is the representation of a system as an extended finite state machine (EFSM). However, in practice, formal models are not common in the industry. Activity diagrams, on the other hand, are well-suited for generating test cases, but their lack of formal semantics can lead to ambiguous interpretations and make them unsuitable for automation. In this paper, we propose an efficient approach that maps UML Activity Diagrams into EFSMs, providing a formal modeling of the system under test (SUT) and utilizes JUnit and ModelJUnit Java libraries to automatically generate test cases, using coverage measures to evaluate them. Our approach aims to overcome the limitations of automation in MBT, while achieving efficient coverage and execution time metrics.**

***Keywords*—Model-Based Testing; Automatic Test Case generation; Activity Diagram; Extended Finite State Machine; ModelJUnit; JUnit**

I. INTRODUCTION

Software engineering is a field that employs a methodical and structured approach to the development, operation, and maintenance of software systems [1]. The goal of software organizations is to develop and deliver software within specified timeframes and budgets [2]. However, software development, particularly for complex systems, is prone to human errors, necessitating the assessment of software for errors during source code generation [3]. Software testing plays a crucial role in identifying faults and is a vital but costly phase in software development and maintenance. One of the most challenging aspects of testing is generating test cases, which is essential to ensure the success of the testing phase [4].

Modeling techniques can enhance software testing productivity when used in this context [5]. Model-based test case generation (MB-TCG) is a methodology that involves the generation of tests using system models, which are based on model-based testing (MBT) [6]. Using MBT allows testing to begin early in the software development process, running tests without requiring access to the source code of the system under test, owing to its black-box nature. This technique uses system models to generate and execute test cases automatically, reducing testing time and effort [7].

Unified Modeling Language (UML) is a popular option for software modeling as it offers a high level of expressiveness [8]. Activity Diagram is the most used UML diagram in model-based test case generation [6]. UML Activity Diagram is a semi-formal specification that can be used to describe the workflow of the system and captures critical system behaviors [9]. Although this Diagram is widely used in this field, there is a lack of automated techniques for test generation from UML activity diagrams [10]. In addition to the mentioned problem, due to the lack of formal semantics, the use of UML diagrams can lead to inconsistency, transformation problems, and different interpretations [5]. Another issue of note is that UML specification presents a challenge in terms of navigation and comprehension [11].

Using formal models is one approach to mitigate these issues, as they offer precise semantics for representing system behavior [8]. Finite State Machines (FSMs) are a favorite notation in formal system modeling and testing software [5]. Although, formal models are rarely used in practice, probably because developers lack the necessary training and familiarity with the mathematical notation [11].

Formal models are highly recommended in Model-Based Testing (MBT) due to their ability to automate the testing process, resulting in improved efficiency and effectiveness [5]. Extended finite state machine (EFSM) is one of the formal models that have received significant attention and extensive study over the past few decades [12]. An EFSM model is an enhanced model based on FSM. This model can represent many complex systems containing both control and data parts [13].

In the context of this paper, we propose an efficient and systematic approach that maps UML Activity Diagrams into EFSM following some transformation rules. By using JUnit and ModelJUnit Java libraries, test cases are automatically generated from EFSM. The main contribution of this paper is the definition of transformation rules to map the various elements of the UML Activity Diagram into constructions of the Extended Finite State Machine (EFSM). Furthermore, to tackle the issue of complexity and understandability of the UML specification problem, we utilize an updated version of the metamodel for UML Activity Diagrams, specifically tailored for simplicity and applicability while ensuring a close alignment with the EFSM metamodel used in our research.

The rest of this paper is organized as follows: Section II describes some related research. Afterwards, Section III presents UML Activity Diagram and Extended Finite State Machine. Section IV describes our approach and its implementation. In Section V, the results obtained by implementing the proposed approach on a sample scenario have been discussed. Finally, the conclusion and future work are placed in Section VI.

II. Related Work

Numerous research studies have been conducted on the subject of generating test cases from UML Activity Diagrams. One of the most relevant related works in the area of automated test case generation from UML models is the article by Smith et al. [5]. They proposed an approach for generating test cases from UML sequence diagrams using Extended Finite State Machines (EFSMs). Their approach showed promising results, but one of its weaknesses was that it only worked for sequence diagrams. We built our work upon the approach proposed by this article, but we extended it to work with UML activity diagrams (AD) since ADs are more commonly used in software development, as shown in a recent survey [6]. Our approach can generate test cases that cover a wider range of scenarios since Activity diagrams can model more complex behavior and allow for more varied interactions between objects than sequence diagrams [14].

In [15], the authors propose an approach for generating scenario-based test cases from UML Activity Diagrams. The proposed approach utilizes an intermediate model named Extended Activity Dependency Graph (EADG), which extends activity graphs to generate test scenarios. However, this approach differs from ours in that it does not utilize model-driven engineering (MDE) concepts or formal models for test generation.

In [16], the authors propose a novel method for generating test cases using UML Activity and Sequence Diagrams. Their approach involves the conversion of Sequence Diagram into a graph, which refers to Sequence Graph, and transforming the Activity Diagram into the Activity Graph. Test suite generation is achieved by merging the graphs into a single software graph. While this methodology shares similarities with ours, it diverges by not employing MDE concepts or formal models for test generation. Moreover, the approach necessitates the manual creation of both Activity and Sequence Diagrams, which can make it difficult to use the tool for more complex software systems.

III. Background

A. Activity Diagram

Using an Activity Diagram in UML allows us to model the dynamic characteristics of systems [17]. Use cases or business processes can also be described with an activity diagram to show how activities flow sequentially and to show logic [16].

An activity diagram is formally defined as a six-tuple $D = (A, T, F, C, a_I, a_F)$, where A is a finite set of $A \subseteq \mathbb{P}(S)$ representing activity states, T is a finite set of $T \subseteq \mathbb{P}(Y)$ denoting completion transitions. $F \subseteq \{A \times T\} \cup \{T \times A\}$ signifies the flow relation connecting activities and transitions. C is a finite set $C \subseteq G(T)$ representing guard conditions, and c_i is in correspondence with t_i, and $Cond(t_i) = c_i$. $a_I \in A$ is the initial state, and $a_F \in A$ is the final state. There exists only one transition $t \in T$ such that $(a_I, t) \in F$, and for any $t' \in T$, $(t', aI) \notin F$ and $(aF, t') \notin F$ [9].

In this paper, we use some of the most common elements of an Activity Diagram, which are as follows:

- Initial Node: An InitialNode is a control node that marks the beginning of a process or activity flow.
- Final Node: A FinalNode is a control node that marks the completion of an activity or process flow.
- Executable Activity Node: ExecutableNode is a type of action node that can contain executable behavior.
- Merge Node: A MergeNode is a control node that merges multiple incoming control flows, allowing for their convergence without any synchronization.
- Decision Node: A DecisionNode is a control node that represents a decision point, where one of several outgoing control flows is chosen based on a condition or criteria evaluation.
- Input and Output Pins: Input and Output Pins represent the transfer of data or objects between activity nodes, where an Input Pin accepts values or objects as input to an activity node, and an OutputPin produces values or objects as output from an activity node.
- Send and Receive Signal Actions: A Signal represents inter-object communication without the need for a reply, initiating an asynchronous reaction in the receiver [8].

Other elements of Activity Diagrams defined by UML 2.5 are not in the scope of this article.

B. Extended Finite State Machine

The Extended Finite State Machine (EFSM) is a well-known formal specification technique that is commonly employed to define the various states and actions of a software system. This method is widely used to describe the behavior of software systems in a precise and unambiguous manner [18].

An EFSM can be formally represented by a 6-tuple (s_0, S, V, I, O, T) where S is a finite set of states with initial state s_0; V is a finite set of context variables; I is a set of transition inputs; O is a set of transition outputs; and T is a finite set of transitions.

Each transition $tx \in T$ can also be represented formally by a tuple $tx = (s_i, s_j, P_{tx}, A_{tx}, i_{tx}, o_{tx})$, where s_i, s_j are the origin and target states of transition tx, and $i_{tx} \in I$ represents the input parameters of the beginning of the transition tx, such as events that can be interpreted as special types of input parameters, and $o_{tx} \in O$ denotes the output results at the end of the transition tx. P_{tx} represents the predicate conditions (guards) with their respective context variables, and A_{tx} denotes the operations (actions) with their respective current variables. EFSM models can be represented as a directed graph G(V, E). The elements of V represent the states of an EFSM, and E denotes its transitions [11].

IV. Proposed Approach

This section presents an automated approach for generating test cases from UML activity diagrams using EFSM. The methodology involves several steps that begin with translating the UML activity diagram into a formal EFSM model using Atlas Transformation Language. The generated EFSM model is used to automate the test generation process, where test cases are generated based on EFSM-based methods. These methods provide coverage for different paths and states within the system. ModelJUnit and JUnit libraries are used to facilitate the generation of executable test cases. Acceleo is then used to perform a Model-To-Text (M2T) transformation, converting the generated test cases from a model representation into a textual representation that can be executed.

A. *Metamodeling Constructs*

In order to establish a clear and structured foundation for our research, we defined two metamodels: the UML Activity Diagram metamodel, which serves as a source for our model transformation process, and the Extended Finite State Machine metamodel, which represents the target model. To implement these metamodels, we utilized the Eclipse Modeling Framework (EMF) and represented them in Ecore.

The official UML specification [8] can be complex to navigate and hard to understand. Therefore, this metamodel has been heavily criticized, and the use of simplified metamodels is prevalent in most of the literature on this subject [11]. To address this issue, a simplified metamodel for the Activity Diagram is proposed in this study, depicted in Fig. 1, in comparison to the metamodel specified by the OMG.

The proposed metamodel contains 19 metaclasses and eliminates constructs that are not frequently used in practice. By streamlining the metamodel, it is easier to understand and apply in practice, providing a more practical and straightforward approach to Activity Diagrams.

As shown in Fig. 2, the metamodel used for EFSM is comprised of six metaclasses, with EFSM serving as an abstraction of an Extended Finite State Machine. Within the EFSM entity, there are states, transitions, and context variables [5].

B. *Metamodels Transformation Principles*

A detailed description of the rules for transforming an Activity Diagram into an Extended Finite State Machine is provided in this section. The defined transformation rules are listed below:

- RInitialNode: For the node of type InititalNode, first an EFSM is created with the same name of the AD, then initial state S0 is added to it. The initial state is used to update both the previous state and the current state.

- RDecisionNode: RDecisionNode rule operates in two stages, depending on the type of decision node being processed. For each conditional decision node, this rule creates one new state and two new transitions. The transition with a guard of "true" leads to the newly created state, while the transition with a guard of "false" returns to the previous state. For switch decision nodes, this rule creates a new state and transition for each guard. Regardless of the type of decision node, each transition has output, guard, and action labeled with the guard of the specific control flow, and any output pin from the node before the decision node is treated as the transition's input. After creating each new state, this rule updates both the previous and current states.

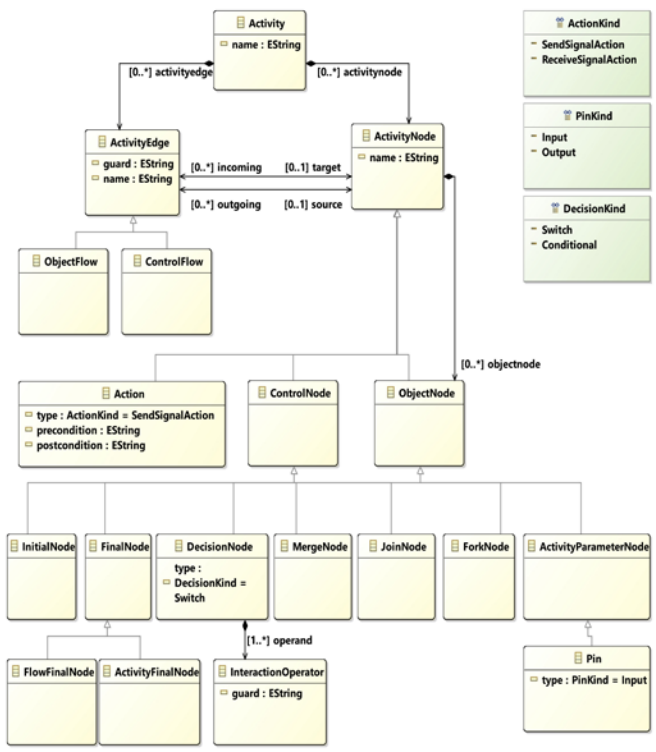

Fig. 1. Activty Diagram Metamodel

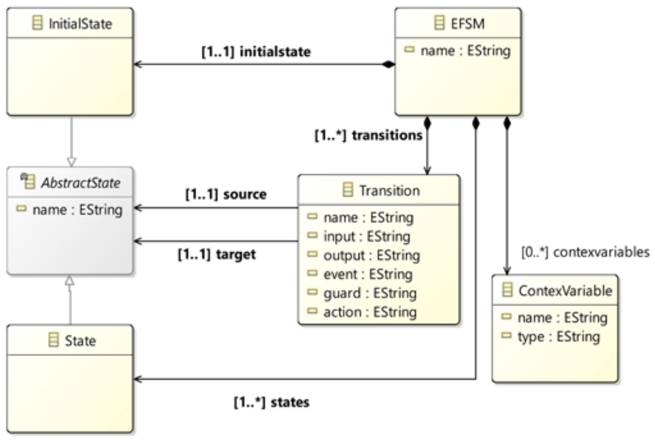

Fig. 2. Extended Finite State Machine Metamodel [5]

- RMergeNode: This rule handles merge nodes in the Activity Diagram (AD) by creating a new transition for each node that has a control flow leading to the merge node. These transitions do not have any output, guard, or action, but any output pin of the nodes connected to the merge node is considered to be a corresponding transition's input. The transitions all lead to the state that was already created for the node after the merge node using the "RActivityNode" rule. After creating these transitions, both the previous and current states are updated.

- RSignalActivityNode: For nodes of type ExecutableActivityNode, SendSignalActionNode, and ReceiveSignalActionNode, this rule creates a new state and transition. The new state is connected to the previous state using the transition, and all pins of the corresponding node in the AD are considered as inputs for the transition. Additionally, this rule checks all control flows of a node to see if any flow goes backward without any conditions. If such flows exist, a new transition is created to connect the corresponding states to each other, and the inputs of these transitions are labeled as previously explained. After creating these new states and transitions, the previous and current states are updated as usual.

Our implementation of the transformation rules relied on Atlas Transformation Language (ATL).

In this study, we are utilizing a slightly modified version of some of the lazy rules proposed in [5], in combination with our matched rules. Specifically, the following lazy rules are employed:

- LrInitialState: The initial state S0 is created, followed by incrementing the state order. Both the previous state and the current state are updated to the newly created initial state. Additionally, the name of the Activity Diagram is stored in a variable.
- LrState: A new state is generated, the state order is incremented, the previous state is updated to the current state, and the current state is changed to the newly created state.
- LrTransition: A transition is established, connecting the previous state to the current state. The transition's input is labeled with the input/output pin of the Activity Diagram. The output, guard, and action can be null and depend on the type of the activity node.

In our study, we used an automatic four-step approach for generating test cases. This approach involves implementing the EFSM model interface, implementing the adapter, generating the test cases, and then concretizing them. These four steps were automatically generated using Acceleo.

V. Experiment

This section presents an overview of our proposed approach, as well as the results and a discussion of the experiment.

A. *Case Study*

This section describes a case study demonstrating the practical application of our approach. The UML Activity Diagram in Fig. 3, depicts an ATM (Automatic Teller Machine) withdrawal process. Initially, we created an Activity Diagram model that described the behavior of the system, using the Activity Diagram editor implemented in the EMF. To transform this Activity Diagram model into an executable model, we utilized the transformation rules implemented in ATL.

After the execution of the transformation rules, we obtained an EFSM model that reflected the behavior of the system. Fig. 4 illustrates the resulting EFSM model that was generated. We created this model using the Visual Paradigm Drawing Tool by utilizing the XML output file generated from executing the transformation rules on the ATM Activity Diagram.

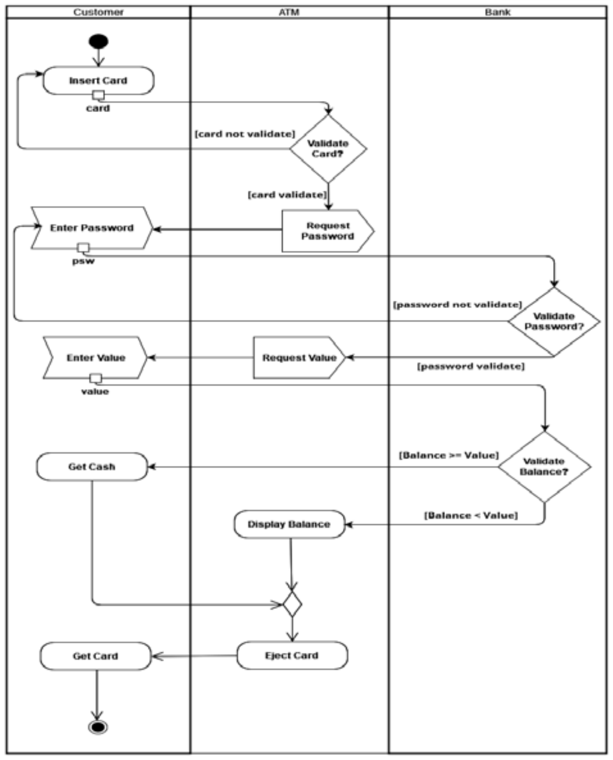

Fig. 3. ATM Activity Diagram

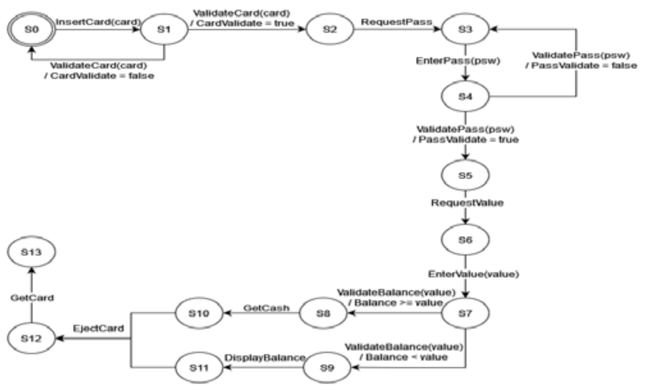
S0
InsertCard(card)
S1
ValidateCard(card) / CardValidate = true
S2
RequestPass
S3
ValidateCard(card) / CardValidate = false
EnterPass(psw)
ValidatePass(psw) / PassValidate = false
S4
ValidatePass(psw) / PassValidate = true
S5
RequestValue
S6
EnterValue(value)
S13
GetCard
ValidateBalance(value) / Balance >= value
S10
GetCash
S8
S7
S12
EjectCard
ValidateBalance(value) / Balance < value
S11
DisplayBalance
S9

Fig. 4. ATM EFSM Model

Afterwards, we followed the 4-step automatic approach, as previously described, to generate test cases using classes AtmModel, AtmAdapter, AtmTest, and AtmJUnit. These classes were obtained through the implementation of the four steps in the approach. Finally, our approach was put to the test through a series of test cases designed to assess its effectiveness. To execute the tests, we utilized the Eclipse Modeling Framework (EMF) in conjunction with the ATMTest and ATMJUnit classes.

Reflecting the importance of coverage criteria to test engineers when creating the test cases [19], we used three distinct coverage types to evaluate the application of our approach. A test model can be represented by an abstract regular set $P \subseteq \Sigma^a$, where Σ is an alphabet denoting possible actions, and $a \in N \cup \{*\}$ indicating that each test can have any length. Considering the fact that P may be huge or even infinite, including 'a' in $N \cup \{*\}$ means that tests can vary in length within specific boundaries [20]. As a result, coverage criteria can be outlined as in Definition 1.

Definition 1 (coverage)**.** A set of tests $S \subseteq P$ is said to cover a test-model $P \subseteq \Sigma^a$ under the coverage criteria $C = \{C(i)\}_{i \in I}$, $C(i) \subseteq \Sigma^a$ if $\forall i \in I : (C(i) \cap P \neq \emptyset \Rightarrow C(i) \cap S \neq \emptyset)$ and $\cup_{i \in I}\ C(i) = \Sigma^a$, where the index set I is used to name the coverage sets and the coverage sets C(i) are used to identify the aspects that are covered by the tests [20].

During testing, we tracked state, action, and transition coverage metrics for each individual test case, as presented in Table I. The state coverage metric indicates the number of states visited, and its formula can be expressed as (1), where |SV| is the number of states visited during testing and |TS| is the number of total states.

$$\text{state coverage} = (|SV|/|TS|) \times 100 \qquad (1)$$

The action coverage metric reflects the number of actions performed during testing. The action coverage formula is given by (2), where |AP| is the number of actions performed during testing and |TA| is the number of total actions.

$$\text{action coverage} = (|AP|/|TA|) \times 100 \qquad (2)$$

The transition coverage metric measures the number of transitions that were visited. The transition coverage formula is shown in (3), where |TV| is the number of states triggered during testing and |TT| is the number of total transitions.

$$\text{transition coverage} = (|TV|/|TT|) \times 100 \qquad (3)$$

Prior to initiating the tests, we established initial values for the card attribute (111), psw attribute (123), and balance attribute (100.00) belonging to the Bank class of the SUT.

B. *Obtained Results*

The generated test cases and the results obtained from their execution are all reported in Table I. We have also included a visually informative bar chart, shown in Fig. 5 to illustrate the metrics associated with each test case.

Fig. 5 displays the coverage metrics for each test case generated using our approach. It is important to note that the possible number of states, actions, and transitions that can be visited may vary depending on the specific test case and its associated parameters.

In particular, for the initial set of generated test cases, the total number of states in the ATM system is determined to be 14 based on the EFSM representation in Fig. 2. However, since the value is not generated within the scope of the test cases T1, T2, and T3, the actual number of states that can be visited is reduced. According to our EFSM analysis, only 5 states are possible to be visited, given the absence of the value.

Similar considerations apply to the transition coverage. For action coverage, all possible actions are performed by executing each test case. Note that the same explanation applies to all the other generated test cases, where the coverage metrics should be interpreted within the context of each individual test case.

C. *Discussion*

Our approach's novelty lies in its ability to automate the entire testing process, with the exception of manually defined stubs. To the best of our knowledge, the level of automation we have achieved with ADs has not been reported before, making our approach a valuable contribution to the field. To further illustrate the superior automation achieved by our approach, we conducted a comprehensive comparison with three of the most automated and recent works in the field, as shown in Table II.

TABLE I. GENERATED TEST CASES

Test case	***card***	***psw***	***value***	***State***	***Action***	***Trans.***	***Time (ms)***
T1	111	123		5/14	12/12	60/144	6.0
T2	111	246		5/14	12/12	60/144	13.6
T3	222	246		2/14	12/12	24/144	7.2
T4	111	123	50	12/14	12/12	144/144	16.7
T5	111	123	200	12/14	12/12	144/144	23.5
Total	_	_	_	14/14	12/12	144/144	_

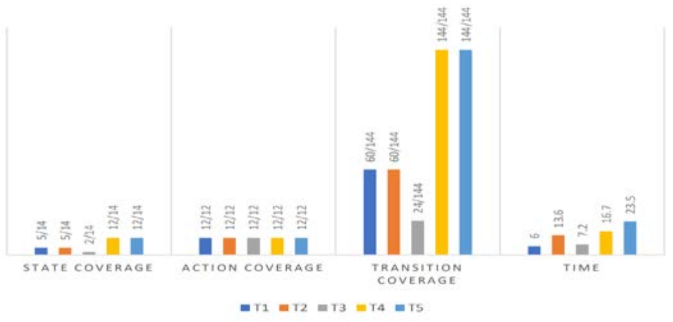

Fig. 5. The state, action, and transition coverage, along with the execution time (in milliseconds), for each generated test case

TABLE II. COMPARISON OF THE LEVEL OF AUTOMATION

Approach	***Level of Automation***	***Description***
M. Rocha et al. [11]	Very High	Achieves superior automation, but not applicable to ADs.
A. Hettab et al. [15]	Moderate	Test data can only be generated manually from the automatically generated test scenarios.
Meiliana, I. et al [16]	Fairly High	Testing itself is automated, but the approach necessitates the manual creation of 2 UML diagrams.
Our approach	High	Automates the entire testing process except for stubs.

Direct comparison between works with different levels of automation would not be valid [21]. Given that no other work has reported achieving the same level of automation with ADs, our proposed approach and the results obtained can only be discussed in a more general sense. As outlined in Table I, by executing all the generated test cases, we achieve complete coverage of all EFSM actions, states, and transitions. Furthermore, the efficiency of our approach is demonstrated by the fact that it generates test cases in less than 1 second.

VI. Conclusion and Future Work

In this paper, we proposed an automated model-based test case generation approach using Activity Diagrams. To formalize the activity diagrams, we proposed a simplified metamodel that was used to map ADs to an EFSM representation using some transformation rules. We then applied a four-step approach to generate test cases, using JUnit and ModelJUnit to execute the tests. Our approach was able to achieve good coverage and time results, demonstrating the effectiveness of our methodology.

As future work, other UML diagrams, such as state and class diagrams, can be incorporated into the proposed test generation process. Moreover, advanced testing techniques, such as mutation testing, can be used to enhance the generated test suite. Finally, our approach can be applied to a real-world software system to further evaluate its effectiveness and scalability.

References

[1] F. Morsali and M. R. Keyvanpour, “Search-based software module clustering techniques: A review article,” in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation, KBEI 2017*, 2018.

[2] N. Mottaghi and M. R. Keyvanpour, “Test suite reduction using data mining techniques: A review article,” in *18th CSI International Symposium on Computer Science and Software Engineering, CSSE 2017*, 2018.

[3] S. Kashefi Gargari and M. R. Keyvanpour, “Comparative Analytical Survey on SBST Challenges from the Perspective of the Test Techniques,” *International Journal of Information and Communication Technology Research*, vol. 14, no. 2, pp. 32–40, Jun. 2022.

[4] M. R. Keyvanpour, H. Homayouni, H. Shirazi, and H. Shirazee, “Automatic software test case generation: An analytical classification framework,” *International Journal of Software Engineering and Its Applications*, vol. 6, no. 4, pp. 1-6, Oct. 2012.

[5] M. Rocha, A. Simão, T. Sousa, and M. Batista, “Test case generation by EFSM extracted from UML sequence diagrams,” in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2019.

[6] M. L. Mohd-Shafie, W. M. N. W. Kadir, H. Lichter, M. Khatibsyarbini, and M. A. Isa, “Model-based test case generation and prioritization: a systematic literature review,” *Softw Syst Model*, vol. 21, no. 2, 2022.

[7] M. L. Mohd-Shafie, W. M. N. W. Kadir, M. Khatibsyarbini, M. A. Isa, I. Ghani, and H. Ruslai, “An EFSM-Based Test Data Generation Approach in Model-Based Testing,” *Computers, Materials and Continua*, vol. 71, no. 2, 2022.

[8] “OMG Unified Modeling Language TM (OMG UML),” 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5

[9] M. Chen, P. Mishra, and D. Kalita, “Coverage-driven automatic test generation for UML activity diagrams,” in *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, 2008.

[10] F. A. D. Teixeira and G. Braga E Silva, “Easytest: An approach for automatic test cases generation from UML activity diagrams,” in *Advances in Intelligent Systems and Computing*, Springer Verlag, 2018, pp. 411–417.

[11] M. Rocha, A. Simão, and T. Sousa, “Model-based test case generation from UML sequence diagrams using extended finite state machines,” *Software Quality Journal*, vol. 29, no. 3, 2021.

[12] R. Yang, Z. Chen, Z. Zhang, and B. Xu, “EFSM-Based Test Case Generation: Sequence, Data, and Oracle,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 4, 2015.

[13] J. Zhang, R. Yang, Z. Chen, Z. Zhao, and B. Xu, “Automated EFSM-based test case generation with scatter search,” in *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings*, 2012.

[14] M. L. Shoemaker, *UML Applied: A .NET Perspective*. *Apress*, 2004.

[15] A. Hettab, A. Chaoui, M. Boubakir, and E. Kerkouche, “Automatic scenario-oriented test case generation from UML activity diagrams: a graph transformation and simulation approach,” *International Journal of Computer Aided Engineering and Technology*, vol. 16, no. 3, pp. 379–415, 2022.

[16] Meiliana, I. Septian, R. S. Alianto, Daniel, and F. L. Gaol, “Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm,” in *Procedia Computer Science*, 2017.

[17] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres, “Model-based testing using UML activity diagrams: A systematic mapping study,” *Computer Science Review*, vol. 33. 2019.

[18] X. Zhou, R. Zhao, and F. You, “EFSM-based test data generation with Multi-Population Genetic Algorithm,” in *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, 2014.

[19] V. Rechtberger, M. Bures, and B. S. Ahmed, “Overview of Test Coverage Criteria for Test Case Generation from Finite State Machines Modelled as Directed Graphs,” in *Proceedings - 2022 IEEE 14th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2022*, 2022.

[20] A. Elyasaf, E. Farchi, O. Margalit, G. Weiss, and Y. Weiss, “Generalized Coverage Criteria for Combinatorial Sequence Testing,” *IEEE Transactions on Software Engineering*, May. 2023.

[21] M. Blackburn, R. Busser, and A. Nauman, “Why Model-Based Test Automation is Different and What You Should Know to Get Started,” in *Proceedings of the International Conference on Practical Software Quality and Testing*, Washington, USA, 2004.